

CS412 Fuzzing Lab Report

Mahdi Atallah (346571), Rassene M'Sadaa (341919),
Yannik Krone (347243), Yonathan Lanzmann (342738)

May 2025

Introduction

For the fuzzing lab, we chose the LIBPNG project <https://github.com/pnggroup/libpng>. It is the official reference library for handling PNG (Portable Network Graphics) image files. Written in the C programming language, it provides a platform-independent set of functions for reading, writing, and manipulating PNG images. Libpng supports almost all features of the PNG format, including transparency, gamma correction, and color profiles. It relies on the zlib library for compression and decompression tasks.

1 Part

1.1 Setup and execution

LIBPNG currently provides only one fuzzing harness: `libpng_read_fuzzer`. Due to a recent commit that introduced a bug (<https://github.com/pnggroup/libpng/issues/678>), we run an additional command in our `run.w_corpus.sh`, which modifies the Dockerfile located in `oss-fuzz/projects/libpng/Dockerfile` to clone the latest release tag, `v1.6.48`:

```
git clone git@github.com:google/oss-fuzz.git && cd oss-fuzz

sudo sed -i 's|git clone --depth 1 https://github.com/pnggroup/libpng.
    git|git clone --branch v1.6.48 --depth 1 https://github.com/pnggroup/
    libpng.git|' projects/libpng/Dockerfile

sudo python3 infra/helper.py build_image libpng
sudo python3 infra/helper.py build_fuzzers libpng

sudo mkdir build/out/w_corpus
sudo timeout 4h python3 infra/helper.py run_fuzzer libpng
    libpng_read_fuzzer --corpus-dir build/out/w_corpus
```

The diff file, located at `submission/part1/oss-fuzz.diff`, adds a custom `build.sh` script that's identical to the one in the libpng repository, except we've commented out the lines responsible for loading the seeds.

In addition, we modified the Dockerfile to ignore the build script from the libpng repo and instead execute the custom version located in the current directory with `sed` as in the first part.

Therefore, to run without seeds do the following commands (make sure to run these commands on the same level as `oss-fuzz.diff` and to apply it from scratch — i.e., in a clean environment where the effects of previous script `run.w_corpus.sh` are not present):

```
git clone git@github.com:google/oss-fuzz.git && cd oss-fuzz

sudo sed -i 's|git clone --depth 1 https://github.com/pnggroup/libpng.git|git clone --branch v1.6.48 --depth 1 https://github.com/pnggroup/libpng.git|' projects/libpng/Dockerfile

sudo git apply ../oss-fuzz.diff

sudo python3 infra/helper.py build_image libpng
sudo python3 infra/helper.py build_fuzzers libpng

sudo mkdir build/out/w_o_corpus
sudo timeout 4h python3 infra/helper.py run_fuzzer libpng
libpng_read_fuzzer --corpus-dir build/out/w_o_corpus
```

1.2 Observations

Detailed coverage analysis with and without seed corpus

The impact of a seed corpus on fuzzing effectiveness is significantly demonstrated by the coverage metrics obtained from two separate fuzzing sessions of the `libpng_read_fuzzer`. The detailed HTML coverage reports reveal profound differences between the two runs, emphasizing the role of seed corpora in steering fuzzing engines into richer and deeper code paths from the start.

PATH	LINE COVERAGE	FUNCTION COVERAGE	REGION COVERAGE
contrib/	63.79% (74/116)	100.00% (5/5)	39.71% (83/209)
png.c	26.57% (452/1701)	31.82% (21/66)	28.38% (411/1448)
pngerror.c	34.47% (131/380)	48.15% (13/27)	29.77% (92/309)
pngget.c	3.50% (26/742)	2.78% (2/72)	3.00% (26/866)
pngmem.c	72.07% (80/111)	76.92% (10/13)	67.31% (70/104)
pngread.c	8.37% (188/2247)	18.42% (7/38)	7.44% (163/2190)
pngrio.c	78.57% (11/14)	100.00% (2/2)	70.00% (7/10)
pngtran.c	8.18% (278/3397)	27.08% (13/48)	10.97% (235/2143)
pngutil.c	55.31% (1442/2607)	72.88% (43/59)	23.85% (1600/6709)
pngset.c	43.18% (478/1107)	42.86% (21/49)	47.56% (449/944)
pngtrans.c	4.10% (17/415)	9.52% (2/21)	4.87% (15/308)
TOTALS	24.75% (3177/12837)	34.75% (139/400)	20.68% (3151/15240)

(a) Without seed corpus

PATH	LINE COVERAGE	FUNCTION COVERAGE	REGION COVERAGE
contrib/	87.93% (102/116)	100.00% (5/5)	59.33% (124/209)
png.c	43.33% (737/1701)	57.58% (38/66)	42.89% (621/1448)
pngerror.c	55.26% (210/380)	70.37% (19/27)	51.13% (158/309)
pngget.c	3.50% (26/742)	2.78% (2/72)	3.00% (26/866)
pngmem.c	79.28% (88/111)	84.62% (11/13)	75.96% (79/104)
pngread.c	28.26% (635/2247)	47.37% (18/38)	27.76% (608/2190)
pngrio.c	78.57% (11/14)	100.00% (2/2)	70.00% (7/10)
pngtran.c	30.17% (1025/3397)	52.08% (25/48)	32.38% (694/2143)
pngutil.c	73.34% (1912/2607)	93.22% (55/59)	29.13% (1954/6709)
pngset.c	53.03% (587/1107)	48.98% (24/49)	54.13% (511/944)
pngtrans.c	8.67% (36/415)	19.05% (4/21)	12.01% (37/308)
TOTALS	41.82% (5369/12837)	50.75% (203/400)	31.62% (4819/15240)

(b) With seed corpus

Figure 1: Coverage after 4 hours of fuzzing `libpng_read_fuzzer` with and without a seed corpus.

Running the harness with an initial corpus considerably increased the line coverage, function coverage, and region coverage. With the seeds, line coverage reached 41.82%, function coverage reached 50.7%, and region coverage attained 31.62%. Without seeds, these metrics were much lower—24.75% line coverage, 34.75% function coverage, and 20.68% region coverage. The initial corpus thus enables more efficient exploration, reaching more functionalities within the same time frame.

Functions and APIs covered by `libpng_read_fuzzer`

The harness with seeds thoroughly exercised core reading functions such as `png_create_read_struct`, `png_destroy_read_struct`, and primary reading pipelines (`png_read_info`, `png_read_image`, and `png_read_end`). Key memory allocation and error-handling routines like `png_malloc`, `png_malloc_warn`, and `png_error` were also consistently triggered, confirming the fuzzer’s effectiveness at exploring libpng’s robustness. Transformation and chunk-handling functionalities (`pngset.c`, `pngrtran.c`, `pngrutil.c`) were partially covered. The harness successfully invoked basic operations like IHDR setting, simple palette expansion, transparency handling, and basic error checks. However, more sophisticated pathways, including advanced gamma corrections, specialized color transformations, and complex chunk handling (such as sCAL and pCAL), were largely unexplored, suggesting potential improvements for seed diversity. Meta-data retrieval functions (`pngget.c`) and specialized ASCII-to-floating-point conversions (`png.c`) received minimal coverage. This indicates that these areas are not currently triggered by typical fuzzing inputs, highlighting a limitation of the existing harness and corpus.

Unreachable and poorly covered areas

Advanced features and rarely-used chunk types (e.g., sCAL, pCAL) remain virtually untouched by the current harness setup. Additionally, advanced pixel-level transformations and detailed meta-data querying functionalities were poorly covered, suggesting the harness might require targeted improvements.

2 Part

2.1 Uncovered region 1

A number of high-level PNG decoding functions provided by libpng are not exercised by the existing fuzzing harness. These include functions such as `png_read_image` and especially `png_read_png`, both of which are designed to abstract away the multi-step decoding process into simpler, higher-level calls. These functions internally invoke multiple operations—such as `png_read_info`, various transformation settings via `png_set_*`, row processing or `png_read_image`, and finalization with `png_read_end`—in a single entry point. In particular, `png_read_png` is a widely used convenience function that performs a full decoding pipeline with optional transformations. It is commonly used in real-world software and libraries that prioritize ease of use and full image decoding in one step, such as ImageMagick, SDL_image, wxWidgets, and many embedded image viewers or GUI frameworks.

Despite its popularity and practical importance, `png_read_png` is not invoked in the current harness (`libpng_read_fuzzer.cc`). As a result, many internal libpng routines that are only triggered through this function—including default transformation paths, chunk handling logic, and cleanup sequences—are entirely untested under fuzzing conditions. This is a significant omission because high-level API functions typically encapsulate complex internal behaviors and interactions between libpng subsystems. Bugs and memory safety issues that may arise from the combined execution of multiple steps can remain undetected when those steps are only exercised in isolation. In contrast, `png_read_png` simulates realistic usage scenarios where decoding is performed in an all-in-one call. The lack of coverage for these high-level decoding functions is likely due to a focus on a few

API functions only, notably `png_read_row`. However, this design choice comes at the expense of overlooking the behavior of other higher-level APIs that are extensively relied upon in production environments, as will be further discussed in Section 2.3.

2.2 Uncovered region 2

The second uncovered region is the entire PNG writing interface, including functions such as `png_create_write_struct`, `png_write_info`, `png_write_row`, and `png_write_end`. The existing fuzzing harness (`libpng_read_fuzzer.cc`) is solely focused on decoding PNG images and entirely omits coverage of libpng’s writing functionalities. As a result, a significant portion of the library’s logic remains untested, a limitation we further analyze in Section 2.3. The writing process in libpng is not a trivial inversion of reading; it involves distinct internal pathways, including chunk assembly, stream output handling, and optional image transformations that are specific to encoding. The PNG writing interface manages several critical tasks:

- Constructing and serializing standard and ancillary PNG chunks
- Compressing image data using zlib and applying filtering heuristics
- Generating and verifying CRC values for output integrity
- Handling transformations such as palette generation, grayscale-to-RGB conversion, and bit-depth normalization

Failures in this logic can lead to security issues such as buffer overflows, malformed output files, or silent data corruption—none of which can be uncovered by a read-only harness. These risks are especially relevant in software that dynamically generates or exports PNG images, including image editors (e.g., GIMP, Krita), rendering engines, medical imaging systems, scientific applications, and mobile or web apps that offer screenshot or export features. Moreover, certain chunks and encoding scenarios are only encountered during PNG creation (e.g., the inclusion of text metadata, gamma correction tags, or time stamps), meaning they will never be exercised unless the writing path is explicitly tested. Without such coverage, errors in rarely used or optional features may go undetected for long time. The current lack of any dedicated fuzzer for the write API creates a blind spot in libpng’s testing surface. By extending fuzzing to the writing interface, it becomes possible to verify the correctness and robustness of file generation across a wider variety of inputs, including edge cases in compression, image dimensions, and chunk layout that could otherwise compromise the integrity of generated files or introduce inconsistencies between readers and writers.

2.3 Shortcoming of existing harnesses

To link our findings above with the existing fuzzer implementation, we analyzed the only available harness in OSS-Fuzz for libpng: `libpng_read_fuzzer.cc`. This harness is solely designed for decoding and exercises only a limited subset of libpng’s API. The core part of the fuzzing logic is the following sequence of calls, which sets up interlacing, updates internal structures, allocates row buffers, and reads image rows manually using `png_read_row`:

```
188 int passes = png_set_interlace_handling(png_handler.png_ptr);  
189 png_read_update_info(png_handler.png_ptr, png_handler.info_ptr);  
190 png_handler.row_ptr = png_malloc(
```

```

191     png_handler.png_ptr, png_get_rowbytes(png_handler.png_ptr,
192                                           png_handler.info_ptr));
193 for (int pass = 0; pass < passes; ++pass) {
194     for (png_uint_32 y = 0; y < height; ++y) {
195         png_read_row(png_handler.png_ptr,
196                     static_cast<png_bytep>(png_handler.row_ptr), nullptr);
197     }
198 }
199 png_read_end(png_handler.png_ptr, png_handler.end_info_ptr);

```

The harness exercises key libpng functions involved in reading PNG images. It begins with `png_read_info`, which reads the image header and prepares internal structures. Next, `png_read_update_info` updates these structures based on transformations and interlacing. The main image data is read row by row using `png_read_row`, allowing manual processing of each scanline. Finally, `png_read_end` completes the read operation by processing any remaining data and finalizing the PNG read process. These functions are the main targets (which are only read functions) of the existing harness and are only very few out of all functions available in the API. This explains the uncovered regions mentioned above (Sections 2.1, 2.2).

3 Part

In this section, we chose to create forks of the two projects `oss_fuzz` and `libPng`. Our goal was to modify the `oss_fuzz` dockerfile so that it would clone our `libPng` fork. In our fork, we established several branches, with the primary one being `stable-v1.6.48`. This branch is based on the commit prior to the discovery of the bug and serves as the main branch for the latest update. Additionally, each team member worked on separate branches to enhance the individual harnesses.

3.1 Improved coverage for region 1

Harness implementation

The fuzzing harness for libpng was extended to cover additional portions of the libpng API, specifically focusing on the function `png_read_png`. This extension is designed to increase the coverage of potential vulnerabilities in the library, particularly in areas that were not previously tested with the original setup. Below is the implementation of the modified harness.

```

210     if (size < kPngHeaderSize || png_sig_cmp(data, 0, kPngHeaderSize)) {
211         return 0;
212     }
213
214     png_structp png_ptr2 = png_create_read_struct(PNG_LIBPNG_VER_STRING,
215                                                    nullptr, nullptr, nullptr);
216     png_infop info_ptr2 = png_create_info_struct(png_ptr2);
217     if (!png_ptr2 || !info_ptr2) {
218         if (png_ptr2) png_destroy_read_struct(&png_ptr2, nullptr, nullptr);
219         if (info_ptr2) png_destroy_info_struct(png_ptr2, &info_ptr2);
220         return 0;
221     }

```

```

221
222     png_set_mem_fn(png_ptr2, nullptr, limited_malloc, default_free);
223
224     BufState* buf_state2 = new BufState();
225     buf_state2->data = data + kPngHeaderSize;
226     buf_state2->bytes_left = size - kPngHeaderSize;
227     png_set_read_fn(png_ptr2, buf_state2, user_read_data);
228     png_set_sig_bytes(png_ptr2, kPngHeaderSize);
229
230     if (setjmp(png_jmpbuf(png_ptr2))) {
231         png_destroy_read_struct(&png_ptr2, &info_ptr2, nullptr);
232         delete buf_state2;
233         return 0;
234     }
235
236     int transforms = PNG_TRANSFORM_EXPAND | PNG_TRANSFORM_GRAY_TO_RGB |
237                     PNG_TRANSFORM_PACKING | PNG_TRANSFORM_SCALE_16 |
238                     PNG_TRANSFORM_STRIP_ALPHA;
239
240     png_read_png(png_ptr2, info_ptr2, transforms, nullptr);
241
242     png_destroy_read_struct(&png_ptr2, &info_ptr2, nullptr);
243     delete buf_state2;

```

Coverage results

The overall coverage did not significantly improve following the harness modification. Most files showed minimal or no change in line, function, or region coverage. The only noticeable increase was observed in `pngtrans.c`, where line coverage rose by approximately 10% (from 18.57% to 25.51%) and region coverage increased by a similar margin (from 12.01% to 26.73%). This low increase is relatively expected, as the harness was extended to call only one additional function: `png_read_png`. While this change did not drastically expand coverage across the codebase, it did reach an important part of the library. The `png_read_png` function is a high-level API that internally exercises various transformation and decoding routines, making it a meaningful addition for testing purposes. It is worth noting that the total reported coverage appears worse than that of the original harness. However, this is due to the inclusion of additional files in the report, specifically write-related files such as `pngwrite.c`, `pngwtran.c`, and `pngwutil.c`, which were not exercised by either harness and thus show 0% coverage. These files were not present in the original report, which partially explains the drop in overall percentages. In summary, although the raw coverage numbers resemble those of the original harness and overall coverage metrics appear slightly lower, the extended harness successfully reaches an important high-level API function that was previously uncovered. This improves the depth and relevance of the fuzzing campaign despite limited numeric gains.

3.2 Improved coverage for region 2

Harness implementation

Region 2 of libpng’s write pipeline—implemented across `pngwrite.c`, `pngwtran.c`, and `pngwutil.c`—was largely unexercised by the original write harness. To address this, we developed a new fuzz target, `libpng_write_fuzzer.cc`, which systematically drives virtually every API in the write path:

- **PNG setup/teardown:** Calls to `png_create_write_struct`, `png_create_info_struct`, and `setjmp(png_jmpbuf)` are used to initialize libpng with custom no-op error/warning handlers. Resources are freed via `png_destroy_write_struct`.
- **Custom I/O configuration:** An in-memory write callback is registered via `png_set_write_fn`, allowing output to a `std::vector<uint8_t>`.
- **Image header and compression settings:** The image is configured using `png_set_IHDR` with fuzzer-controlled width and height (clamped to 128×128), along with fixed values for bit depth (8) and color type (`PNG_COLOR_TYPE_RGBA`). Compression level and strategy are set via `png_set_compression_level` and `png_set_compression_strategy`, while filtering and interlacing are configured using `png_set_filter` and `png_set_interlace_handling`.
- **Ancillary chunks (conditionally included):**
 - `tIME`: Modification time via `png_set_tIME` if enough data is available.
 - `gAMA`: Gamma correction applied using `png_set_gAMA`.
 - `sRGB`: Embedded profile via `png_set_sRGB`.
 - `cHRM`: Static chromaticity settings via `png_set_cHRM_fixed`.
 - `tEXt`: Optional text comment using `png_set_text`.
 - `pHYs`: Physical resolution using `png_set_pHYs`.
- **Unknown-chunk injection:** Arbitrary 4-byte chunk names and up to 32 bytes of payload are extracted from input and injected using `png_set_unknown_chunks` and `png_set_unknown_chunk_location`.
- **Image data emission:** A row buffer is initialized via a helper class, and pixel data is populated from input. Writing is performed using either `png_write_image` (bulk) or `png_write_row` (per-row), followed by finalization with `png_write_end`.

In `build.sh` we compile and install the new harness alongside the read fuzzer:

```
$CXX $CXXFLAGS -std=c++11 -I. \  
  $SRC/libpng/contrib/oss-fuzz/libpng_write_fuzzer.cc \  
  -o $OUT/libpng_write_fuzzer \  
  -lFuzzingEngine .libs/libpng16.a -lz
```

To seed this harness, we wrote `generate_write_seeds.py`, which:

1. Recursively locates all `*.png` under `$SRC/libpng` (excluding crashers).

2. Clamps each image to at most 128×128 pixels and serializes it to a [width] [height] [RGBA...] binary.
3. Generates synthetic pattern seeds (uniform, checkerboard, gradients, noise) at power-of-two sizes.
4. Packages all .bin seeds into libpng_write_fuzzer_seed_corpus.zip, nesting them under write_seed_bins/.

We invoke it in build.sh:

```
# run our generator script (requires Pillow)
python3 $SRC/libpng/contrib/oss-fuzz/generate_write_seeds.py
cp $SRC/libpng/contrib/oss-fuzz/libpng_write_fuzzer_seed_corpus.zip \
  $OUT/libpng_write_fuzzer_seed_corpus.zip
```

and ensure Pillow is installed in the Docker image.

Coverage Results

PATH	LINE COVERAGE	FUNCTION COVERAGE	REGION COVERAGE	PATH	LINE COVERAGE	FUNCTION COVERAGE	REGION COVERAGE
contrib/	87.93% (102/116)	100.00% (5/5)	59.33% (124/209)	contrib/	96.88% (124/128)	83.33% (5/6)	96.36% (106/110)
png.c	43.25% (737/1704)	56.72% (38/67)	42.82% (620/1448)	png.c	14.20% (242/1704)	20.90% (14/67)	14.36% (208/1448)
pngerror.c	54.06% (213/394)	70.37% (19/27)	50.00% (161/322)	pngerror.c	11.68% (46/394)	22.22% (6/27)	10.87% (35/322)
pngget.c	3.47% (26/749)	2.78% (2/72)	2.99% (26/870)	pngget.c	0.53% (4/749)	1.39% (1/72)	0.57% (5/870)
pngmem.c	79.28% (88/111)	84.62% (11/13)	75.96% (79/104)	pngmem.c	72.07% (80/111)	76.92% (10/13)	64.42% (67/104)
pngread.c	28.36% (638/2250)	47.37% (18/38)	27.80% (609/2191)	pngread.c	0.00% (0/2250)	0.00% (0/38)	0.00% (0/2191)
pngrio.c	65.38% (17/26)	100.00% (2/2)	61.54% (8/13)	pngrio.c	0.00% (0/26)	0.00% (0/2)	0.00% (0/13)
pngtran.c	30.17% (1025/3397)	52.08% (25/48)	32.38% (694/2143)	pngtran.c	0.00% (0/3397)	0.00% (0/48)	0.00% (0/2143)
pngutil.c	73.31% (1912/2608)	93.22% (55/59)	29.12% (1953/6707)	pngutil.c	0.00% (0/2608)	0.00% (0/59)	0.00% (0/6707)
pngset.c	51.63% (587/1137)	48.98% (24/49)	53.01% (511/964)	pngset.c	19.70% (224/1137)	26.53% (13/49)	20.75% (200/964)
pngtrans.c	8.88% (39/439)	19.05% (4/21)	11.64% (37/318)	pngtrans.c	2.05% (9/439)	4.76% (1/21)	2.52% (8/318)
pngwio.c	0.00% (0/30)	0.00% (0/3)	0.00% (0/16)	pngwio.c	56.67% (17/30)	66.67% (2/3)	50.00% (8/16)
pngwrite.c	0.00% (0/1324)	0.00% (0/42)	0.00% (0/1145)	pngwrite.c	30.29% (401/1324)	30.95% (13/42)	29.26% (335/1145)
pngwtran.c	0.00% (0/404)	0.00% (0/5)	0.00% (0/203)	pngwtran.c	8.91% (36/404)	20.00% (1/5)	11.82% (24/203)
pngwutil.c	0.00% (0/1618)	0.00% (0/54)	0.00% (0/1424)	pngwutil.c	50.43% (816/1618)	62.96% (34/54)	46.28% (659/1424)
TOTALS	33.02% (5384/16307)	40.20% (203/505)	26.67% (4822/18077)	TOTALS	12.25% (1999/16319)	19.76% (100/506)	9.21% (1655/17978)

(a) Baseline read harness (libpng_read_fuzzer) (b) Improved write harness (libpng_write_fuzzer) with seed corpus)

Figure 2: Coverage comparison between the baseline read harness and our write-focused fuzzer.

The newly developed fuzzer explicitly targets Region 2—the write-side API of libpng—which had previously received no coverage at all. This is clearly reflected in the table: prior to our contribution, files such as pngwio.c, pngwrite.c, pngwtran.c, and pngwutil.c each had 0% line, function, and region coverage. With the addition of our write harness, these files now achieve meaningful coverage. pngwio.c improved from 0% to 56.67% line coverage and 66.67% function coverage. pngwrite.c rose from 0% to 30.29% in line coverage and 30.95% in function coverage. Similarly, pngwtran.c increased to 8.91% line coverage and 20.00% function coverage, while pngwutil.c reached 50.43% and 62.96% respectively. These gains validate that our harness successfully exercises key components of the previously untested write path, including chunk construction, filtering logic, and compression routines.

These gains validate that our harness exercises significant portions of the previously untouched encoding pipeline, including chunk writing, filter configuration, and compression logic. Naturally, this write-focused fuzzer does not target read-related modules such as `pngread.c`, `pngtran.c`, and `pngrio.c`, which explains their 0% coverage in the second image. This trade-off is expected and highlights the complementary nature of targeted fuzzers. Therefore, combining this new write harness with the improved read harness described earlier would allow comprehensive coverage across both decoding and encoding paths in libpng. Together, they form a strong foundation for full-surface fuzz testing of the library's API.

3.3 Fuzzer execution

There are two separate run files available for cloning a new oss-fuzz project into the directory. If an existing oss-fuzz project is already located in the directory, it should be removed prior to executing either of these run files. This run files will fuzz with the needed harness for 4 hours. For the improved read harness `libpng_read_fuzzer.cc`:

```
git clone --branch submit_improve1 --single-branch git@github.com:
    Enessar/oss-fuzz.git
cd oss-fuzz

sudo python3 infra/helper.py build_image libpng
sudo python3 infra/helper.py build_fuzzers libpng

# Run the fuzzer with the seed corpus
sudo mkdir build/out/read_seed
sudo timeout 4h python3 infra/helper.py run_fuzzer libpng
    libpng_read_fuzzer --corpus-dir build/out/read_seed
```

For the new Write harness `libpng_write_fuzzer.cc`:

```
git clone --branch submit_improve2 --single-branch git@github.com:
    Enessar/oss-fuzz.git
cd oss-fuzz

sudo python3 infra/helper.py build_image libpng
sudo python3 infra/helper.py build_fuzzers libpng

# Run the fuzzer with the seed corpus
sudo mkdir build/out/write_seed
sudo timeout 4h python3 infra/helper.py run_fuzzer libpng
    libpng_write_fuzzer --corpus-dir build/out/write_seed
```

4 Part

In this part of the assignment, our objective was to triage a crash uncovered through fuzzing. Although we did not discover a new crash ourselves, we focused on reproducing a previously reported memory corruption vulnerability. We wondered what would happen with images having larger dimensions, as the images used in our harness were relatively small. After reviewing issues

reported on GitHub, we found that libpng version 1.5.20 contained a known bug related to the handling of large interlaced images (see CVE-2014-9495 [1]), which could cause heap buffer overflows and potentially arbitrary code execution.

To explore this, we attempted to reproduce the crash by generating a large Adam7 interlaced PNG image with an unusually wide dimension. During our initial trials, our custom image generator produced malformed PNGs due to incorrect encoding of interlaced data. Surprisingly, this led us to observe a heap buffer overflow flagged by AddressSanitizer during execution of the `png_write_row` function. This unintended behavior revealed shortcomings in libpng’s input validation: the API accepted structurally inconsistent inputs and failed to catch mismatches between declared image parameters and internal memory expectations. While this was not the original CVE trigger, our attempt to reproduce it surfaced related robustness issues in the write pipeline.

4.1 Root cause of the bug

The bug is rooted in incorrect buffer size calculations in the `png_write_row` function when handling Adam7 interlaced images. The Adam7 interlacing method divides the image into 7 passes, each with different widths and heights. When generating an interlaced image, it is critical that the row buffers for each pass are correctly allocated based on the computed pass width and pass height. Incorrect logic in the application (or potentially the library itself in vulnerable versions) can lead to:

- Incorrect row buffer allocation: The program may allocate a buffer that is too small for the interlaced row being processed, causing a heap buffer overflow when data is copied into the buffer.
- Interlacing-specific issues: The vulnerability seems to be most prominent when processing images with larger widths. In such cases, incorrect logic in the allocation of row buffers can cause the program to allocate insufficient memory for the image row buffers, triggering a heap overflow when the rows are written.

The overflow occurs when the program attempts to write an incorrectly sized row buffer. The issue arises specifically in `pngwrite.c`, where the following line is executed:

```
760 ...  
761 /* Copy user's row into buffer, leaving room for filter byte. */  
762 png_memcpy(png_ptr->row_buf + 1, row, row_info.rowbytes);  
763 ...
```

In cases where `row_info.rowbytes` exceeds the size of the buffer pointed to by `png_ptr->row_buf`, this line causes a heap buffer overflow. This typically arises in interlaced images where the size of decompressed row data can vary unexpectedly, especially under malformed inputs. The absence of sufficient bounds checking and proper allocation logic in the presence of complex image layouts, such as those generated by Adam7 interlacing, makes this a subtle yet impactful vulnerability.

4.2 Proposed fix

To fix this issue, we need to ensure that the calculated row size is checked before attempting to write the row data. Here’s the approach to fix it:

- **Validate row buffer size:** Before writing any row, we must check that the calculated buffer size (`rowbytes`) is valid and matches the allocated buffer size. If it doesn't, we should raise an error or return early to prevent a buffer overflow.
- **Bounds checking:** Add bounds checking to verify that the calculated row width (pass width \times bytes per pixel) is within the limits of a 32-bit signed integer and doesn't overflow.
- **Error handling:** In case the buffer size calculation is invalid or the row buffer is too small, the function should fail gracefully by returning an error instead of attempting to write the row data.

Here's a basic example of how the fix could be implemented in the `png_write_row` function:

```

760 ...
761 if (row_info.rowbytes <= 0 || row_info.rowbytes > png_ptr->row_buf_size)
762     {
763         png_error(png_ptr, "Row buffer size mismatch: calculated size is too
764             large or invalid");
765     }
766     /* Copy user's row into buffer, leaving room for filter byte. */
767     png_memcpy(png_ptr->row_buf + 1, row, row_info.rowbytes);
768     ...

```

This will ensure that any incorrectly sized row will be flagged and the program will not attempt to write into an insufficiently allocated buffer.

4.3 Severity

The heap buffer overflow in the `png_write_row` function arises when an application supplies a malformed Adam7 interlaced image with incorrectly sized row buffers, leading libpng to write more data than was allocated. This vulnerability is particularly severe because many applications rely on libpng to handle PNG image writing and may not validate interlaced image data properly. While this bug requires a malformed image—specifically one with improperly sized rows for certain interlacing passes—it remains realistically triggerable in production environments where input image data originates from untrusted or partially validated sources, such as user uploads or network streams. The malformed input can cause libpng to overflow its allocated heap buffers during the row writing process, leading to heap corruption and program instability. From an exploitability perspective, this overflow provides an attacker with opportunities to corrupt heap metadata or adjacent memory, potentially overwriting function pointers or other critical control structures. Such corruption could allow for arbitrary code execution, information leakage, or denial of service through targeted crashes. Although modern memory safety mitigations make exploitation more challenging, the underlying bug remains a significant risk, especially in security-sensitive contexts where image data processing is exposed to hostile inputs. Given the widespread deployment of libpng in many applications, this vulnerability was relevant (note that the issue does not appear in modern versions).

4.4 Steps to reproduce

To reproduce the crash locally, execute the following commands:

```
cd part4
chmod +x run.poc.sh
./run.poc.sh
```

Conclusion

This lab provided valuable insights into libpng’s fuzzing surface and highlighted key limitations in the existing OSS-Fuzz harness. As shown in Section 2.3, the default harness only targets few high-level APIs (Section 2.1) and leaves the entire writing interface (Section 2.2) untested.

To address this, we developed two targeted fuzzers: one that invokes `png_read_png` to better emulate real-world decoding usage, and another that thoroughly exercises the write pipeline. Although the vulnerability we examined in Section 4 was not discovered directly through our fuzzing, it resides in the write-side code path—specifically in `png_write_row`—and thus reinforces the relevance of extending fuzzing coverage to this part of the API.

Overall, this assignment reinforced the importance of realistic, API-aware fuzzing and offered hands-on experience with vulnerability discovery, triage, and remediation.

References

- [1] National Vulnerability Database, *CVE-2014-9495*, <https://nvd.nist.gov/vuln/detail/CVE-2014-9495>, Accessed May 2025.