# CS-412 Fuzzing Lab Report

Tsvetan Kolev (404190), Anton Park (342166),
Ariel Pelayo (311424), Yago Pérez Pérez (339637)

May 15, 2025

## Introduction

For the fuzzing lab, we chose the LIBPNG project `https://github.com/pnggroup/libpng`, the official PNG reference library. Our fork of the OSS-Fuzz repository is located at `https://github.com/yperez-ZzzZz/cs412-oss-fuzz`, and the fork of the LIBPNG library with our changes can be found at `https://github.com/yperez-ZzzZz/libpng-better`.

## 1 Part 1

### 1.1 Running the fuzzer with and without a seed corpus

To run the default fuzzer, execute

```
sudo bash run.w_corpus.sh
```

in an otherwise empty folder.

To run the fuzzer without a corpus, execute

```
sudo bash run.w_o_corpus.sh
```

in an otherwise empty folder.

The scripts can be found in the `part1/` subfolder. Make sure that no other HTTP server is running on port 8008 to see the final coverage report.

The script clones our libpng fork and the OSS-Fuzz repository, then creates a CLI file that simplifies the fuzzing commands, builds the image and the fuzzer, runs the fuzzer and displays the coverage report.

The diff files are empty because our script deletes the corpus `.zip` file after building and before running the fuzzer. This method successfully runs the fuzzer without the corpus, and it is simpler than modifying the build script.

### 1.2 Discussion

Table 1 shows the difference in coverage between the two runs.

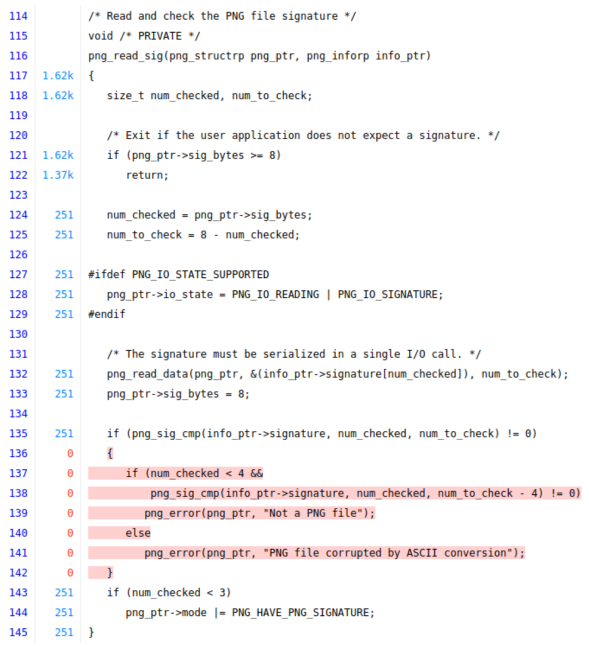
We observe that fuzzing with an initial seed corpus achieves higher coverage than fuzzing with an empty seed corpus within the same time frame.

Some code regions were reached by both methods alike. For example, all functions were covered in the `/src/libpng/pngrio.c` file in both cases. This file helps read inputs to the library, and its key function, `png_set_read_fn()`, is called directly at the beginning of the harness using the `data` input given to `LLVMFuzzerTestOneInput()`, and other predefined parameters (lines 144 to 147 of the `contrib/oss-fuzz/libpng_read_fuzzer.cc` file):

| | Line coverage | Function coverage | Region coverage |
|---|---|---|---|
| Default corpus | 40.87% | 48.50% | 30.84% |
| Empty corpus | 26.03% | 37.25% | 21.58% |
| Difference | -14.84% | -11.25% | -9.26% |

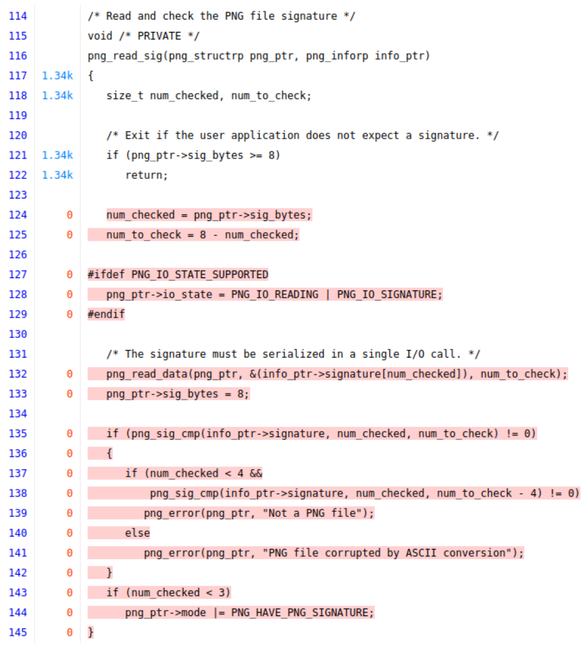
see

Table 1: Difference between the coverage with a default and empty corpus



```
        /* Read and check the PNG file signature */
        void /* PRIVATE */
        png_read_sig(png_structrp png_ptr, png_inforp info_ptr)
1.62k   {
1.62k      size_t num_checked, num_to_check;

           /* Exit if the user application does not expect a signature. */
1.62k      if (png_ptr->sig_bytes >= 8)
1.37k         return;

  251      num_checked = png_ptr->sig_bytes;
  251      num_to_check = 8 - num_checked;

  251   #ifdef PNG_IO_STATE_SUPPORTED
  251      png_ptr->io_state = PNG_IO_READING | PNG_IO_SIGNATURE;
  251   #endif

           /* The signature must be serialized in a single I/O call. */
  251      png_read_data(png_ptr, &(info_ptr->signature[num_checked]), num_to_check);
  251      png_ptr->sig_bytes = 8;

  251      if (png_sig_cmp(info_ptr->signature, num_checked, num_to_check) != 0)
    0      {
    0         if (num_checked < 4 &&
    0             png_sig_cmp(info_ptr->signature, num_checked, num_to_check - 4) != 0)
    0            png_error(png_ptr, "Not a PNG file");
    0         else
    0            png_error(png_ptr, "PNG file corrupted by ASCII conversion");
    0      }
  251      if (num_checked < 3)
  251         png_ptr->mode |= PNG_HAVE_PNG_SIGNATURE;
  251   }
```

(a) With initial seed corpus



```
        /* Read and check the PNG file signature */
        void /* PRIVATE */
        png_read_sig(png_structrp png_ptr, png_inforp info_ptr)
1.34k   {
1.34k      size_t num_checked, num_to_check;

           /* Exit if the user application does not expect a signature. */
1.34k      if (png_ptr->sig_bytes >= 8)
1.34k         return;

    0      num_checked = png_ptr->sig_bytes;
    0      num_to_check = 8 - num_checked;

    0   #ifdef PNG_IO_STATE_SUPPORTED
    0      png_ptr->io_state = PNG_IO_READING | PNG_IO_SIGNATURE;
    0   #endif

           /* The signature must be serialized in a single I/O call. */
    0      png_read_data(png_ptr, &(info_ptr->signature[num_checked]), num_to_check);
    0      png_ptr->sig_bytes = 8;

    0      if (png_sig_cmp(info_ptr->signature, num_checked, num_to_check) != 0)
    0      {
    0         if (num_checked < 4 &&
    0             png_sig_cmp(info_ptr->signature, num_checked, num_to_check - 4) != 0)
    0            png_error(png_ptr, "Not a PNG file");
    0         else
    0            png_error(png_ptr, "PNG file corrupted by ASCII conversion");
    0      }
    0      if (num_checked < 3)
    0         png_ptr->mode |= PNG_HAVE_PNG_SIGNATURE;
    0   }
```

(b) Without initial seed corpus

Figure 1: Coverage of the `png_read_sig()` function

```
png_handler.buf_state = new BufState();
png_handler.buf_state->data = data + kPngHeaderSize;
png_handler.buf_state->bytes_left = size - kPngHeaderSize;
png_set_read_fn(png_handler.png_ptr, png_handler.buf_state, user_read_data);
```

Therefore, as long as the fuzzer provides a valid input, this function will be executed. This shows that the basic functions necessary for a library, and therefore the ones that are higher on a call graph, are very likely to be fuzzed regardless of the use of a seed corpus.

An initial corpus, however, usually helps to generate a broad range of observable behaviours in the target code. It makes sense for a fuzzer with an empty corpus to take more time to reach certain code locations, since it must construct valid inputs from "nothing". It is especially true in the case of the LIBPNG library, where the inputs are PNG image that follow a predefined structure that is hard to synthesise without any knowledge of the PNG file format. [1]

Moreover, some code regions might depend on specific conditions that are hard to fulfill through mutations alone.

One example of this is the `png_read_sig()` function in the `/src/libpng/pngrutil.c` file, which reads and checks a PNG's signature. The function begins by checking for a condition and exits if it is not fulfilled. Figure 1 shows the coverage for this function, highlighting in red the uncovered lines. It can be observed that fuzzing with the initial seed corpus (figure 1a) created inputs that passed the first check, unlike fuzzing without an initial seed corpus (figure 1b), which was thus not able to cover the rest of the function.

Providing a seed corpus with a variety of inputs that reach different code regions thus speeds up coverage, which explains why in the same time frame, fuzzing without a seed corpus achieved 14.86% less line coverage than fuzzing with a seed corpus.

The maximum coverage achieved by OSS-Fuzz (the official report) for the LIBPNG library is 43.31%. Had we fuzzed for more than four hours with the initial seed corpus, we would have achieved a value closer to this. As for fuzzing without an initial seed corpus, reaching the same coverage is potentially possible, but would have required even more time. Even then, there might still be some regions that the fuzzer would only be able to reach by chance, and therefore might never be reached in a specific run, due to conditions that are hard to satisfy.

## 2 Part 2

### 2.1 Uncovered Region 1 - `png_read_png()`

The `png_read_png()` is a high-level API in libpng that abstracts and consolidates a full read of a PNG image, including header parsing, transformation application, image reading, and finalization. It internally calls lower-level functions like `png_read_info()`, `png_read_update_info()`, `png_read_image()`, and `png_read_end()`, orchestrating a complete PNG decoding pipeline in one single call.

This makes it highly relevant for:

- **Real-world usage:** Many applications and libraries rely on this single entry point rather than piecing together the lower-level read functions. While specific software application that use `png_read_png()` are not always explicitly documented, this function is commonly employed in various image processing tools, graphic editors, and applications that require straightforward PNG image loading. For instance, sample code and tutorials often demonstrate its usage for reading and writing PNG files, highlighting its practicality in handling complete image data with minimal code complexity[2]. This tutorial-driven adoption makes it particularly well-suited for educational contexts, beginner-level projects, and proof-of-concept implementations, situation where easy of use and minimal boilerplate are often prioritized over fine-grained control or robustness. Although such projects may not always emphasize strict input validation or fault tolerance, they can still find their way into production environment or be reused across codebases. This increases the importance of throughoughly fuzzing high-level convenience functions like `png_read_png()`, which are likely to act as entry points in diverse and potentially under-tested contexts.

- **Complex transformation paths:** The `png_read_png()` function encapsulates and triggers many optional transformations within the libpng pipeline—such as `PNG_TRANSFORM_GRAY_TO_RGB`, `PNG_TRANSFORM_SWAP_ENDIAN`, `PNG_TRANSFORM_INVERT_ALPHA`, and more. These are conditionally applied based on the input PNG format and user-specified flags. Because many of these flags are rarely used in isolation, fuzzing lower-level read functions often misses important edge cases and combinations. The compound nature of these transformations means that subtle bugs, misconfigurations, or missed validation logic could lie dormant until a specific flag combination or image structure is encountered, making this function a key target for comprehensive fuzz testing.

- **Safety and correctness:** This function is responsible for several memory-intensive operations, including the allocation of row pointers and image buffers, as well as invoking transformation routines and performing cleanup. Improper handling in any of these stages can lead to critical vulnerabilities such as memory corruption, buffer overflows, or resource leaks. In addition, since `png_read_png()` implicitly coordinates several state transitions inside the libpng decoder (e.g., info update, interlace handling, and final read-out), any inconsistencies or invalid states may cause hard-to-detect bugs. Notably, `png_read_png()` has experienced integer overflow issues in the past [3], where improper checks in the function led to overflows under certain conditions. Ensuring its robustness through fuzzing helps protect downstream consumers who rely on this convenience function for complete image loading workflows.

The current LibFuzzer-based harness for libpng, as shown in `libpng_read_fuzzer.cc`, exhibits several limitations that restrict its ability to exercise the full spectrum of functionality provided by the libpng library. Notably, the function `png_read_png()`, a high-level API that simplifies reading PNG images in a single call, is not being covered by this harness. The key shortcomings are outlined below:

- **Manual Parsing Workflow:** The harness manually invokes the lower-level libpng API functions such as `png_read_info()`, `png_read_row()`, and `png_read_end()` instead of using the all-in-one `png_read_png()` function. Consequently, `png_read_png()` is never reached during fuzzing, and any potential bugs or edge cases in its internal implementation remain unexplored.

- **Lack of API Diversity:** The harness narrowly exercises a single, fixed decoding pipeline with a specific set of transformation functions (e.g., `png_set_expand()`, `png_set_gray_to_rgb()`, etc.). This reduces variability in code coverage and ignores alternative pathways such as those that would be triggered via simplified APIs or less common decoding configurations.

- **No Use of `png_read_png()`:** Because `png_read_png()` internally calls multiple lower-level read functions and handles transformations automatically, it provides an essential target for fuzzing. Its omission means that bugs related to its convenience-layer abstraction are entirely missed by the current setup.

## 2.2 Uncovered Region 2 - Completing `png_image_finish_read()`

Through targeted modifications to the existing harness, we successfully exercised previously unreached branches in `png_image_finish_read()`, leading to the coverage of four internal functions: `png_image_read_colormap()`, `png_image_read_colormapped()`, `png_image_read_background()`, and `png_image_ read_composite()`. These functions are integral to the final stages of image decoding in the simplified read API and are involved in processing image palettes, background blending, and compositing pixel values. Although these routines are only invoked after initial input validation, their internal logic contains nontrivial processing paths that benefit significantly from fuzz testing. Ensuring their correctness and resilience is important for the safe and accurate decoding of PNG images, especially in environments relying on libpng's high-level APIs.

- **Complex post-processing logic:** Functions like `png_image_read_background()` and `png_image _read_composite()` perform pixel-level transformations that combine image data with background colors or apply alpha blending. These are sensitive to subtle arithmetic and memory errors, making them valuable fuzzing targets.
- **Palette handling in `png_image_read_colormap()`:** This function decodes indexed color data using a colormap. Mistakes in bounds checking or colormap application could result in out-of-bounds memory access or color misinterpretation.
- **Diverse control flow:** The path to these functions includes conditionals based on format flags, interlace modes, and color types. Fuzzing can help ensure that all logic branches are exercised and behave correctly.
- **Downstream correctness and rendering fidelity:** These functions directly influence the final pixel output of the image. Any latent bugs could cause incorrect rendering in downstream applications that rely on libpng for display or processing.
- **Memory safety in image buffer manipulation:** Despite input validation at earlier stages, these functions operate on large buffers and perform stride-based memory access. Fuzzing helps catch any incorrect assumptions or edge cases in buffer dimensions and alignment.
- **Exposure through simplified API:** Since these functions are invoked via `png_image_finish_read()`, they are accessible through the simplified API used by many developers. Their safety and stability are crucial for applications relying on this user-friendly interface.

The default `libpng_read_fuzzer.cc` harness fails to trigger key branches and functions within the simplified image reading API due to hardcoded assumptions and limited parameterization. As a result, several critical decoder paths remain untested, even though they are accessible and used by other applications or APIs that depend on `libpng`'s simplified interface. Below is a breakdown of each newly covered function, along with its significance and the reasons why it was previously unreachable:

- `png_image_read_composite()`
  - **Harness shortcoming:** The original harness fixed the output format to `PNG_FORMAT_RGBA`, preventing execution of compositing logic in linear space. The updated harness introduces variations such as `PNG_FORMAT_GRAY`, enabling this path.
  - **Usage in other software:** Used when decoding images with alpha channels into premultiplied linear RGB, often required for precise compositing in graphics applications.
- `png_image_read_background()`

  - **Harness shortcoming:** The original harness never requires the workaround to perform double gamma correction because the output format is always `PNG_FORMAT_RGBA`. The updated version triggers this function by using the grayscale format that invokes background blending.
  - **Usage in other software:** Commonly employed to composite transparent PNGs against solid backgrounds, a typical requirement in UI frameworks and image preprocessing tools.

- `png_image_read_colormap()`
  - **Harness shortcoming:** The default harness did not supply palette-based images or use the `RGBA_COLORMAP` format. The new harness explicitly sets this format and provides a colormap buffer to enable coverage.
  - **Usage in other software:** Necessary for decoding indexed-color PNGs, still prevalent in icons, favicons, and older game assets.

- `png_image_read_colormapped()`
  - **Harness shortcoming:** The previous setup never performed palette index expansion to RGB. By switching to `PNG_FORMAT_RGBA_COLORMAP`, the updated harness activates this decoding step.
  - **Usage in other software:** Part of the full image readout process for indexed-color images, particularly when transitioning from palette data to usable RGB output.

## 3 Part 3

### 3.1 Uncovered Region 1 - `png_read_png()`

#### 3.1.1 Discussion of changes

To expand code coverage in libpng, we developed a new fuzzing harness that explicitly targets the `png_read_png()` function — a high-level interface for decoding an entire PNG image, including header parsing, decompression, and optional transformations. This API was not exercised by the original fuzzers, leaving significant code regions untouched.

Our implementation begins by checking the PNG signature and rejecting oversized inputs to prevent excessive memory usage. It sets up libpng structures (`png_structp`, `png_infop`), installs a buffer-backed I/O source via `png_set_read_fn()`, and configures a custom memory allocator to safely reject overly large allocations. After calling `png_set_sig_bytes()` to account for the consumed header, the harness makes a direct call to `png_read_png()`.

To increase coverage of internal decoding paths, the harness uses the final byte of input data to generate a random combination of transformation flags, which are passed as the third argument to `png_read_png()`. These transformations include options like `PNG_TRANSFORM_STRIP_16`, `PNG_TRANSFORM_PACKING`, and `PNG_TRANSFORM_SWAP_ENDIAN`, enabling diverse code paths to be triggered based on the input. Finally, the decoded row pointers are retrieved with `png_get_rows()` to force row memory access and ensure that image decoding is not optimized away.

#### 3.1.2 How to build and run the new harness

Execute `sudo bash run.improve1.sh` in an otherwise empty folder. The script can be found in the part3/improve1 folder.

#### 3.1.3 Target Code Region & Rationale

As discussed in Part 2, the `png_read_png()` function was not previously reached by any harness. This API is widely used in practical applications and educational examples, offering a one-call interface for full PNG decoding. Since it internally calls multiple lower-level functions—including `png_read_info()`, `png_read_update_info()`, `png_read_image()`, and `png_read_end()`—it provides a valuable abstraction layer where bugs in state management, transformation logic, or memory cleanup may occur.

The original libpng harness used a manual, step-by-step decoding workflow that entirely bypassed `png_read_png()`. Consequently, any logic specific to this high-level function, or transformation behavior gated behind it, remained untested. Our modification directly addresses this gap by exercising the convenience-layer code paths, transformation logic, and coordinated state transitions that `png_read_png()` encapsulates.

#### 3.1.4 Line Coverage

To assess the effectiveness of our change, we ran both the original and new harnesses under identical conditions for 3 sets of four hours each and measured line coverage. The results are summarized below. It should be noted that minor discrepancies were observed due to varying hardware specifications across Part 1 and Part 3.

| | **Base** | **New** |
|---|---|---|
| 1 | 40.81 | 37.28 |
| 2 | 41.78 | 36.64 |
| 3 | 41.80 | 38.08 |
| Merged | 41.85 | 38.49 |

Table 2: Comparison of existing harness and our new harness.

The decrease in line coverage stems from the fact that the original LibFuzzer-based harness exercises a broader range of lower-level libpng API functions, which are not invoked in our new harness. Specifically, functions such as `png_read_row()` and manual transformation configurations are omitted in favor of calling the high-level `png_read_png()` function. As a result, some portions of the libpng codebase that are specifically triggered through the manual decoding path remain uncovered.

However, the new harness achieves its goal of significantly increasing coverage of `png_read_png()` and the internal pathways it manages. We verified through code instrumentation that this function is now being reached with a wide range of fuzzed inputs. This includes various combinations of transformation flags—randomly selected based on the last input byte—that exercise conditional logic within libpng's transformation routines (e.g., alpha inversion, endianness swapping, packing/unpacking). These paths were not previously tested, and many are only reachable when using the high-level read API. Thus, while the overall percentage is lower, the new harness provides complementary coverage, targeting an important, previously untouched part of the library that is commonly used in real-world applications. This makes it a valuable addition to the fuzzing suite.

#### 3.1.5 Key challenges and further possible improvements

There were no technical obstacles preventing access to `png_read_png()`, the original fuzzer simply omitted it in favor of manually sequencing lower-level functions. Our change demonstrates that a simple harness can reach this important function and yield coverage gains.

There are several directions for improving the new harness and further expanding its coverage:

- **Improving Transformation Logic:** Systematically generating combinations of transformation flags to explore deeper into under-tested image format conversions and memory layouts, as not all transformations within `png_read_png()` were triggered, due to missing macro definitions.
- **Hybrid Harness Design:** Combining the manual decoding path from the original harness with high-level API calls like `png_read_png()` could help retain broad coverage while still targeting convenience functions. The performance of a hybrid design is better than running two fuzzers separately, since in each cycle there are common setup and cleanup steps.

## 3.2 Uncovered Region 2 - Completing `png_image_finish_read()`

#### 3.2.1 Discussion of changes

The default `libpng_read_fuzzer.cc` primarily exercises the traditional low-level read API using the `png_read_info()` and `png_read_row()` functions. However, the simplified API introduced in recent versions of `libpng`, particularly the `png_image_*` family of functions, remained largely untested.

To address this, we restructured the fuzzer harness to focus on exercising the higher-level interface provided by `libpng`. Specifically, we added logic to invoke `png_image_begin_read_from_memory()`, set different `image.format` values, and call `png_image_finish_read()` with valid memory buffers. These changes enabled the fuzzer to traverse paths involving automatic format conversion, compositing, and colormap resolution.

### 3.2.2 How to build and run the new harness

Execute `sudo bash run.improve2.sh` in an otherwise empty folder. The script can be found in the part3/improve2 folder.

### 3.2.3 Target Code Region & Rationale

The main target was the simplified read pipeline inside `pngread.c`, specifically functions such as:

- `png_image_read_composite()`
- `png_image_read_background()`
- `png_image_read_colormap()`
- `png_image_read_colormapped()`

These functions encapsulate much of the logic typically performed manually when using the low-level API (e.g., color format conversion, gamma correction, transparency handling). They are also commonly used in high-level applications and image loaders.

By targeting these routines, we aimed to expand coverage to high-level processing logic, especially areas performing buffer allocation, blending arithmetic, and palette lookups. These are critical from a security and robustness standpoint, yet are often conditionally executed based on format flags and image metadata.

### 3.2.4 Line Coverage

The experimental setup here remained identical as to what was described for uncovered region 1.

| | **Base** | **Modified** |
|---|---|---|
| 1 | 40.81 | 53.18 |
| 2 | 41.78 | 52.47 |
| 3 | 41.80 | 52.97 |
| Merged | 41.85 | 53.21 |

Table 3: Comparison of existing harness and our modified harness.

The coverage gains from our modifications were substantial. Prior to our changes, line coverage for these code regions was zero, as the original harness never invoked `png_image_finish_read` with the formats necessary to reach our target functions. After integrating the simplified read API into the fuzz harness, we observed a substantial increase in coverage across these routines. Specifically, the call to `png_image_begin_read_from_memory()` successfully initialized the image structure, and follow-up invocations of `png_image_finish_read()` triggered the internal logic for format conversion, palette handling, and compositing which was previously guaraded by image format flags. Overall, these changes increased the fuzzer's line coverage from **41.85%** to **53.21%**. More importantly, they opened access to nuanced internal paths that perform image transformations and buffer operations—code that is both security-critical and likely to contain subtle bugs if left untested.

### 3.2.5 Key challenges and further possible improvements

One of the most significant challenges we faced during this fuzzing campaign was reverse engineering the internal logic of the simplified read API in `pngread.c` to identify the correct flags and configurations required to trigger previously unreachable code paths. The existing harness, while functional for basic fuzzer input, did not exercise the full breadth of functionality provided by the simplified read API.

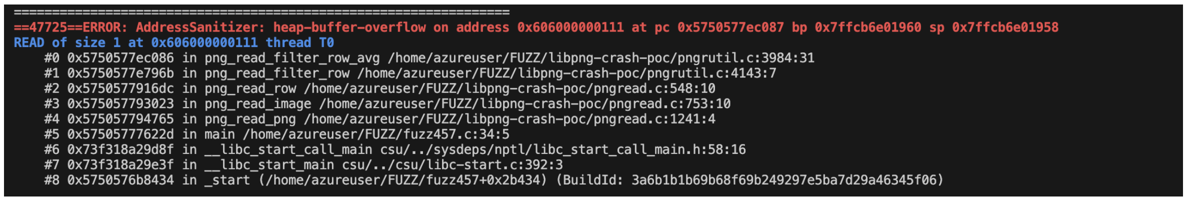
While our current modifications have increased coverage and improved the detection of potential vulnerabilities in the image decoding pipeline, there are still avenues for further improvement:

- **Increased Coverage of Complex Code Paths:** While we have made progress in reaching critical functions such as `png_image_read_composite()` and `png_image_read_colormap()`, there are still several lines within `png_image_read_background()` that have not been fully exercised. Further refinement of the fuzzer could help cover these paths. A more exhaustive exploration of the different image formats and specific parameters involved in background reading could unlock additional code paths.

## 4 Part 4

With our new improved harness from Part 3, we reached previously uncovered regions of the `png_read_png` function. Unfortunately, we were not able to trigger a bug with it. However, as mentioned in Part 2, we discovered that `png_read_png` has experienced integer overflow issues in the past. We identified an open issue on the project's GitHub issue tracker, which reports an integer overflow found by fuzzing in `png_read_png` [3] and we decided to triage this crash.

To reproduce the crash, we built the vulnerable version of libpng and ran a PoC we found in the comments of the issue. We encapsulated repository cloning, build configuration, and sanitized PoC execution within the script `run.poc.sh`, which streamlines the process. Running with AddressSanitizer our script reliably reproduces a heap buffer overread inside the row-filter routine `png_read_filter_row_avg`, which was caught by ASan, as shown in Figure 2, confirming the crash. To replicate our results, simply `chmod +x run.poc.sh` (if necessary) and run `./run.poc.sh`.



```
=================================================================
==47725==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x606000000111 at pc 0x5750577ec087 bp 0x7ffcb6e01960 sp 0x7ffcb6e01958
READ of size 1 at 0x606000000111 thread T0
    #0 0x5750577ec086 in png_read_filter_row_avg /home/azureuser/FUZZ/libpng-crash-poc/pngrutil.c:3984:31
    #1 0x5750577e796b in png_read_filter_row /home/azureuser/FUZZ/libpng-crash-poc/pngrutil.c:4143:7
    #2 0x5750577916dc in png_read_row /home/azureuser/FUZZ/libpng-crash-poc/pngread.c:548:10
    #3 0x575057793023 in png_read_image /home/azureuser/FUZZ/libpng-crash-poc/pngread.c:753:10
    #4 0x575057794765 in png_read_png /home/azureuser/FUZZ/libpng-crash-poc/pngread.c:1241:4
    #5 0x57505777622d in main /home/azureuser/FUZZ/fuzz457.c:34:5
    #6 0x73f318a29d8f in __libc_start_call_main csu/../sysdeps/nptl/libc_start_call_main.h:58:16
    #7 0x73f318a29e3f in __libc_start_main csu/../csu/libc-start.c:392:3
    #8 0x5750576b8434 in _start (/home/azureuser/FUZZ/fuzz457+0x2b434) (BuildId: 3a6b1b1b69b68f69b249297e5ba7d29a46345f06)
```

Figure 2: AddressSanitizer report showing the heap-buffer-overflow in `png_read_filter_row_avg` (pngrutil.c:3984), with the backtrace through `png_read_filter_row`, `png_read_row`, `png_read_image`, and finally `png_read_png`.

### 4.1 Root Cause Analysis

After investigating the issue, the shared PoC and the codebase, we figured out that in a libpng v1.6.37, an invalid sequence of API calls can trigger a heap-buffer-overflow during PNG decoding. The failure is observed along the internal call chain

`png_read_png` $\rightarrow$ `png_read_image` $\rightarrow$ `png_read_row` $\rightarrow$ `png_read_filter_row` $\rightarrow$ `png_read_filter_row_avg`.

Normally `png_read_row` initializes row decoding, decompresses IDAT data, and then applies the PNG filter algorithms to reconstruct pixel bytes. The crash occurs in the "average" filter (`png_read_filter_row_avg`) when it processes an unexpectedly zero-length row, causing an underflow in its loop bounds calculation.

**Triggering Condition** The PoC invokes `png_read_update_info` before the required `png_read_info` step and then continues with `png_read_png`. This is an out-of-order sequence, as one of the maintainers of the software pointed out. It calls `png_read_start_row` while the IHDR data have not yet been loaded, so `png_struct::width` is 0. Consequently `png_struct::iwidth` is initialised to 0 and remains so when `png_read_png` re-enters the decoder. Under normal settings, libpng would detect this misuse and invoke `png_error`, aborting immediately. However, the harness used in the fuzzer enabled benign-error handling via

```
png_set_benign_errors(png_ptr, PNG_BENIGN_WARNINGS_MASK);
```

which, as one of the maintainers noted, "turns the `png_error` off," demoting the fatal error to a warning and allowing decoding to proceed in an undefined state [3].

**Consequence - Overflow in `png_read_filter_row_avg`** Once the decoder is in the inconsistent state where `row_info.rowbytes = 0`, libpng applies the average filter to a non-existent scanline. The implementation of `png_read_filter_row_avg` in pngrutil.c is shown below:

```

static void
png_read_filter_row_avg(png_row_infop row_info, png_bytep row,
    png_const_bytep prev_row)
{
   size_t i;
   png_bytep rp = row;
   png_const_bytep pp = prev_row;
   unsigned int bpp = (row_info->pixel_depth + 7) >> 3;
   size_t istop = row_info->rowbytes - bpp;

   for (i = 0; i < bpp; i++)
   {
      *rp = (png_byte)(((int)(*rp) +
         ((int)(*pp++) / 2 )) & 0xff);

      rp++;
   }

   for (i = 0; i < istop; i++)
   {
      *rp = (png_byte)(((int)(*rp) +
         (int)(*pp++ + *(rp-bpp)) / 2 ) & 0xff);

      rp++;
   }
}
}
```

In a valid $32 \times 32$ paletted image (`bpp = 1`, `rowbytes = 32`) we have `istop = 31`; the two loops touch exactly 32 bytes. In the faulty state, however, `rowbytes = 0` while `bpp = 1`, hence

$$\texttt{istop} = 0 - 1 \;=\; 2^w - 1 = \texttt{SIZE\_MAX}.$$

The first loop already writes one byte past the one-byte buffer that was allocated for the filter-type byte (`rowbytes + 1`). The second loop then iterates almost $2^w$ times, rapidly advancing `rp` and `pp` far beyond the bounds of `row` and `prev_row`, producing a out-of-bounds reads and writes until the process crashes (see Figure 2). Thus, the arithmetic underflow in `istop` made possible only because `iwidth` and `rowbytes` were silently allowed to be zero is the immediate cause of the heap corruption, while the duplicate API initialisation combined with benign-error mode is the underlying root cause for the crash.

A possible remediation for this vulnerability would be to add defensive checks so that libpng will error out instead of continuing when critical initialization hasn't occurred. This is exactly what one of the maintainers proposed :

```

diff --git a/pngrutil.c b/pngrutil.c
index 068ab193a..5af3d14d2 100644
--- a/pngrutil.c
+++ b/pngrutil.c
@@ -4410,6 +4410,13 @@ png_read_start_row(png_structrp png_ptr)

    png_debug(1, "in png_read_start_row");

+   /* This is a really bad app programming error; we can't set up the data
+    * to read a row until the IHDR has been read (because that tells us the
+    * PNG data format).
+    */
+   if ((png_ptr->mode & PNG_HAVE_IHDR) == 0)
+      png_error(png_ptr, "PNG header must be read before reading rows");
+
 #ifdef PNG_READ_TRANSFORMS_SUPPORTED
    png_init_read_transformations(png_ptr);
 #endif
```

With the patch, a `png_error happens` in `png_read_start_row` if the IHDR has not been read, causing a controlled abort with an error message rather than letting execution proceed into the filters [3].

## 4.2 Security Implication

In practice, this issue is unlikely to be exploitable in normal PNG-processing applications, because it requires an unusual misuse of the API and the libpng's benign errors mode need to be explicitly enabled – a combination that typical PNG-decoding workflows never employ. An attacker cannot directly trigger this vulnerability with a malformed PNG file alone – the crash arises only if the host application makes libpng calls in the wrong sequence. Essentially, the trigger is poor application logic rather than a specific image content. An attacker would need to target a specific program that is known to call the libpng API incorrectly. For example, an application would have to call a PNG read-start function twice or skip the initial `png_read_info()` step and still proceed to read image data. Well written programs follow libpng's required call sequence, so under correct usage this bug does not occur.

If the bug is somehow reached, it results in a heap-based buffer overflow inside the PNG decoding logic. This constitutes a memory corruption vulnerability – the immediate outcome is a crash (denial of service), as the process reads/writes into invalid heap memory and triggers an abort. Beyond simply crashing the application, such out-of-bounds memory access can potentially be leveraged by a determined adversary to compromise security. An out-of-bounds read might expose adjacent memory content, creating an information leak, and an out-of-bounds write could overwrite nearby data or control structures in memory. In this case, the overflow writes a sequence of bytes derived from image data past the end of the buffer, rather than an arbitrary "write-what-where" scenario, which makes exploitation more challenging. Nevertheless, with crafted input and a vulnerable application, there is a theoretical risk that the attacker could corrupt critical program data or metadata on the heap. In the worst case, such memory corruption might be manipulated to achieve remote code execution. However, given the uncontrolled nature of the overflow and the uncommon prerequisites to reach it, reliable exploitation beyond a denial-of-service remains highly unlikely. The primary real-world risk is therefore that a malicious PNG input might crash the decoder if it runs in an application that incorrectly uses the libpng API, while more severe outcomes are only a remote possibility.

The fact that this bug only occurs when benign-error warnings are enabled in combination of a misuse of the the API mitigates its severity. Still, one of the maintainers of the software pointed out that application errors are not benign and a patch has been already developed and will roll out into the 1.8 version of libpng. [3]

**Supply-chain angle (comparison with the XZ Utils back-door).** A possible attack scenario we thought of is the following: An adversary who controls the build pipeline of a PNG-processing application could inject a small patch or build the application in a way that enables `png_set_benign_errors` and calls `png_read_update_info` before `png_read_png`, reproducing the vulnerable state inside libpng. Or even if an adversary does not have control over the build pipeline, the attacker might achieve that by social engineering such permissions. This mirrors the February 2024 XZ Utils compromise, where a malicious contributor spent nearly two years obtaining commit rights and then hid an obfuscated RCE payload in `liblzma`, ultimately enabling remote code execution in OpenSSH. [4]
However, the payoff here is far smaller. Even with the misuse wired in, an attacker only gains a heap-buffer-overflow that is hard to steer towards code execution and most often results in a denial-of-service. By contrast, the XZ back-door granted reliable pre-authentication code execution in OpenSSH. Consequently we rate the likelihood of a real-world supply-chain exploit for this libpng issue as low: the attacker would expend effort comparable to the XZ operation for, at best, a crash primitive rather than full takeover.

From a security standpoint, this issue is a Denial-of-Service vulnerability with very limited scope. It underscores the importance of handling even "benign" errors carefully – had libpng not offered the option to ignore certain errors, this overflow wouldn't have occurred. The fix makes libpng safer against improper usage, closing the hole that allowed a memory corruption. In practice, end-users of libpng are unlikely to be affected unless they are using a custom or naive PNG handling loop. The corrective patch restores fail-fast semantics - any attempt to process a zero-length row is now rejected from the beginning. Implementing such defensive checks ensures that logical API errors cannot cascade into exploitable vulnerabilities.

## References

[1] A. Herrera, H. Gunadi, S. Magrath, M. Norrish, M. Payer, and A. L. Hosking, "Seed selection for successful fuzzing," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 230–243. [Online]. Available: https://doi.org/10.1145/3460319.3464795

[2] "Use libpng to read and write png file — carvecode.net," https://carvecode.net/use-libpng-to-read-and-write-png-file/, [Accessed 07-05-2025].

[3] "An integer overflow happened in png_read_png-png_read_filter_row_avg · Issue 457 · pnggroup/libpng — github.com," https://github.com/pnggroup/libpng/issues/457, [Accessed 10-05-2025].

[4] Akamai Security Intelligence Group, "Xz utils backdoor — everything you need to know, and what you can do," https://www.akamai.com/blog/security-research/critical-linux-backdoor-xz-utils-discovered-what-to-know, Apr. 2024, [Accessed 15-05-2025].