CS412 Fuzzing Lab

Iman Attia(387653), Rodrigue de Guerre(327683),
Ahyoung Seo(390238), Nadine Alfadelraad(395802)

April 2025

1 Introduction

Fuzzing is a software testing technique that involves feeding unexpected or random inputs into a program to discover potential vulnerabilities and bugs. In this lab, we aim to evaluate and improve the fuzzing harnesses provided for open-source projects integrated with Google's OSS-Fuzz infrastructure. Our chosen project for this assignment is **libpng**.

2 Part 1: Running Existing Fuzzing Harnesses

2.1 Project Setup

We selected **libpng** as our target project. The project was cloned from:

- OSS-Fuzz repository: `https://github.com/google/oss-fuzz/tree/master/projects/libpng`
- Libpng repository: `https://github.com/pnggroup/libpng`

The initial fuzzing harness we focused on is `libpng_read_fuzzer` (the only existing one).

2.2 Commands Used

The following steps were performed to build and run the fuzzer, first with a seed corpus:

1. Clone the `oss-fuzz` repo:

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

2. Build the docker image and fuzzers:

```
python3 infra/helper.py build_image --pull libpng
python3 infra/helper.py build_fuzzers libpng
```

3. Run fuzzer WITH a seed corpus:

```
# 4h = 14400sec
python3 infra/helper.py run_fuzzer libpng libpng_read_fuzzer --corpus-dir build/out/corpus
     -- -max_total_time=14400
```

4. Generate coverage WITH a seed corpus:

```
# Build fuzzers with coverage instrumentation
python3 infra/helper.py build_fuzzers --sanitizer coverage libpng
# Generate coverage report
python3 infra/helper.py coverage libpng --corpus-dir build/out/corpus
                                        --fuzz-target libpng_read_fuzzerr
```

And then with an empty seed corpus. For this we just changed the first step:

1. Clone our fork of the `oss-fuzz` repo:

```
git clone --branch original-without-seeds --single-branch https://github.com/ImanEPFL/oss-
    fuzz.git && cd oss-fuzz
```

2.3 Modifications

To fuzz without initial seeds, we modified the Dockerfile for the libpng project on our fork of the OSS-fuzz repo:

```
FROM gcr.io/oss-fuzz-base/base-builder
RUN apt-get update && \
    apt-get install -y make autoconf automake libtool zlib1g-dev

RUN git clone --depth 1 https://github.com/madler/zlib.git
# RUN git clone --depth 1 https://github.com/pnggroup/libpng.git
RUN git clone --branch original-without-seeds --single-branch --depth 1 https://github.com/
    ahzero7d1/libpng.git
RUN cp libpng/contrib/oss-fuzz/build.sh $SRC
WORKDIR libpng
```

This enables us to clone our own fork of the libpng repository, where we modified `libpng/contrib/oss-fuzz/build.sh` by commenting out the following lines responsible for creating the seed corpus by aggregating all the png files included in the libpng project folders:

```
# find $SRC/libpng -name "*.png" | grep -v crashers | \
# xargs zip $OUT/libpng_read_fuzzer_seed_corpus.zip
```

A `project.diff` and a `oss-fuzz.diff` files were generated to reflect these changes.

2.4 Coverage Reports

Coverage reports were generated for both scenarios:

- With corpus: located in `part1/report/w_corpus/`
- Without corpus: located in `part1/report/w_o_corpus/`

Running the test for 4 hours, the fuzzer with an initial seed corpus showed 40.87% line coverage, 48.5% function coverage, 30.83% region coverage. In contrary, the fuzzer without an initial seed corpus showed 24.29% line coverage, 34% function coverage and 20.34% region coverage. Note that we can observe less coverage with the fuzzing without initial seed corpus.

2.5 Discussion on Coverage

Using an initial seed corpus provides the fuzzer with valid PNG file structures including PNG file signiture from the start, allowing it to explore deeper code paths faster. This is especially important for the case of fuzzing `libpng` since it is a library that processes PNG format files thus the fuzzing harness will not forward the random bytes if it does not satisfy the structure of PNG. Without an initial corpus, the fuzzer needs to discover valid inputs by pure mutation, which results in significantly lower initial coverage. The difference in line coverage clearly shows the importance of a good seed corpus in improving fuzzer effectiveness.
Our coverage results confirm this:

Mode	**Line Coverage**	**Function Coverage**	**Region Coverage**
With corpus	40.87%	48.50%	30.83%
Without corpus	24.29%	34.00%	20.34%

3 Part 2: Analyzing Existing Fuzzing Harnesses

3.1 Uncovered Region 1: Improving PNG Read Code Regions

Uncovered Functions:

- size == 0 conditional branch in `png_handle_iCCP`
- Functions in `pngget.c`
- Functions in `pngtrans.c`

Functionality

ICC profiles are used to determine how colors are interpreted on devices so that they are properly and consistently rendered. The mentioned branch is the entry point that calls into zlib to decompress the `iCCP` chunk of the image, fills the `profile_header` buffer, reads `profile_length`, and invokes the length checks. Covering this branch is important because it indicates that we've driven the decompressor far enough to remove the header-check logic. Fuzzing the very first "profile header" is critical to make sure that neither buffer-overflows nor logic bugs that are present, as ICC-profile parsing has previously been a source of many vulnerabilities.

In addition, functions in `pngget.c` access image metadata (e.g., width, height, bit depth...) and check the presence and the validity of specific chunks like the `iCCP` chunk. Without these getter functions, applications won't have a proper way of accessing information of PNG files they downloaded.

Finally, there are also functions in `pngtrans.c` which hold the transformation logic of the image after it has been read. These functions are important because they determine how pixel data is interpreted, rearaanged, or adjusted to meet the desired output format.

Usage in Real Software

Although not all PNG images have ICC profiles, many professional applications and software prioritize color accuracy. For example, there are image editing softwares, graphic design applications (e.g., Photoshop, GIMP), and print pipelines that rely heavily on the functionality of the ICC handling functions when such profiles are present in images. Common uses also include color space conversions between profiles like AdobeRGB, sRGB, or CMYK. Moreover, we also have modern PDF rendering pipelines and browser engines (e.g., FireFox and Chrome) that actively parse ICC profiles to ensure consistent color reproduction across devices.

Functions in `pngget.c` are used by image viewers and editors to access and edit image metadata, and to correctly interpret the image properties.

On the other hand, `pngtrans.c` functions are also used by browsers and rendering engines like Firefox, Chromium, and PDF viewers to deliver consistent and uniform visuals.

Reason for Lack of Coverage

The default seed corpus that the fuzzer uses is `libpng_read_fuzzer_seed_corpus.zip` which is generated using the `build.sh` script and placed in `build/out/libpng` directory after the fuzzer is built. This current seed corpus does not include these embedded ICC profiles in any of the PNG images, and specifically the `iCCP` chunk that stores color management metadata. Although the current read harness can process `iCCP` chunks, but it doesn't do so unless these chunks are included in the input sample. In addition, the function `png_handle_iCCP` does not look at the `profile_header` until it has taken at least 132 bytes of compressed ICC data. So, small PNGs will never enter the if (size == 0) block. As a result, the conditional branch of `png_handle_iCCP` is never reached during fuzzing runs. This means that there are blind spots in the fuzzer's coverage and that potential bugs or vulnerabilities in the ICC profile handling logic may go unnoticed.

Functions in pngget.c and pngtrans.c are not executed because they are not explicitly called by the current `libpng_read_fuzzer.cc`. Functions like `png_get_text(...)`, `png_get_y_offset_pixels(...)`, `png_set_bgr(...)`, and `png_set_swap(...)` remain uncovered. Moreover, the API functions in the harness do not implicitly call these functions so they are not triggered neither explicitly nor implicitly. As a result, many of the functions in both files are not reached during fuzzing.

3.2 Uncovered Region 2: PNG Writing Functions

Uncovered Functions:

- `png_write_image(...)`
- `png_write_info(...)`
- `png_write_row(...)`

Functionality

These functions are present in `pngwrite.c` and they are critical to writing valid PNG files, covering both metadata and actual pixel data. First, `png_write_info` is responsible for writing the initial chunks of the PNG file, which contain essential metadata about the image. This includes the `IHDR` (Image Header) chunk, which specifies the image's dimensions, color type, bit depth, compression method, filter method, and interlace method. It also handles writing other crucial ancillary chunks like `PLTE` (palette), `tRNS` (transparency), `bKGD` (background color), `tIME` (last modification time), and text chunks (`tEXt`, `zTXt`, `iTXt`) if they are present in the png info structure.

Second, `png_write_image` writes the rows of given image data. If the image is not interlaced, the image shall be written in a single pass writing the core pixel data of the image. It relies on the information written by png write info to correctly format and write the image data chunks.

Finally, the function `png_write_row` writes a single row of pixel data to the PNG output stream and it must be called for each row in the image. This function handles many tasks like row data filtering (using one of the PNG filtering types), compressing it using zlib, and adding it on the end of the image's IDAT blocks. Overall, this function is used to construct valid PNG images row-by-row, particularly when saving interlaced images or when memory constraints require streaming the data instead of saving the entire image in a single pass.

Usage in Real Software

The mentioned functions are widely used among softwares that use libpng to write PNGs, whether it's for UI graphics, scientific plots, screenshots, or camera output. For example, graphics and image editing applications like GIMP use libpng to export or save images in PNG format. In addition to scientific and visualization software like MATLAB and OpenCV use libpng to export plots, simulations, or processed images as PNGs. They often depend on `png_write_image()` when the image is held in memory as a complete buffer.

Reason for Lack of Coverage

Although these two write functions have a central role in PNG generation yet they remain uncovered by the `OSS-Fuzz` Introspector report. This region is uncovered because the default `OSS-Fuzz` harness for libpng focus only on reading and decoding PNG images (e.g, `libpng_read_fuzzer.cc`). The `OSS-Fuzz` repository for libpng does not have a fuzzer that targets the PNG write API, which means that functions like `png_write_image`, `png_write_info`, and `png_write_row` cannot be reached with the current fuzzing infrastructure. These are essential for writing valid PNG files. To elaborate, The `png_write_info` is important as without a correctly written information header, a PNG reader cannot properly interpret the image data that follows. Errors in this function can lead to an unreadable or incorrectly displayed image. Additionally, fuzzing `png_write_image` identifies vulnerabilities in the handling of pixel data, scan-line processing, filtering, and interaction with the compression library (zlib), which are all critical for generating a correct and secure PNG file.

4 Part 3: Improving the Existing Fuzzers

4.1 Overview

To improve the limitation of existing harness described in part 2, in part 3, we introduced two main improvements. First, covering **libpng** write functions by writing a new libpng write harness, and second improving the harness and seed corpus of read fuzzer to cover previously uncovered region and functions worth fuzzing. All of the coverage report and merged output corpus results of 3 runs are included under *part3/improve1*, and *part3/improve2* path of the submitted repository.

4.2 Improvement 1: Libpng Write Harness

4.2.1 Discussion of Change

- **Description:** This improvement introduces a dedicated fuzzing harness targeting the writing functionalities of libpng, particularly focusing on the functions `png_write_info` and `png_write_image` located in `pngwrite.c`.
- **Rationale:** Prior to this improvement, OSS-Fuzz's coverage for libpng focused primarily on reading functionality. Coverage analysis revealed **zero coverage** for key functions in the PNG writing pipeline, including `png_write_info` `png_write_image`, and `png_write_row`.

4.2.2 Harness Implementation:

As the `png_read_fuzzer.cc` followed the libpng book chapter 13, we tried to do the same and follow the libpng book chapter 15, about the write functionality. The harness initializes a `png_struct` and `png_info` for writing, sets up a custom in-memory write function to capture output, and uses the fuzzer input to derive parameters such as image dimensions, bit depth, color type, interlace method, and optional chunks like gAMA, bKGD, tIME, tEXt, PLTE, and tRNS. To ensure robustness and memory safety during fuzzing, like in the original read harness, we employ a limited memory allocator and safeguard the flow with setjmp/longjmp for error recovery. This comprehensive setup allows the fuzzer to explore diverse code paths in libpng's write functionality while avoiding excessive memory usage or invalid configurations.

4.2.3 How to build and run the write harness

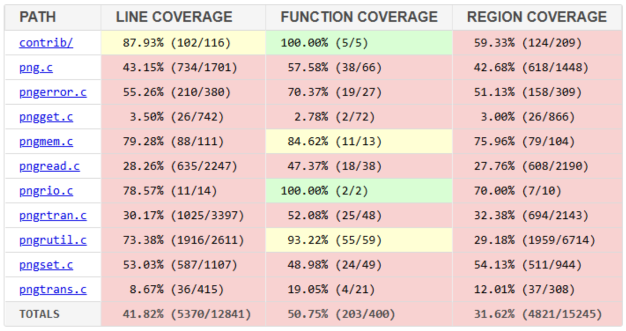
To build and run the write harness, use the provided script:

```
$ chomd +x run.improve1.sh
$ ./run.improve1.sh
```

Also, if you want to build and run step by step with your own commands, then please pay attention to the changes applied in the project.diff and the oss-fuzz.diff files found under *part3/improve1.*

4.2.4 Difference in line coverage: Original VS. Improvement 1

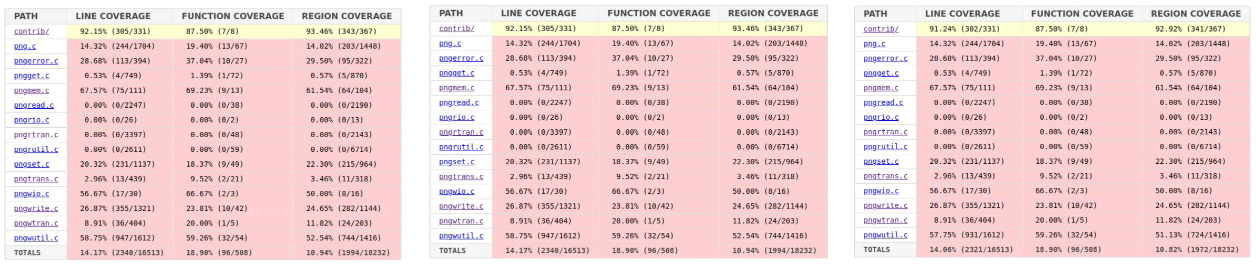
- **Before:** No coverage recorded in `png_write_info` or `png_write_image`. Both regions were untouched by previous read fuzzer, confirming the lack of PNG write-path fuzzing.



PATH	LINE COVERAGE	FUNCTION COVERAGE	REGION COVERAGE
contrib/	87.93% (102/116)	100.00% (5/5)	59.33% (124/209)
png.c	43.15% (734/1701)	57.58% (38/66)	42.68% (618/1448)
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)
pngrtran.c	30.17% (1025/3397)	52.08% (25/48)	32.38% (694/2143)
pngrutil.c	73.38% (1916/2611)	93.22% (55/59)	29.18% (1959/6714)
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% (5370/12841)	50.75% (203/400)	31.62% (4821/15245)

Figure 1: Original results without any modifications

- **After:** Post-improvement coverage reports confirm execution of significant blocks in `pngwrite.c`, notably inside the `png_write_info`, `png_write_image` and `png_write_row` functions. This includes paths responsible for writing the IHDR chunk and the IDAT data stream. To achieve that, we ran the fuzzer 3 times for 4h each and then created the average report using the 3 corpuses of the 3 runs.



PATH	LINE COVERAGE	FUNCTION COVERAGE	REGION COVERAGE
contrib/	92.15% (305/331)	87.50% (7/8)	93.46% (343/367)
png.c	14.32% (244/1704)	19.40% (13/67)	14.02% (203/1448)
pngerror.c	28.68% (113/394)	37.04% (10/27)	29.50% (95/322)
pngget.c	0.53% (4/749)	1.39% (1/72)	0.57% (5/870)
pngmem.c	67.57% (75/111)	69.23% (9/13)	61.54% (64/104)
pngread.c	0.00% (0/2247)	0.00% (0/38)	0.00% (0/2190)
pngrio.c	0.00% (0/26)	0.00% (0/2)	0.00% (0/13)
pngrtran.c	0.00% (0/3397)	0.00% (0/48)	0.00% (0/2143)
pngrutil.c	0.00% (0/2611)	0.00% (0/59)	0.00% (0/6714)
pngset.c	20.32% (231/1137)	18.37% (9/49)	22.30% (215/964)
pngtrans.c	2.96% (13/439)	9.52% (2/21)	3.46% (11/318)
pngwio.c	56.67% (17/30)	66.67% (2/3)	50.00% (8/16)
pngwrite.c	26.87% (355/1321)	23.81% (10/42)	24.65% (282/1144)
pngwtran.c	8.91% (36/404)	20.00% (1/5)	11.82% (24/203)
pngwutil.c	58.75% (947/1612)	59.26% (32/54)	52.54% (744/1416)
TOTALS	14.17% (2340/16513)	18.90% (96/508)	10.94% (1994/18232)

(a) First Run

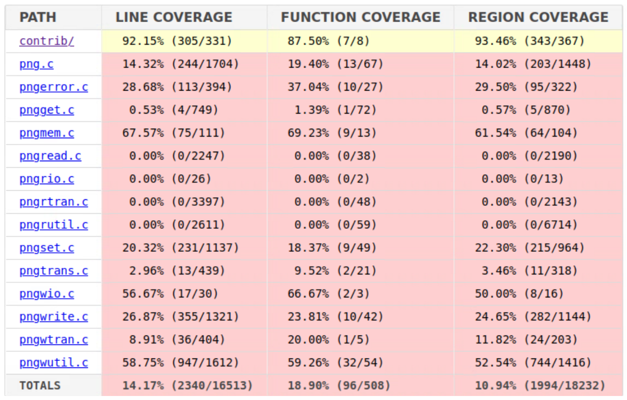
PATH	LINE COVERAGE	FUNCTION COVERAGE	REGION COVERAGE
contrib/	92.15% (305/331)	87.50% (7/8)	93.46% (343/367)
png.c	14.32% (244/1704)	19.40% (13/67)	14.02% (203/1448)
pngerror.c	28.68% (113/394)	37.04% (10/27)	29.50% (95/322)
pngget.c	0.53% (4/749)	1.39% (1/72)	0.57% (5/870)
pngmem.c	67.57% (75/111)	69.23% (9/13)	61.54% (64/104)
pngread.c	0.00% (0/2247)	0.00% (0/38)	0.00% (0/2190)
pngrio.c	0.00% (0/26)	0.00% (0/2)	0.00% (0/13)
pngrtran.c	0.00% (0/3397)	0.00% (0/48)	0.00% (0/2143)
pngrutil.c	0.00% (0/2611)	0.00% (0/59)	0.00% (0/6714)
pngset.c	20.32% (231/1137)	18.37% (9/49)	22.30% (215/964)
pngtrans.c	2.96% (13/439)	9.52% (2/21)	3.46% (11/318)
pngwio.c	56.67% (17/30)	66.67% (2/3)	50.00% (8/16)
pngwrite.c	26.87% (355/1321)	23.81% (10/42)	24.65% (282/1144)
pngwtran.c	8.91% (36/404)	20.00% (1/5)	11.82% (24/203)
pngwutil.c	58.75% (947/1612)	59.26% (32/54)	52.54% (744/1416)
TOTALS	14.17% (2340/16513)	18.90% (96/508)	10.94% (1994/18232)

(b) Second Run

PATH	LINE COVERAGE	FUNCTION COVERAGE	REGION COVERAGE
contrib/	91.24% (302/331)	87.50% (7/8)	92.92% (341/367)
png.c	14.32% (244/1704)	19.40% (13/67)	14.02% (203/1448)
pngerror.c	28.68% (113/394)	37.04% (10/27)	29.50% (95/322)
pngget.c	0.53% (4/749)	1.39% (1/72)	0.57% (5/870)
pngmem.c	67.57% (75/111)	69.23% (9/13)	61.54% (64/104)
pngread.c	0.00% (0/2247)	0.00% (0/38)	0.00% (0/2190)
pngrio.c	0.00% (0/26)	0.00% (0/2)	0.00% (0/13)
pngrtran.c	0.00% (0/3397)	0.00% (0/48)	0.00% (0/2143)
pngrutil.c	0.00% (0/2611)	0.00% (0/59)	0.00% (0/6714)
pngset.c	20.32% (231/1137)	18.37% (9/49)	22.30% (215/964)
pngtrans.c	2.96% (13/439)	9.52% (2/21)	3.46% (11/318)
pngwio.c	56.67% (17/30)	66.67% (2/3)	50.00% (8/16)
pngwrite.c	26.87% (355/1321)	23.81% (10/42)	24.65% (282/1144)
pngwtran.c	8.91% (36/404)	20.00% (1/5)	11.82% (24/203)
pngwutil.c	57.75% (931/1612)	59.26% (32/54)	51.13% (724/1416)
TOTALS	14.06% (2321/16513)	18.90% (96/508)	10.82% (1972/18232)

(c) Third Run

Figure 2: Run results after applying improvement 1



PATH	LINE COVERAGE	FUNCTION COVERAGE	REGION COVERAGE
contrib/	92.15% (305/331)	87.50% (7/8)	93.46% (343/367)
png.c	14.32% (244/1704)	19.40% (13/67)	14.02% (203/1448)
pngerror.c	28.68% (113/394)	37.04% (10/27)	29.50% (95/322)
pngget.c	0.53% (4/749)	1.39% (1/72)	0.57% (5/870)
pngmem.c	67.57% (75/111)	69.23% (9/13)	61.54% (64/104)
pngread.c	0.00% (0/2247)	0.00% (0/38)	0.00% (0/2190)
pngrio.c	0.00% (0/26)	0.00% (0/2)	0.00% (0/13)
pngrtran.c	0.00% (0/3397)	0.00% (0/48)	0.00% (0/2143)
pngrutil.c	0.00% (0/2611)	0.00% (0/59)	0.00% (0/6714)
pngset.c	20.32% (231/1137)	18.37% (9/49)	22.30% (215/964)
pngtrans.c	2.96% (13/439)	9.52% (2/21)	3.46% (11/318)
pngwio.c	56.67% (17/30)	66.67% (2/3)	50.00% (8/16)
pngwrite.c	26.87% (355/1321)	23.81% (10/42)	24.65% (282/1144)
pngwtran.c	8.91% (36/404)	20.00% (1/5)	11.82% (24/203)
pngwutil.c	58.75% (947/1612)	59.26% (32/54)	52.54% (744/1416)
TOTALS	14.17% (2340/16513)	18.90% (96/508)	10.94% (1994/18232)

Figure 3: Final report results after improvement 1 generated from the 3 runs corpuses

To have a better understanding of the performance we achieve using the harness, Table 1 includes only the write functionality-related source files and some dependant source files.

Thus, we can see that the true improvement is even much higher since we are achieving an increase of 28.43% in line coverage and 26.59% in Function Coverage and 27.80% in Region Coverage for the write functionality. To be more specific, Table 2 focuses on the write-related files and you can even observe better performance.

Path	**Line Coverage**	**Function Coverage**	**Region Coverage**
contrib/	92.15% (305/331)	87.50% (7/8)	93.46% (343/367)
png.c	14.32% (244/1704)	19.40% (13/67)	14.02% (203/1488)
pngerror.c	28.68% (113/394)	37.04% (10/27)	29.50% (95/322)
pngget.c	0.53% (4/749)	1.39% (1/72)	0.57% (5/870)
pngmem.c	67.57% (75/111)	69.23% (9/13)	61.54% (64/104)
pngset.c	20.32% (231/1137)	18.37% (9/49)	22.30% (215/964)
pngtrans.c	2.96% (13/439)	9.52% (2/21)	3.46% (13/375)
pngwio.c	56.67% (17/30)	66.67% (2/3)	50.00% (8/16)
pngwrite.c	26.87% (355/1321)	23.81% (10/42)	24.65% (282/1144)
pngwtran.c	8.91% (36/404)	20.00% (1/5)	11.82% (24/203)
pngwutil.c	58.75% (947/1612)	59.26% (32/54)	52.54% (744/1416)
TOTALS	28.43% (2340/8232)	26.59% (96/361)	27.80% (1994/7172)

Table 1: Code Coverage Report for Improvement 1 after excluding read related files.

Path	**Line Coverage**	**Function Coverage**	**Region Coverage**
pngwio.c	56.67% (17/30)	66.67% (2/3)	50.00% (8/16)
pngwrite.c	26.87% (355/1321)	23.81% (10/42)	24.65% (282/1144)
pngwtran.c	8.91% (36/404)	20.00% (1/5)	11.82% (24/203)
pngwutil.c	58.75% (947/1612)	59.26% (32/54)	52.54% (744/1416)
TOTALS	40.24% (1355/3367)	43.27% (45/104)	38.71% (1058/2779)

Table 2: Code Coverage Report for Improvement 1 for write files only.

4.2.5 Potential Improvements:

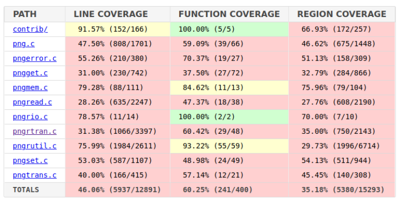
- Extend Function Coverage: While the current harness covers core writing functions such as `png_write_info`, `png_write_image`, and `png_write_row`, additional write-related functions remain only partially covered. Future harness iterations should aim to trigger more code paths.
- Increase Input Diversity: Generating a more diverse set of inputs (various bit depths, color types, compression settings,...) could help testing edge cases in the libpng write implementation and improve path exploration.
- Incorporate Error-Handling Paths: Make the harness also cover erroneous write cases, such as unsupported formats or write failures, to improve the robustness of the harness.

4.3 Improvement 2: Improving Read Harness

4.3.1 Discussion of Change

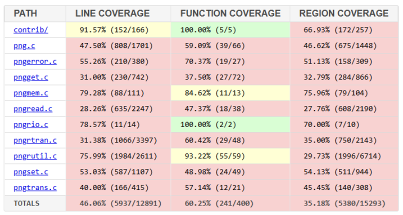
In this section, we focus on the code regions related to png read functions of libpng library.

As discussed in part 2, using the existing harness and seed corpus results in significantly low line coverage of `pngget.c` and `pngtrans.c` although there's a potential that bug can be found among the functions included in these files. Additionally, we observed that some of the `ICC(International Color Consortium)` profile related functions are not covered throughout all the read files. To improve these two shortcomings of existing read fuzzer and improve the region coverage, we introduced two main changes. First, addition of extra functions in read harness and improving seed corpus by adding more seeds.



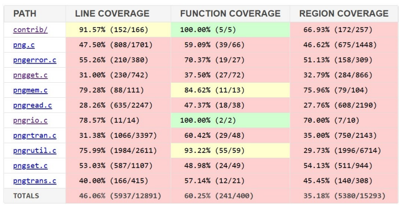
PATH	LINE COVERAGE	FUNCTION COVERAGE	REGION COVERAGE
contrib/	91.57% (152/166)	100.00% (5/5)	66.93% (172/257)
png.c	47.50% (808/1701)	59.09% (39/66)	46.62% (675/1448)
pngerror.c	55.26% (210/380)	70.37% (19/27)	51.13% (158/309)
pngget.c	31.00% (230/742)	37.50% (27/72)	32.79% (284/866)
pngmem.c	79.28% (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)
pngrtran.c	31.38% (1066/3397)	60.42% (29/48)	35.00% (750/2143)
pngrutil.c	75.99% (1984/2611)	93.22% (55/59)	29.73% (1996/6714)
pngset.c	53.03% (587/1107)	48.98% (24/49)	54.13% (511/944)
pngtrans.c	40.00% (166/415)	57.14% (12/21)	45.45% (140/308)
TOTALS	46.06% (5937/12891)	60.25% (241/400)	35.18% (5380/15293)

(a) First Run



PATH	LINE COVERAGE	FUNCTION COVERAGE	REGION COVERAGE
contrib/	91.57% (152/166)	100.00% (5/5)	66.93% (172/257)
png.c	47.50% (808/1701)	59.09% (39/66)	46.62% (675/1448)
pngerror.c	55.26% (210/380)	70.37% (19/27)	51.13% (158/309)
pngget.c	31.00% (230/742)	37.50% (27/72)	32.79% (284/866)
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)
pngrtran.c	31.38% (1066/3397)	60.42% (29/48)	35.00% (750/2143)
pngrutil.c	75.99% (1984/2611)	93.22% (55/59)	29.73% (1996/6714)
pngset.c	53.03% (587/1107)	48.98% (24/49)	54.13% (511/944)
pngtrans.c	40.00% (166/415)	57.14% (12/21)	45.45% (140/308)
TOTALS	46.06% (5937/12891)	60.25% (241/400)	35.18% (5380/15293)

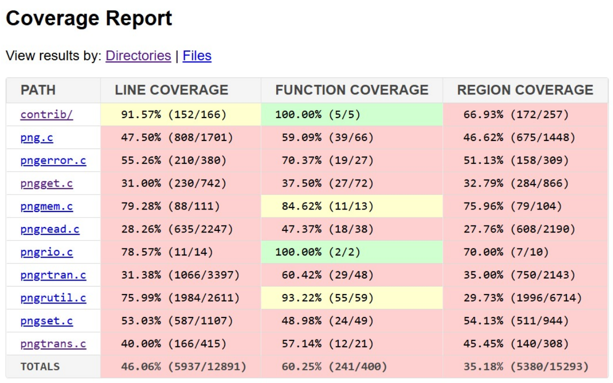
(b) Second Run



PATH	LINE COVERAGE	FUNCTION COVERAGE	REGION COVERAGE
contrib/	91.57% (152/166)	100.00% (5/5)	66.93% (172/257)
png.c	47.50% (808/1701)	59.09% (39/66)	46.62% (675/1448)
pngerror.c	55.26% (210/380)	70.37% (19/27)	51.13% (158/309)
pngget.c	31.00% (230/742)	37.50% (27/72)	32.79% (284/866)
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)
pngrtran.c	31.38% (1066/3397)	60.42% (29/48)	35.00% (750/2143)
pngrutil.c	75.99% (1984/2611)	93.22% (55/59)	29.73% (1996/6714)
pngset.c	53.03% (587/1107)	48.98% (24/49)	54.13% (511/944)
pngtrans.c	40.00% (166/415)	57.14% (12/21)	45.45% (140/308)
TOTALS	46.06% (5937/12891)	60.25% (241/400)	35.18% (5380/15293)

(c) Third Run

Figure 4: Run results after applying improvement 2



Coverage Report

View results by: Directories | Files

PATH	LINE COVERAGE	FUNCTION COVERAGE	REGION COVERAGE
contrib/	91.57% (152/166)	100.00% (5/5)	66.93% (172/257)
png.c	47.50% (808/1701)	59.09% (39/66)	46.62% (675/1448)
pngerror.c	55.26% (210/380)	70.37% (19/27)	51.13% (158/309)
pngget.c	31.00% (230/742)	37.50% (27/72)	32.79% (284/866)
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)
pngrtran.c	31.38% (1066/3397)	60.42% (29/48)	35.00% (750/2143)
pngrutil.c	75.99% (1984/2611)	93.22% (55/59)	29.73% (1996/6714)
pngset.c	53.03% (587/1107)	48.98% (24/49)	54.13% (511/944)
pngtrans.c	40.00% (166/415)	57.14% (12/21)	45.45% (140/308)
TOTALS	46.06% (5937/12891)	60.25% (241/400)	35.18% (5380/15293)

Figure 5: Final report results after improvement 2 generated from the 3 runs corpuses

4.3.2 Harness Implementation

Running fuzzer using exising harness and seed corpus showed significantly low line coverage for `pngget.c` and `pngtrans.c` files. Compared to the average line coverage percentage 41.82%, `pngget.c` showed line coverage of 3.5% and `pngtrans.c` of 8.67%. Therefore, with this first change, we targeted improving the coverage of `pngget.c` file and `pngtrans.c` file.

To improve the coverage of these regions, we improved existing libpng read harness by adding `pngget` and `pngtrans` functions in the harness explicitly. We analyzed the reason why existing harness does not cover these regions well is the functions that were included in the existing harness do not call `pngget` and `pngtrans` functions actively. Therefore, by introducing the changes, the improved harness can explicitly call the previously uncovered function and test by feeding the fuzzer input. We also used the improved seed corpus along with the improved harness to get a positive impact on improved line coverage.

4.3.3 Improved Seed Corpus

We observed that generating a seed corpus using the method of existing fuzzing build script cannot cover the `size == 0` condition of `png_handle_iCCP` function which is only ran when the input png file contains legitimate ICC profile chunk. We analyzed that this is because the existing seed corpus does not contain png seed files with legitimate ICC profile chunk. As a second change, we improved the seed corpus of fuzzer, targeting to cover ICC profile related conditional branch of `png_handle_iCCP` function.

We changed the seed corpus by combining the corpus below

- Seed corpus that the existing fuzzer build script generated. They collected all the png files included in the libpng library recursively, excluded malformed png files and fed this collection as seed corpus.
- output corpus collected after running the existing fuzzing harness
- 100 randomly generated png images

- pre-existing seed corpus from github resource

For the randomly generated png images, we used a python script that generates a png file with width and height between 16 and 256 pixels, using different color modes among grayscale, RGB, with and without alpha channel, and different pattern types among solid colors, gradients, checkerboards, noise patterns, lines, and palette-based images. This python script uses `pypng` library for image generation and we generated 100 random images to generate improved seed corpus.

We also included png corpus that is available in github, to feed more diverse png files with different sizes. This seed corpus is originally created for coverage-guided fuzzer for Go packages named go-fuzz.

4.3.4 How to build and run the improved read harness

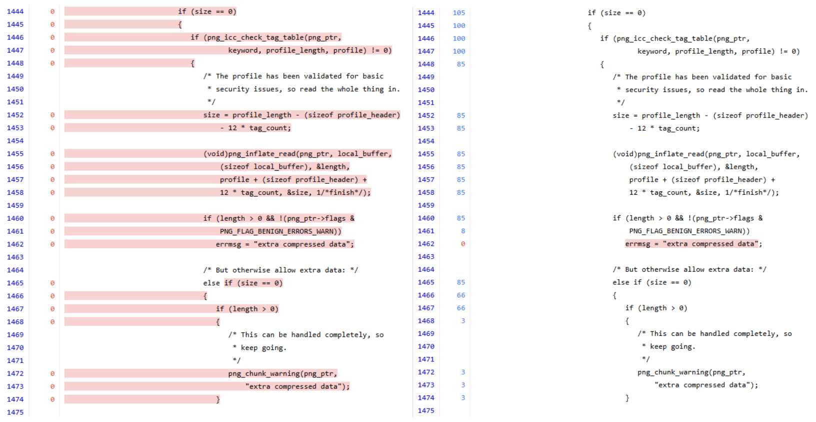
To build and run the improved read harness, use the provided script:

```
$ chomd +x run.improve2.sh
$ ./run.improve2.sh
```

4.3.5 Difference in line coverage: Original VS. Improvement 2

Compared to the existing fuzzing harness, the line coverage of pngget.c was improved from 3.5% to 31%. The line coverage of `pngtrans.c` was improved from 8.67% to 40%. Due to the significant improvement of coverage of these two regions, the overall coverage improved by 4.24%. Detailed count of the coverage, function coverage and region coverage are in figure 4 and 5.

Additionally, by enabling the fuzzer start exploring on top of the result of previous run and more diverse seeds, the fuzzer could generate wider mutation of inputs, including inputs containing proper ICC profile. This allowed the fuzzer to cover our target region, which was the `size == 0` conditional branch of `png_handle_iCCP` function in `pngrutil.c`. The screenshot of the coverage of this region is in figure 6.



```
  0      if (size == 0)
  0      {
  0         if (png_icc_check_tag_table(png_ptr,
  0                  keyword, profile_length, profile) != 0)
  0         {
              /* The profile has been validated for basic
               * security issues, so read the whole thing in.
               */
  0           size = profile_length - (sizeof profile_header)
  0               - 12 * tag_count;

  0           (void)png_inflate_read(png_ptr, local_buffer,
  0               (sizeof local_buffer), &length,
  0               profile + (sizeof profile_header) +
  0               12 * tag_count, &size, 1/*finish*/);

  0           if (length > 0 && !(png_ptr->flags &
  0               PNG_FLAG_BENIGN_ERRORS_WARN))
  0              errmsg = "extra compressed data";

              /* But otherwise allow extra data: */
  0           else if (size == 0)
  0           {
  0              if (length > 0)
  0              {
                    /* This can be handled completely, so
                     * keep going.
                     */
  0                 png_chunk_warning(png_ptr,
  0                     "extra compressed data");
  0              }
```

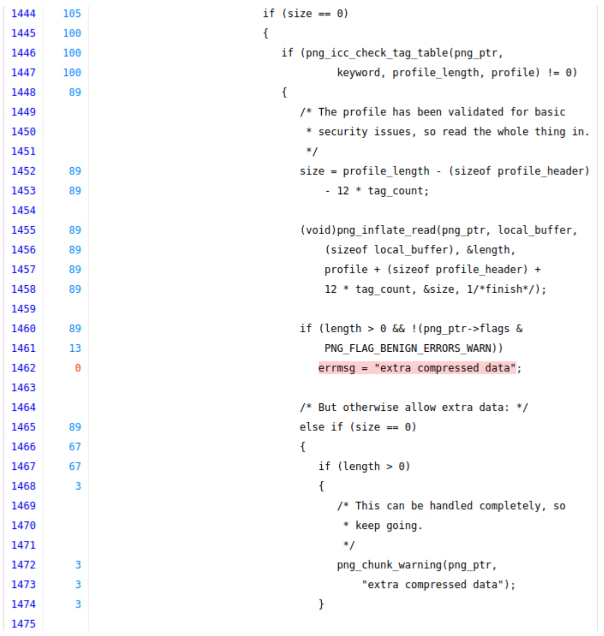
(a) Region is not covered with existing seed corpus

```
105      if (size == 0)
100      {
100         if (png_icc_check_tag_table(png_ptr,
100                  keyword, profile_length, profile) != 0)
 85         {
              /* The profile has been validated for basic
               * security issues, so read the whole thing in.
               */
 85           size = profile_length - (sizeof profile_header)
 85               - 12 * tag_count;

 85           (void)png_inflate_read(png_ptr, local_buffer,
 85               (sizeof local_buffer), &length,
 85               profile + (sizeof profile_header) +
 85               12 * tag_count, &size, 1/*finish*/);

 85           if (length > 0 && !(png_ptr->flags &
  8               PNG_FLAG_BENIGN_ERRORS_WARN))
  0              errmsg = "extra compressed data";

              /* But otherwise allow extra data: */
 85           else if (size == 0)
 66           {
 66              if (length > 0)
  3              {
                    /* This can be handled completely, so
                     * keep going.
                     */
  3                 png_chunk_warning(png_ptr,
  3                     "extra compressed data");
  3              }
```

(b) Region is covered with improved seed corpus

Figure 6: Coverage of before and after updating the seed corpus

Note that this improvement is solely due to the improvement of the seed corpus since we experimented with the non-improved harness and improved seed corpus combination. The screenshot of that case is in figure 7.



```
105                      if (size == 0)
100                      {
100                         if (png_icc_check_tag_table(png_ptr,
100                                  keyword, profile_length, profile) != 0)
 89                         {
                               /* The profile has been validated for basic
                                * security issues, so read the whole thing in.
                                */
 89                            size = profile_length - (sizeof profile_header)
 89                                - 12 * tag_count;

 89                            (void)png_inflate_read(png_ptr, local_buffer,
 89                                (sizeof local_buffer), &length,
 89                                profile + (sizeof profile_header) +
 89                                12 * tag_count, &size, 1/*finish*/);

 89                            if (length > 0 && !(png_ptr->flags &
 13                                PNG_FLAG_BENIGN_ERRORS_WARN))
  0                               errmsg = "extra compressed data";

                               /* But otherwise allow extra data: */
 89                            else if (size == 0)
 67                            {
 67                               if (length > 0)
  3                               {
                                     /* This can be handled completely, so
                                      * keep going.
                                      */
  3                                  png_chunk_warning(png_ptr,
  3                                      "extra compressed data");
  3                               }

```

Figure 7: Target region coverage is solely due to the improved seed corpus

4.3.6 Potential Improvements:

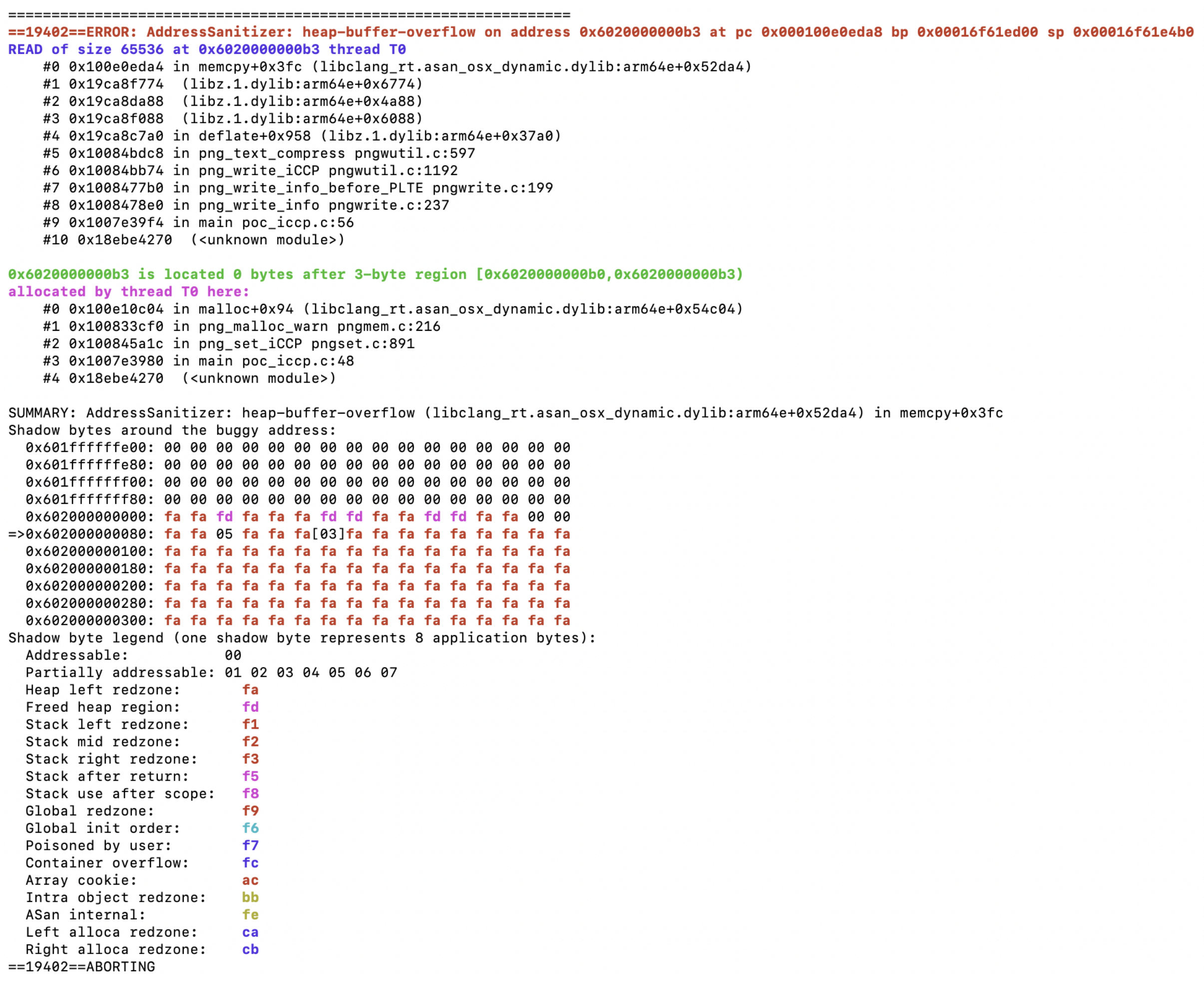
Although we could improve the coverage of `pngget.c` and `pngtrans.c` by simply calling the functions explicitly inside the harness, analyzing the relationship between functions, like analyzing function sub-calls will make the harness more resource efficient and therefore we will be able to get better result using the same fuzzing time.

Additionally, in order to further improve regarding the seed corpus, we suggest making more diverse and bigger seed corpus by using web scrapping. Our improved version of seed corpus is 83.9MB. Generating bigger, diverse seed corpus will allow the fuzzer to explore the full png read functions.

5 Part 4: Crash Analysis

From issue #648 we got that in a previous version of libpng, if the allocation size of the `'profile'` argument passed to `png_write_iCCP()` is less than 4 bytes, there is a heap overflow that occurs in `pngwutil.c` at line 1151:

```
   /* These are all internal problems: the profile should have been checked
    * before when it was stored.
    */
   if (profile == NULL)
      png_error(png_ptr, "No profile for iCCP chunk"); /* internal error */

   profile_len = png_get_uint_32(profile);

   if (profile_len < 132)
      png_error(png_ptr, "ICC profile too short");
```



```
=================================================================
==19402==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x6020000000b3 at pc 0x000100e0eda8 bp 0x00016f61ed00 sp 0x00016f61e4b0
READ of size 65536 at 0x6020000000b3 thread T0
    #0 0x100e0eda4 in memcpy+0x3fc (libclang_rt.asan_osx_dynamic.dylib:arm64e+0x52da4)
    #1 0x19ca8f774  (libz.1.dylib:arm64e+0x6774)
    #2 0x19ca8da88  (libz.1.dylib:arm64e+0x4a88)
    #3 0x19ca8f088  (libz.1.dylib:arm64e+0x6088)
    #4 0x19ca8c7a0 in deflate+0x958 (libz.1.dylib:arm64e+0x37a0)
    #5 0x10084bdc8 in png_text_compress pngwutil.c:597
    #6 0x10084bb74 in png_write_iCCP pngwutil.c:1192
    #7 0x1008477b0 in png_write_info_before_PLTE pngwrite.c:199
    #8 0x1008478e0 in png_write_info pngwrite.c:237
    #9 0x1007e39f4 in main poc_iccp.c:56
    #10 0x18ebe4270  (<unknown module>)

0x6020000000b3 is located 0 bytes after 3-byte region [0x6020000000b0,0x6020000000b3)
allocated by thread T0 here:
    #0 0x100e10c04 in malloc+0x94 (libclang_rt.asan_osx_dynamic.dylib:arm64e+0x54c04)
    #1 0x100833cf0 in png_malloc_warn pngmem.c:216
    #2 0x100845a1c in png_set_iCCP pngset.c:891
    #3 0x1007e3980 in main poc_iccp.c:48
    #4 0x18ebe4270  (<unknown module>)

SUMMARY: AddressSanitizer: heap-buffer-overflow (libclang_rt.asan_osx_dynamic.dylib:arm64e+0x52da4) in memcpy+0x3fc
Shadow bytes around the buggy address:
  0x601fffffe00: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x601fffffe80: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x601ffffff00: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x601ffffff80: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x602000000000: fa fa fd fa fa fa fd fd fa fa fd fd fa fa 00 00
=>0x602000000080: fa fa 05 fa fa fa[03]fa fa fa fa fa fa fa fa fa
  0x602000000100: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x602000000180: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x602000000200: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x602000000280: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x602000000300: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
Shadow byte legend (one shadow byte represents 8 application bytes):
  Addressable:           00
  Partially addressable: 01 02 03 04 05 06 07
  Heap left redzone:       fa
  Freed heap region:       fd
  Stack left redzone:      f1
  Stack mid redzone:       f2
  Stack right redzone:     f3
  Stack after return:      f5
  Stack use after scope:   f8
  Global redzone:          f9
  Global init order:       f6
  Poisoned by user:        f7
  Container overflow:      fc
  Array cookie:            ac
  Intra object redzone:    bb
  ASan internal:           fe
  Left alloca redzone:     ca
  Right alloca redzone:    cb
==19402==ABORTING
```

Figure 8: The iCCP main crash

5.1 Steps to reproduce

With the following POC, we managed to reproduce the bug. The idea to trigger the bug lies in calling:

```
png_set_iCCP(png_ptr, info_ptr,
            profile_name, compression_type,
            profile_data, profile_len);
```

with the malformed iCC profile. We then trigger the crash by writing this malformed when calling:

```
png_write_info(png_ptr, info_ptr);
```

In order to build this POC, one must first build the vulnerable version of libpng:

```
$ ./configure
$ make
$ sudo make install
```

Then build and run the exec:

```
$ gcc -fsanitize=address -g poc_iccp.c -o poc_iccp -lpng -lz
$ ./poc_iccp
```

5.2 Root Cause Analysis

In this vulnerable version of libpng, when writing the ICC (International Color Consortium) profile of a png with `png_write_iCCP`, the function incorrectly interprets the first 4 bytes of the profile buffer as a `uint32_t` size field (with `png_get_uint_32(profile)`), which leads to an OOB read if the profile data was less than 4 bytes long.
As mentioned earlier, the root of this vulnerability comes first from setting the malformed profile:

```
const char profile_name[] = "sRGB";
const png_charp profile_data = (png_charp)"ABC";
png_uint_32 profile_len = 3;
int compression_type = 0;

png_set_iCCP(png_ptr, info_ptr,
            profile_name, compression_type,
            profile_data, profile_len);
```

This will not trigger the crash by itself, but once the program reaches the point where it writes the png's informations (metadata of the png) with `png_write_info(png_ptr, info_ptr)`, it will call in the following order:

```
png_write_info(png_ptr, info_ptr); // call from main prog
png_write_info_before_PLTE(png_ptr, info_ptr);
png_write_iCCP(png_ptr, info_ptr->iccp_name, info_ptr->iccp_profile);
```

where the crash will occur at line 1151:

```
profile_len = png_get_uint_32(profile);
```

This results in a read-beyond-end-of-malloc bug if the profile data is less than 4 bytes long.

5.3 Security Implication

As this is used for write operations and the ICC profile is an important feature of pngs, it is not impossible to imagine a server that would rely on libpng to process images. Such heap-buffer-overflow could be triggered (depending on the features of the application) by a remote attacker to very easily perform a Denial-of-Service (DoS) by just crashing the application (with a simple ICC profile simply smaller than 4 bytes). What could lead to ever bigger concern is if the attacker could control the heap layout and be able to chain primitives. This being a much more complicated attack, would require more work and would not certainly be possible on every system using libpng. It is however truly recomanded to update to a fixed version of libpng.

5.4 Proposed fix

A fix has now been implemnted in the library:

```
png_write_iCCP(png_structrp png_ptr, png_const_charp name,
- png_const_bytep profile)
+ png_const_bytep profile, png_uint_32 profile_len)
{
...
- png_uint_32 profile_len;
...
- profile_len = png_get_uint_32(profile);
...
+ if (png_get_uint_32(profile) != profile_len)
+ png_error(png_ptr, "Incorrect data in iCCP");
...
}
```

We now pass the length of the profile as an argument to the png_write_iCCP function. This enables checking that it is long enough to avoid an OOB read:

```
  if (profile_len < 132)
    png_error(png_ptr, "ICC profile too short");
```

We can then safely call png_get_uint_32(profile) to make sure it matches the provided profile_len:

```
  if (png_get_uint_32(profile) != profile_len)
    png_error(png_ptr, "Incorrect data in iCCP");
```

Another measure to prevent this crash, that can be taken by the programmer (and that is probably good to use in general) is to check the validity of the ICC profile before setting it (and writing it):

```
if (png_get_valid(png_ptr, info_ptr, PNG_INFO_iCCP)) {
    // The iCCP chunk is valid
    png_set_iCCP(write_ptr, write_info, name, compression, profile, len);
} else {
    // The iCCP chunk is invalid
}
```

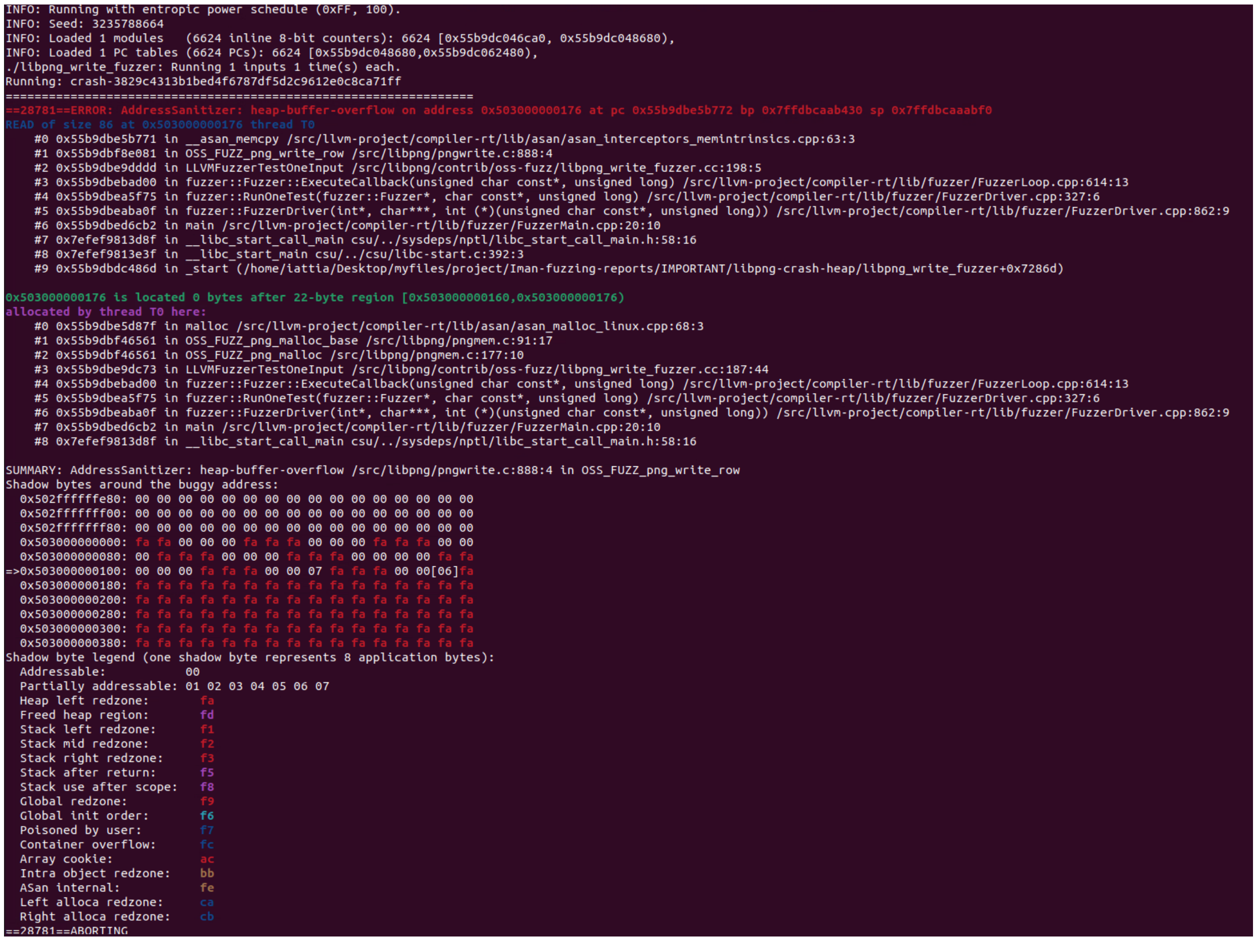
5.5 Important Note on Finding a crash

We actually managed to find a crash using a different version of libpng_write_fuzzer.cc. We successfully found a heap buffer overflow vulnerability in the function png_write_row, defined in pngwrite.c, at line 888:

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

We can see here that if the row size doesn't match the row_info.rowbytes size previously set, this results in an OOB write. Although like for the bug we previously covered, this could have a critical security impact, we were not sure if this was an actual bug in libpng or simply a programmer error. Either way, it would probably be a good addition to the library to mention in the documentation that the programmer should make sure that the size of the row buffer matches the one set in row_info.rowbytes before calling png_write_row(png_ptr, row_ptr).
To reproduce this crash, you can run the following script:

```
#!/bin/bash
set -e

git clone --branch crash_write --single-branch https://github.com/ImanEPFL/oss-fuzz.git && cd oss-
    fuzz

python3 infra/helper.py build_image --pull libpng

# Build fuzzers
python3 infra/helper.py build_fuzzers libpng

mkdir -p build/out/corpus/

# Run the fuzzer
python3 infra/helper.py run_fuzzer libpng libpng_write_fuzzer --corpus-dir build/out/corpus
```



```
INFO: Running with entropic power schedule (0xFF, 100).
INFO: Seed: 3235788664
INFO: Loaded 1 modules   (6624 inline 8-bit counters): 6624 [0x55b9dc046ca0, 0x55b9dc048680),
INFO: Loaded 1 PC tables (6624 PCs): 6624 [0x55b9dc048680,0x55b9dc062480),
./libpng_write_fuzzer: Running 1 inputs 1 time(s) each.
Running: crash-3829c4313b1bed4f6787df5d2c9612e0c8ca71ff
=================================================================
==28781==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x503000000176 at pc 0x55b9dbe5b772 bp 0x7ffdbcaab430 sp 0x7ffdbcaaabf0
READ of size 86 at 0x503000000176 thread T0
    #0 0x55b9dbe5b771 in __asan_memcpy /src/llvm-project/compiler-rt/lib/asan/asan_interceptors_memintrinsics.cpp:63:3
    #1 0x55b9dbf8e081 in OSS_FUZZ_png_write_row /src/libpng/pngwrite.c:888:4
    #2 0x55b9dbe9dddd in LLVMFuzzerTestOneInput /src/libpng/contrib/oss-fuzz/libpng_write_fuzzer.cc:198:5
    #3 0x55b9dbebad00 in fuzzer::Fuzzer::ExecuteCallback(unsigned char const*, unsigned long) /src/llvm-project/compiler-rt/lib/fuzzer/FuzzerLoop.cpp:614:13
    #4 0x55b9dbea5f75 in fuzzer::RunOneTest(fuzzer::Fuzzer*, char const*, unsigned long) /src/llvm-project/compiler-rt/lib/fuzzer/FuzzerDriver.cpp:327:6
    #5 0x55b9dbeaba0f in fuzzer::FuzzerDriver(int*, char***, int (*)(unsigned char const*, unsigned long)) /src/llvm-project/compiler-rt/lib/fuzzer/FuzzerDriver.cpp:862:9
    #6 0x55b9dbed6cb2 in main /src/llvm-project/compiler-rt/lib/fuzzer/FuzzerMain.cpp:20:10
    #7 0x7efef9813d8f in __libc_start_call_main csu/../sysdeps/nptl/libc_start_call_main.h:58:16
    #8 0x7efef9813e3f in __libc_start_main csu/../csu/libc-start.c:392:3
    #9 0x55b9dbdc486d in _start (/home/iattia/Desktop/myfiles/project/Iman-fuzzing-reports/IMPORTANT/libpng-crash-heap/libpng_write_fuzzer+0x7286d)

0x503000000176 is located 0 bytes after 22-byte region [0x503000000160,0x503000000176)
allocated by thread T0 here:
    #0 0x55b9dbe5d87f in malloc /src/llvm-project/compiler-rt/lib/asan/asan_malloc_linux.cpp:68:3
    #1 0x55b9dbf46561 in OSS_FUZZ_png_malloc_base /src/libpng/pngmem.c:91:17
    #2 0x55b9dbf46561 in OSS_FUZZ_png_malloc /src/libpng/pngmem.c:177:10
    #3 0x55b9dbe9dc73 in LLVMFuzzerTestOneInput /src/libpng/contrib/oss-fuzz/libpng_write_fuzzer.cc:187:44
    #4 0x55b9dbebad00 in fuzzer::Fuzzer::ExecuteCallback(unsigned char const*, unsigned long) /src/llvm-project/compiler-rt/lib/fuzzer/FuzzerLoop.cpp:614:13
    #5 0x55b9dbea5f75 in fuzzer::RunOneTest(fuzzer::Fuzzer*, char const*, unsigned long) /src/llvm-project/compiler-rt/lib/fuzzer/FuzzerDriver.cpp:327:6
    #6 0x55b9dbeaba0f in fuzzer::FuzzerDriver(int*, char***, int (*)(unsigned char const*, unsigned long)) /src/llvm-project/compiler-rt/lib/fuzzer/FuzzerDriver.cpp:862:9
    #7 0x55b9dbed6cb2 in main /src/llvm-project/compiler-rt/lib/fuzzer/FuzzerMain.cpp:20:10
    #8 0x7efef9813d8f in __libc_start_call_main csu/../sysdeps/nptl/libc_start_call_main.h:58:16

SUMMARY: AddressSanitizer: heap-buffer-overflow /src/libpng/pngwrite.c:888:4 in OSS_FUZZ_png_write_row
Shadow bytes around the buggy address:
  0x502ffffffe80: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x502fffffff00: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x502fffffff80: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x503000000000: fa fa 00 00 00 fa fa fa 00 00 00 fa fa fa 00 00
  0x503000000080: 00 fa fa fa 00 00 00 fa fa fa 00 00 00 00 fa fa
=>0x503000000100: 00 00 00 fa fa fa 00 00 07 fa fa fa 00 00[06]fa
  0x503000000180: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x503000000200: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x503000000280: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x503000000300: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x503000000380: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
Shadow byte legend (one shadow byte represents 8 application bytes):
  Addressable:           00
  Partially addressable: 01 02 03 04 05 06 07
  Heap left redzone:       fa
  Freed heap region:       fd
  Stack left redzone:      f1
  Stack mid redzone:       f2
  Stack right redzone:     f3
  Stack after return:      f5
  Stack use after scope:   f8
  Global redzone:          f9
  Global init order:       f6
  Poisoned by user:        f7
  Container overflow:      fc
  Array cookie:            ac
  Intra object redzone:    bb
  ASan internal:           fe
  Left alloca redzone:     ca
  Right alloca redzone:    cb
==28781==ABORTING
```

Figure 9: Our extra found crash: heap buffer overflow