CS-412: fuzzing libpng with oss-fuzz

HAMZA REMMAL, École Polytechnique Fédérale de Lausanne, Switzerland
CHANGLING WANG, École Polytechnique Fédérale de Lausanne, Switzerland
PEIRAN MA, École Polytechnique Fédérale de Lausanne, Switzerland
IBRAHIMA KALIL SAMA BOUZIGUES, École Polytechnique Fédérale de Lausanne, Switzerland

1 Introduction

In this project, we improved the existing fuzzing harness of the `libpng` library, a widely used library for working with portable network graphics (PNG) files. We guarantee the reproducibility of our results by fixing dependencies and modifying the build process. We identified two significant uncovered regions, analysed why they are not covered by the current fuzzing harness, and managed to get them under fuzzing coverage by improving the existing fuzzing harness and boosting the initial seed corpus with specifically crafted random seeds. Since we couldn't trigger any crash during our own fuzzing process, we give an analysis of the vulnerability https://issues.oss-fuzz.com/issues/372819450, which is due to an integer overflow, as well as feasible suggestions for fixing.

2 Part 1: Running Existing Fuzzing Harnesses

As stated in 1, we will be analysing and fuzzing the `libpng` project. As reproducibility is key in any project and research, we first started by forking all the repositories we depend on. Instead of having multiple repositories to manage, we decided to push every project into its own orphan branch in a single repository. As such, the following repository holds all the code required to reproduce the results. To identify all the required dependencies, we will first analyse the configuration of the project on `oss-fuzz`. The revision used during the analysis, and broadly the version of `oss-fuzz` used for this project, has been pushed to the `oss-fuzz` branch on the project's repository. The analysis of the project's `Dockerfile` highlighted a dependency to two other unpinned repositories on GitHub. The first step would be to gather all those repositories and to pin the revisions used during this project. As such, the repository contains the following branches:

- `oss-fuzz`: branch containing a fork of `oss-fuzz` as of commit `0a15922e5667404c3591e213ac964e96f87a5721`;
- `libpng`: branch containing a fork of `libpng` as of commit `ea127968204cc5d10f3fc9250c306b9e8cbd9b80`;
- `zlib`: branch containing a fork of `zlib` as of commit `5a82f71ed1dfc0bec044d9702463dbdf84ea3b71`.

Now that we have control over the codebases used for this project, we will start changing the build process to use the pinned versions for reproducibility. The first changes occurred in the `oss-fuzz` codebase, where the `Dockerfile` used to build the `libpng` environment. Commit f74446118f0341300878139f772214afeb16c5cd pins the version of `gcr.io/oss-fuzz-base/base-builder` —the base image used to build the `libpng` environment.

```
diff --git projects/libpng/Dockerfile projects/libpng/Dockerfile
index 6f281cd55..4e6322bc7 100644
--- projects/libpng/Dockerfile
+++ projects/libpng/Dockerfile
```

Authors' Contact Information: Hamza Remmal, hamza.remmal@epfl.ch, École Polytechnique Fédérale de Lausanne, Lausanne, Vaud, Switzerland; Changling Wang, changling.wang@epfl.ch, École Polytechnique Fédérale de Lausanne, Lausanne, Vaud, Switzerland; Peiran Ma, peiran.ma@epfl.ch, École Polytechnique Fédérale de Lausanne, Lausanne, Vaud, Switzerland; Ibrahima Kalil Sama Bouzigues, ibrahima.samabouzigues@epfl.ch, École Polytechnique Fédérale de Lausanne, Lausanne, Vaud, Switzerland.

```
@@ -14,7 +14,7 @@
 #
 ################################################################################

-FROM gcr.io/oss-fuzz-base/base-builder
+FROM gcr.io/oss-fuzz-base/base-builder@sha256:60965728fe2f95a1aff980e33f7c16ba378b57b4b4c9e487b44938a4772d0d2d
 RUN apt-get update && \
     apt-get install -y make autoconf automake libtool zlib1g-dev
```

Listing 1. Content of commit `f74446118f03413008781 39f772214afeb16c5cd`

As of now, we will using nested branches [1]. The following branches in the repository will contain modifications to `oss-fuzz` and `libpng` depending on the configuration (*with corpus* or *without corpus*).

- `with-corpus/libpng`: changes to the `libpng` project to be able to fuzz with the default corpus
- `with-corpus/oss-fuzz`: changes to the `oss-fuzz` project to be able to fuzz with the default corpus
- `without-corpus/libpng`: changes to the `libpng` project to be able to fuzz without the default corpus
- `without-corpus/oss-fuzz`: changes to the `oss-fuzz` project to be able to fuzz without the default corpus

> Please note that no changes occurred on the `zlib` project, and both versions (*with corpus* and *without corpus* uses the same codebase)

2.1 Running with the default corpus

Before running the script, we updated our own branch of `oss-fuzz` to use our own projects instead of the upstream repositories.

```
diff --git a/projects/libpng/Dockerfile b/projects/libpng/Dockerfile
index 6f281cd55..58bfb1ac4 100644
--- a/projects/libpng/Dockerfile
+++ b/projects/libpng/Dockerfile
@@ -14,11 +14,11 @@
 #
 ################################################################################

-FROM gcr.io/oss-fuzz-base/base-builder
+FROM gcr.io/oss-fuzz-base/base-builder@sha256:60965728fe2f95a1aff980e33f7c16ba378b57b4b4c9e487b44938a4772d0d2d
 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 --depth 1 https://github.com/hamzaremmal/fuzz-libpng.git -b zlib zlib
+RUN git clone --depth 1 https://github.com/hamzaremmal/fuzz-libpng.git -b with-corpus/libpng libpng
 RUN cp libpng/contrib/oss-fuzz/build.sh $SRC
 WORKDIR libpng
```

Listing 2. Running `git diff oss-fuzz..with-corpus/oss-fuzz`

No changes had to be made to `libpng`, hence both the `libpng` and `with-corpus` have the similar history.

To start running the experiment, we will first have to clone our fork of `oss-fuzz` and use it as the root for the rest of the experiment.

[1] Not an official Git term, but refers to branches following the pattern `feature-a/subfeature`

```
git clone --depth 1 https://github.com/hamzaremmal/fuzz-libpng.git -b with-corpus/oss-fuzz oss-fuzz
cd oss-fuzz
```

We then build the images using `oss-fuzz`. Remember, we build the image as defined in our own fork:

```
python3 infra/helper.py build_image libpng
```

We then proceed to build the fuzzers:

```
python3 infra/helper.py build_fuzzers --clean libpng
```

and run the only harness available for `libpng`:

```
python3 infra/helper.py run_fuzzer --corpus-dir "$CORPUS" libpng libpng_read_fuzzer -e FUZZER_ARGS=-max_total_time=
    $DURATION
```

We finish by building the coverage:

```
python3 infra/helper.py build_fuzzers --sanitizer coverage libpng
python3 infra/helper.py coverage --corpus-dir "$CORPUS" --fuzz-target libpng_read_fuzzer --no-corpus-download libpng
```

The full script to reproduce the results is available in `part1/run.w_corpus.sh` file in the project's repository. Please use the script as provided instead of running the described steps above manually. The script will set up a temporary folder for the corpus, and cleans it at the end of the script. That said, we do not save the report outside the temporary folder. To fetch it, please go inspect the temporary folder and copy the report to another folder manually **before** exiting and killing the process executing the script. By the end of the script, the coverage report will be served from a local http server. Unfortunately, we didn't find a way to not start the server (see 2.3 for more details).

The coverage report of our experiment is available in the `part1/report/w_corpus` folder.

2.2 Running without the default corpus

The steps to run the projects without the default corpus are the same steps as described in 2.1. Only the version of `oss-fuzz` changes in the script. The *diff* to apply to `oss-fuzz` is provided in the `part1/oss-fuzz.diff` file in the project's repository. The *diff* to apply to `libpng` is provided in the `part1/project.diff` file in the project's repository.

To remove the default corpus, we analysed the build script located in the `libpng` project and commented the lines responsible of adding the seed corpus. Thanks to the authors of the project, the code was easily identified because of comments.

```
diff --git a/contrib/oss-fuzz/build.sh b/contrib/oss-fuzz/build.sh
index 7b8f02639..be8498b72 100755
--- a/contrib/oss-fuzz/build.sh
+++ b/contrib/oss-fuzz/build.sh
@@ -43,8 +43,8 @@ $CXX $CXXFLAGS -std=c++11 -I. \
      -lFuzzingEngine .libs/libpng16.a -lz

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

 cp $SRC/libpng/contrib/oss-fuzz/*.dict \
      $SRC/libpng/contrib/oss-fuzz/*.options $OUT/
```

Listing 3. Running `git diff libpng..without-corpus/libpng`

The full script to reproduce the results is available in `part1/run.w_o_corpus.sh` file in the project's repository. The script suffers from the same limitations mentionned in 2.1.

The coverage report is available in the `part1/report/w_o_corpus` folder.

2.3 A few encountered issues

While writing this script, we encountered a few issues with the `oss-fuzz` and `libpng`. We will list a few of them here. First, a recent commit in the `libpng` project breaks the build in `oss-fuzz`. See https://github.com/pnggroup/libpng/issues/678 issue on the `libpng` issue tracker. We went around this issue by pinning the version of `libpng` to the commit before the breakage was introduced. We also encountered an issue with the `coverage` command in the where the `-no-serve` causes the `infra/helper.py` script to crash. Please note that while we pin the revisions of the codebases used in this project, and the base image of the environment, we don't pin the version of the docker image used by `python3 infra/helper.py run_fuzzer`. This was intentional as we don't want to surcharge the diff with changes that are not fully necessary in the scope of this project.

2.4 Differences between both runs

Filename(.c)	w_corpus	w_o_corpus
png	43.23%	28.92%
pngerror	54.91%	49.34%
pngget	3.50%	3.50%
pngmem	79.28%	79.28%
pngread	28.36%	17.24%
pngrio	78.57%	78.57%
pngrtran	30.50%	24.55%
pngrutil	72.96%	68.71%
pngset	53.03%	43.72%
pngtrans	8.67%	4.10%

Table 1. Line coverage of runs with corpus and without corpus

Comparing the results of runs with and without initial corpus, we can see that the coverage of almost all files are significantly higher when the initial seed corpus is given. This is consistent with the expectation, as png files come with a fixed 8-byte header, and having this header at the beginning of the file is the necessary condition to be a valid png file. Without the initial corpus, the fuzzer will only pass this check when it happens to generate the exact same bytes at the beginning of the file, which, even with the help of the given dictionary, is very unlikely to happen. Hence, without the initial corpus, it will take much more time for the fuzzer to generate valid inputs to trigger any meaningful behavior of the library, which greatly damages the fuzzing efficiency and is reflected on difference in the coverage.

3 Part 2

3.1 First Region

We detect the function **png_read_png** as one of the most extensively uncovered areas of the libpng codebase. From OSS-Fuzz Introspector we can see the coverage of this function and some of its callees is 0% in the current fuzzing setup.

The png_read_png function plays a crucial role in the libpng library as a high-level public API widely adopted by real-world software. It offers an all-in-one interface for reading and decoding complete PNG images in a single call, simplifying interaction with libpng's lower-level functionalities. As an official part of the PNGAPI, png_read_png handles the entire decoding pipeline: reading image metadata, setting up transformations, processing scanlines, and finalizing chunk operations.

Its widespread adoption is evident: a GitHub search restricted to the C language and querying png_read_png yields over 3,000 results, excluding private repositories and non-indexed codebases. This highlights the function's importance across various domains, particularly in graphics processing, embedded systems, and retro game engines.

To further illustrate its real-world significance, we investigated notable open-source repositories utilizing this function:

- freakwan implements a compressed image format and relies on png_read_png to decode standard PNG files as part of its firmware processing pipeline.
- cave-story-md, a fan recreation of the classic Cave Story game for the Sega Genesis, uses this function for asset loading and display.
- space-nerds-in-space, a networked multiplayer spaceship bridge simulator, leverages png_read_png for texture handling in its graphical interface.

These examples, all from projects with hundreds of stars, underline the relevance of png_read_png in mature, community-vetted codebases. Improving fuzzing coverage of this function is therefore critical for enhancing the robustness of software depending on libpng.

There are a lot of transformation setup calls wrapped internally by the **png_read_png**, such as **png_set_strip_alpha**, **png_set_expand_16**, **png_set_invert_mono** and **png_set_bgr**. These functions are only selected depending on the transformation flags passed to the transforms parameter.

Importantly, many of these transformation functions have 0% coverage with **png_read_png** being the only call site they have. That is, that the logic inherent in them is entirely untested, unless we call upon this function. It also includes calls to **png_read_update_info** – used to complete transformation logic, **png_read_image** - which deals with row-wise image decoding, and **png_read_end** – in charge of parsing the rest of the chunks after image data.

Such functions, their nested callees, including **png_read_row** and **png_combine_row**, have contained the fundamental logic for memory management, bit-depth expansion, interlacing, and filtering, and last rows transformation. Many of these operations are low level byte manipulation and buffer logic - prime candidates for memory corruption vulnerabilities if malformed inputs are passed.

Taking a closer look at the existing fuzzing harness, we find that it actually does a similar job (among others) to that of **png_read_png**: it specifies several hard-coded transformations which are referred to in the comment as **typically used by browsers**, finishes the setup of the transformations with a call to **png_read_update_info**, it then reads the image row by row with **png_read_row** and completes the parsing with **png_read_end**.

This explains why **png_read_png** is not covered by the existing fuzzing harness, as the harness itself is trying to functionally mimic what **png_read_png** does. However, this design choice creates a blind spot in the fuzzing pipeline: any code that is only triggered through the high-level read interface but omitted by the harness will not be tested at all.

Furthermore, most of the transformation logic in **png_read_png** is flag-dependent, meaning the paths taken within the function depend heavily on the bits set in the transforms field.

We verified this by examining the libpng-proto project, which does fuzz **png_read_png** using its own custom harness and transformation seeds. However, libpng-proto is explicitly out of scope of the official OSS-Fuzz integration for libpng. This strongly supports our hypothesis: the coverage gap is not due to irrelevance of the function, but due to architectural exclusion in the main fuzzing configuration.

3.2 Second Region

In the following analysis, we reviewed **png_image_read_colormap** function, which is part of the libpng library. Using the OSS-Fuzz introspector, we can see that the coverage of this function is 0%. One of its main responsibilities is to process images dependent on colormaps by producing a lookup table that maps palette indices to RGBA colour. This procedure is responsible for the right grayscale, palette, or low-bit-depth RGB(A) encoded image interpretation.

Intended to make the process easier, this function offers an approach that is simple to use in comparison with the standard **png_struct** and **png_info** technique. Based on these specifications, the function creates its internal colormap and changes to meet the specified parameters.

It achieves this by calling helper functions, such as **make_gray_colormap**, **make_ga_colormap**, **make_rgb_colormap**, **make_gray_file_colormap**, **png_create_colormap_entry**. All of these functions have no tests to cover them yet. Besides, there are functions like **colormap_compose** and **decode_gamma**, that contribute to configuring the colormap. Such processes are characterized by critical transformations like gamma decoding, alpha blending, and color quantization which are demanding for performance and crucial in safeguarding security.

Currently, the existing harness only invokes this API with the most simple configuration: feeding in the fuzzing data, with the typical format specification (**PNG_FORMAT_RGBA**), without specifying any colormap or background color. As a result, all the functionality of **png_image_read_colormap** will never be invoked and is left untested in the current fuzzing environment.

The absence of coverage of this function, and its callees, implies that many behaviors do not get inspected in testing. For instance, **png_create_colormap_entry** finds and stores the RGBA values in the colormap by means of transformations. **colormap_compose** does alpha-based combination of foreground and background pixels; **decode_gamma** carries out color channel adjustments according to gamma specifications. These procedures are very susceptible to underflow, overflow, and rounding errors. Moreover, the complexity of control logic in **make_rgb_colormap** and **make_ga_colormap**, engines of transformations, render these functions susceptible to silent failure or abuse, especially if tRNS chunks are incorrect, background values inconsistent, or gamma values non-conventional.

png_image_read_colormap is a major part of libpng's simplified API, is logic-intensive and transformation-oriented. Its total absence of coverage in the existing fuzzer demonstrates a potential weakness. Although the function is called upon to perform important and subtle image processing endeavors such as blending, compositing, and color decoding, it is not triggered in the current harness. Since it is the only pathway to necessary transformation and blending capabilities, we consider it a critical uncovered area. When applied to an advanced harness or customized fuzzer, attention to this function could reveal previously unknown aspects of its functioning, thus enormously increasing and reinforcing the test scope of libpng.

4 Part 3

Filename(.c)	Introspector	Original	Improve1	Improve2
png	48.32%	43.27%	43.97%	55.44%
pngerror	55.26%	55.26%	55.26%	55.26%
pngget	3.50%	3.50%	2.96%	3.50%
pngmem	79.28%	79.28%	79.28%	79.28%
pngread	28.36%	28.36%	32.62%	60.88%
pngrio	78.57%	78.57%	78.57%	78.57%
pngrtran	30.17%	30.17%	37.24%	49.04%
pngrutil	77.34%	73.35%	84.31%	73.34%
pngset	53.03%	53.03%	53.03%	53.03%
pngtrans	8.67%	8.67%	83.61%	26.51%

Table 2. Line coverage from introspector report and reports of our own runs

4.1 png_generator.py : A Random Seed Generator

Currently, the libpng fuzzer simply takes all the valid png files inside the repository as the seed corpus. However, based on the results from the official introspector report, it seems that the current seed corpus does not comprehensively include all the valid ancillary chunks, and the coverage of their corresponding processing routines heavily relies on the input mutations generated by the fuzzer. However, since the mutation dictionary only contains the 4-byte chunk type tokens, the chunks are created in a syntax-agnostic way during the fuzzing process. To complement this, we decided to develop a seed generator (**`png_generator1.py`**) that randomly generates seeds comprising different combinations of chunks, in the hope of resulting in more interesting execution paths.

During the generation, the generator randomly picks a subset of chunk types to include in the output; and for each chunk, it also randomly decides whether the data in the chunk should be valid or corrupted.

In our configuration, this generator is invoked in the fuzzer building script (**`build.sh`**) to generate 10 images to be added to the seed corpus. Since the fuzzer is rebuilt before every 4-hour run, the images in the seed corpus across different runs will be different. This generator is used in **both improve1** and **improve2**.

4.2 Improve1

Based on our analysis in 3.1, we decided to modify the fuzzing harness to directly invoke **`png_read_png`**, instead of its current approach of manually calling lower-level routines. Since **`png_read_png`** is basically a superset of the existing fuzzing harness in terms of the processing functionalities invoked, such a modification ensures that all the regions that are already covered are still covered by the improved fuzzing harness. Furthermore, since **`png_read_png`** allows one to specify which transformations to perform by setting the corresponding bit flags in its argument **`transform`**, we can map part of the fuzzing input to this argument to randomly invoke different combinations of transformations, including transformations not invoked by the existing harness, and thereby achieve higher coverage.

Comparing the coverage results 2 of our improved harness with that from the introspector report [2], we can see that our improvement yields a huge increase in the coverage of **`coverage.c`**, which directly results from the invocation of different transformations specified by the argument **`transform`**. In addition, we also see significant increase in the coverage of **`pngrtran.c`** and **`pngrutil.c`**, which results from subroutines invoked by the various transformations. Although the coverage of **`png.c`** is lower, it is probably due to the difference in running time, as it is comparable to that of our own run of the original harness.

Since we have already made the invocation of this interface as general as we can, there is no obvious option to make further improvements along this direction.

4.3 Improves2

Based on our analysis in 3.2, we decided to modify the harness to invoke the API with a given background color and colormap. Since the colormap-related processing routines are only invoked when the corresponding flag (**`PNG_FORMAT_FLAG_COLORMAP`**) is set in the argument **`format`**, we need to manually set this flag to enforce the invocation of these target routines. Similar to improve1, we also map part of the fuzzing input to the argument **`format`** to randomly modify other configurations such as the number of pixel channels.

Comparing the coverage results 2 of our improved harness with that from the introspector report, we can see a huge increase in the coverage of **`pngread.c`**, as well as significant increase in that of **`pngrtran.c`**, **`pngtrans.c`** and **`png.c`**. Similarly to **`improve1`**, the negative difference in the coverage of **`pngrutil.c`** compared to the introspector report is likely due to the difference in running time, as the coverage is comparable with that of our own run of the original harness.

As for further improvements, since currently we are using hard-coded colormap and background color, it is possible to adjust the harness to map part of the fuzzing input to these variables. Such a modification would enable fuzzing the API with random and potentially invalid color configurations.

4.4 How-tos

Both improvements are achieved by modifying the existing fuzzing harness and things should just work after applying the changes (**`git apply libpng.diff`**) to the code of the original libpng repository. To make the scripts runnable standalone, the repository states after the changes are applied are stored as 2 branches (**`improve1/libpng`** and **`improve2/libpng`**) in our project repository.

To facilitate the development process, the running scripts will generate a new local fuzzing project called **`libpng_cs412`** and run everything in the new project. However, since the report generation process of oss-fuzz automatically sets up a server to serve the report, and we could not find a way to terminate the server gracefully within the script without root privilege, the script needs to be manually terminated with ctrl+c after the server is set up. After that, the generated report can be found at the default location where oss-fuzz stores reports (**`build/out/libpng_cs412/report`**). The build script (**`build.sh`**) is modified to invoke the random seed generator (**`png_generator1.py`**) before packing up the seed corpus.

[2] https://storage.googleapis.com/oss-fuzz-coverage/libpng/reports/20250501/linux/file_view_index.html

5 Part 4

We did not find a bug with our newly updated fuzzer so we will triage https://issues.oss-fuzz.com/issues/372819450. To triage we build the vulnerable version of libpng (commit `20f819c29e49f4b8c1d38e3f475b82a9cdce0da6`) and write a small script to reproduce using the OSS-fuzz helper. First, we clone the oss-fuzz repo using our own version. We had to modify the `Dockerfile` of the `libpng` project to use the vulnerable version of `zlib` and `libpng`. These versions can be found in our repo in the `part4/zlib` and `part4/libpng` branches. Next, we build the libpng image and fuzzers using the sanitizer flag set to `undefined` as our bug is an undefined behaviour. Finally, we reproduce the bug using the `reproduce` command from the oss-fuzz helper file.

5.1 Root Cause Analysis

The bug is present in versions of libpng prior to a patch committed in October 2024. It is reported as: **`runtime error: signed integer overflow: 31523510 + 2145133844 cannot be represented in type 'png_int_32'`**. This occurs in the function **`png_xy_from_XYZ`** at line 1240 of **`png.c`**. The overflow was detected via Undefined Behavior Sanitizer (UBSan) while executing this expression: **`dwhite += d`**.

The **`png_xy_from_XYZ`** function converts color information from the CIE XYZ color space into chromaticity coordinates (x, y) used by libpng internally. This conversion is essential for processing cHRM chunks in PNG files, which store chromaticity information for red, green, blue, and white points. The function operates on 32-bit signed fixed-point values (**`png_int_32`**) and performs calculations like:

- X + Y + Z per color
- Accumulated white point values: whiteX, whiteY, dwhite
- Normalized chromaticity output using png_muldiv(...)

The crash happens because these additions are done without overflow checks, so malformed or extreme input values in the cHRM chunk can exceed the representable range of int32_t.

In the vulnerable version, libpng calculates the reference white point by summing the red, green, and blue components individually — without checking for overflow. Here's the problematic logic:

```
d = XYZ->red_X + XYZ->red_Y + XYZ->red_Z; // first triplet sum
//...
dwhite = d;
d = XYZ->green_X + XYZ->green_Y + XYZ->green_Z; // second triplet
//...
dwhite += d;
d = XYZ->blue_X + XYZ->blue_Y + XYZ->blue_Z; // third triplet
//...
dwhite += d;
```

All these values are **`png_int_32`** (a 32-bit signed integer). The variable dwhite accumulates the sum of 9 integers in total — 3 per color component (R, G, B) — without any overflow protection. This causes an undefined integer overflow, which is not caught at compile time or at runtime without sanitizers like UBSan.

Our proposed fix uses 64-bit integers (int64_t) to perform the intermediate arithmetic, followed by a bounds check before assigning back to 32-bit. using a 64 bit for big_d the code would be:

```
big_d = XYZ->red_X + XYZ->red_Y + XYZ->red_Z; // first triplet sum
if (big_d > INT32_MAX  big_d < INT32_MIN) return 1; // check for overflow
```

```
d = (png_int_32)big_d;
//...
dwhite = d;

big_d = XYZ->green_X + XYZ->green_Y + XYZ->green_Z; // second triplet
if (big_d > INT32_MAX  big_d < INT32_MIN) return 1; // check for overflow
d = (png_int_32)big_d;
//...
dwhite += d;

big_d = XYZ->blue_X + XYZ->blue_Y + XYZ->blue_Z; // third triplet
if (big_d > INT32_MAX  big_d < INT32_MIN) return 1; // check for overflow
d = (png_int_32)big_d;
//...
dwhite += d;
```

We could even create a helper function to follow DRY principle. Indeed, the fix implemented in the libpng repository replaces direct additions with guarded ones using a helper function **`png_safe_add`**:

```
if (png_safe_add(&d, dred, dgreen)) return 1;
```

`png_safe_add` uses an internal **`png_fp_add`** routine to detect overflow explicitly and returns an error if the operation would overflow. Introducing this function is a better approach because it can be reused in other places and allows to detect overflows.

5.2 Security Implication

This seems to be a bug rather than an actively exploitable vulnerability.

The issue is reachable by passing a malformed PNG image containing a corrupted `cHRM` chunk. The vulnerable function, **`png_xy_from_XYZ`**, is called via **`png_read_info`**, a function commonly used in applications that rely on libpng to parse image headers. This means the crash can be triggered in many real-world scenarios without a lot of work from the adversary:

- When an application loads or previews a user-supplied PNG file
- During image conversion or processing (image uploads for example)
- In services that analyze or validate PNG metadata

The immediate impact is:

- **Denial of Service (DoS)**: if UBSan is enabled, the crash halts the application.
- **Silent corruption**: if UBSan is disabled, incorrect chromaticity values may propagate undetected.

While there is no direct control over memory layout or instruction flow, the presence of undefined behavior in a core parsing path is serious. In some contexts, such behaviour may be used in combination with other vulnerabilities to mount more advanced attacks.

In conclusion, this bug has not a big impact by itself but is relatively easy to trigger.