CS412 Fuzzing Lab Report

Anthony Philippe CHRISTOFOROU
Mohammad Massi RASHIDI
Christian WILLIAM

April 2025

Abstract

1 Introduction

Students:

- Anthony, 328790
- Massi, 394309
- Christian, 313229
- ~~4th Person~~

We had a fourth person that was supposed to be in our group but suddenly dropped out without notice which made the scheduling a bit more difficult.

For the fuzzing lab we chose the project **libpng** `https://github.com/pnggroup/libpng`, the official PNG reference library. A simple and not so much fuzzed yet very important library.

We forked the original oss-fuzz project, as well as the project repo and a main repo that will be our main source of truth and the final submission.

- `https://github.com/hectellian/oss-fuzz`
- `https://github.com/hectellian/libpng`
- `https://github.com/hectellian/cs-412-oss-fuzz`

2 Running Existing Fuzzing Harnesses

For the `libpng` project there was only one fuzzer to run, `libpng_read_fuzzer`. So we ran this fuzzer, seeded, for 4h and another 4 unseeded using 2 scripts.

2.1 Commands Used

All commands below assume you are in the top-level of the submission directory and that `run.w_corpus.sh` and `run.wo_corpus.sh` are executable.

2.1.1 With the Default Seed Corpus

```
# Ensure the script is executable
chmod +x run.w_corpus.sh

# Run the seeded campaign
./run.w_corpus.sh
```

Listing 1: Running the run.w_corpus.sh script.

This will perform the following steps:

1. Clones `https://github.com/google/oss-fuzz.git` into `oss-fuzz/`

2. Builds the `libpng` fuzzers (`build_image` + `build_fuzzers`).

```
            cd "$OSS_FUZZ_DIR"
python3 infra/helper.py build_image "$PROJECT"
python3 infra/helper.py build_fuzzers "$PROJECT"
```

Listing 2: Building images.

3. Runs `libpng_read_fuzzer` inside the official OSS-Fuzz Docker image (which packages the seed-corpus zip) for `$TIMEOUT` seconds which is set to 4h.

```
docker run --rm --privileged \
    -v "$CORPUS_DIR":/corpus \
    -v "$BUILD_DIR/$PROJECT":/out \
    gcr.io/oss-fuzz/"$PROJECT" \
    /out/"$PROJECT_FUZZER" \
    -artifact_prefix=/corpus/ \
    -max_total_time="$TIMEOUT" \
    /corpus
```

Listing 3: Running the fuzzer.

4. Rebuilds with coverage instrumentation (`-sanitizer coverage`) and invokes `infra/helper.py coverage` to generate the HTML report under

```
python3 infra/helper.py build_fuzzers --sanitizer coverage "$PROJECT"
python3 infra/helper.py coverage "$PROJECT" \
  --corpus-dir "$CORPUS_DIR" \
  --fuzz-target "$PROJECT_FUZZER" \
  --no-serve

DESTDIR="$WORKDIR/part1/report/w_corpus"
mkdir -p "$DESTDIR"
rm -rf "${DESTDIR:?}"/*

cp -r "$COVERAGE_DIR"/* "$DESTDIR"/
```

Listing 4: Generate Coverage.

`part1/report/w_corpus/`

2.1.2 Without Any Seed Corpus

```
# Ensure the script is executable
chmod +x run.w_o_corpus.sh

# Run the seeded campaign
./run.w_o_corpus.sh
```

Listing 5: Running the run.w_o_corpus.sh script.

This script does almost the same thing than the previous one but with a few different steps:

1. Clones our forks of both `libpng` and `oss-fuzz` on branch no-seed-corpus (which already contain our patches).

Configuration	Covered Lines	Total Lines	Coverage(%)
With seed corpus	3 287	12 837	25.61%
No seed corpus	4 699	12 837	36.61%

Table 1: Line coverage comparison (4h fuzzing)

2. Exports the two diff files into `part1/report/`
    - `project.diff` (in `libpng/contrib/oss-fuzz/build.sh`, the lines that zip up `.png` files are commented out)
    - `oss-fuzz.diff` (in `projects/libpng/Dockerfile`, the Dockerfile is modified to pull from our `libpng` fork `no-seed-corpus branch`)
3. The rest stays the same except it outputs to `part1/report/w_o_corpus`

2.2 Diff File Locations & Changes

- `submission/part1/report/project.diff`

```
# 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
```

Listing 6: project.diff file.

Removes the packaging of all `.png` files into the seed corpus.

- `submission/part1/report/oss-fuzz.diff`

```
-RUN git clone --depth 1 https://github.com/pnggroup/libpng.git
+RUN git clone --depth 1 https://github.com/hectellian/libpng.git --branch no-seed-corpus
```

Listing 7: oss-fuzz.diff file.

Points the Docker build at our patched `libpng` fork (so the removed-seed build.sh is used).

2.3 Coverage Results & Comparison

After 4 hours of fuzzing each setup, the overall line coverage across all libpng sources is:

We can observe an unexpected higher coverage without the seeding, it outperformed the seeded run by over 11%

So what could be the reason for this? We think that because the shipped seed corpus (all `.png` under the libpng tree) consists of valid PNG files. Fuzzing with these seeds spends much of its time driving deep parsing of well-formed images—valuable, but narrow in scope.

In contrast, starting from an empty corpus forces the fuzzer mutations to generate numerous small, malformed inputs. These rapidly exercise error-handling paths and early-exit logic (e.g. sanity checks, unexpected-chunk handlers, minimal header parsing), covering large swaths of code that valid images never touch.

A few exemples where coverage can differ:

```
if (size < kPngHeaderSize) {
    return 0;
}
```

Listing 8: Exemple of where line coverage can differ for libpng_read_fuzzer.

```
// only valid PNGs will ever
// skip into the real parser; random mutations almost always
// trigger this check and bail out early.
std::vector<unsigned char> v(data, data + size);
if (png_sig_cmp(v.data(), 0, kPngHeaderSize)) {
return 0;
}
```

Listing 9: Exemple of where line coverage can differ for libpng_read_fuzzer.

```
// Everything below here (png_read_info, transforms, read_row loops, etc.)
// will only be covered by inputs that pass the signature check.
png_read_info(png_ptr, info_ptr);
...
for (int pass = 0; pass < passes; ++pass)
    for (png_uint_32 y = 0; y < height; ++y)
      png_read_row(png_ptr, row_ptr, nullptr);
```

Listing 10: Exemple of where line coverage can differ for libpng_read_fuzzer.

3 Part 2

Uncovered Region 1

Location: `png.c`
Function: `png_free_data`

Functionality. The `png_free_data` function is responsible for releasing all memory linked to optional metadata chunks present in a PNG file. This includes fields such as textual comments (`tEXt`), calibration data (`pCAL`, `sCAL`), Exif information (`eXIf`), palette histograms (`hIST`), and more. The function is internally invoked during cleanup phases to ensure memory safety and proper deallocation.

Why it is not covered. In the original fuzzing setup, these code paths were not executed because the default seed corpus did not contain PNG files that triggered allocation of these optional metadata structures. For instance, if no `eXIf` or `pCAL` chunk is present, the corresponding memory never gets allocated, and the cleanup logic for those structures is skipped entirely.

Why it matters. This part of the code is crucial for robust memory handling. Incorrect freeing of fields like dynamically allocated arrays or nested strings can cause memory leaks, double-frees, or use-after-free vulnerabilities. Historical bugs in similar image libraries have stemmed from such logic errors during cleanup. These paths are difficult to fuzz unless the structures are first allocated via valid input.

How to reach it. To activate this logic, one must supply valid PNG inputs that contain these less-common chunks. In our case, we crafted PNGs with proper `eXIf`, `sCAL`, and `pCAL` chunks using a custom generator script. Once these chunks are parsed and corresponding fields allocated, the cleanup logic in `png_free_data` becomes reachable and is covered during program termination or memory release phases.

Uncovered Region 2: `png_set_filler` in `pngtrans.c`

Location: `src/libpng/pngtrans.c`
Function: `png_set_filler`

Functionality. The `png_set_filler` function configures libpng to insert a filler byte (usually for alpha) when reading or writing image data. It updates the internal structure to apply a post-processing transformation that affects how pixels are interpreted and stored. This transformation is used in scenarios such as converting RGB to RGBA or adding alignment bytes.

Why it is not covered. The default `libpng_read_fuzzer` does not apply any transformations: it decodes PNG files without invoking optional setup functions such as `png_set_filler`. Therefore, the corresponding flags (e.g., `PNG_ADD_FILLER`, `PNG_ADD_ALPHA`) in `png_ptr->transformations` are never set, and the code handling filler insertion is skipped. Since this function is not conditionally triggered by the PNG file content but must be explicitly configured via API, the probability that random input alone activates this code is zero, unless the harness is modified to call it explicitly.

Why it matters. This function affects how image buffers are allocated and manipulated. If used incorrectly (e.g., mismatch between expected and actual number of channels), it may lead to out-of-bounds writes, crashes, or subtle corruption of image output. These transformations are particularly tricky when interacting with bit-depth-specific paths or when chained with other transformations (e.g., `png_set_swap` or `png_set_packing`).

How to reach it. A simple improvement would be to modify the existing fuzzer harness to call `png_set_filler(png_ptr, 0xFF, PNG_FILLER_AFTER);` just after creating the read structures. This will activate the code paths associated with filler insertion and cause the relevant checks and assignments to be executed during fuzzing.

4 Part 3

In this part, we aim to improve the code coverage of `libpng` by targeting regions that remained uncovered in our initial fuzzing campaigns. These campaigns consisted of 3 runs of 4 hours each using the default harness and seed corpus. The resulting baseline coverage was:

- Line coverage: 24.46%
- Function coverage: 34.00%
- Region coverage: 20.47%

Based on the analysis conducted in Part 2, we identified two critical code paths that were not exercised:

- `png_free_data()` in `png.c`, responsible for deallocating memory associated with optional metadata chunks
- `png_set_filler()` in `pngtrans.c`, which adjusts pixel formats by inserting padding bytes

These paths are not reachable through random mutation alone. Therefore, we implemented two improvements:

1. **Corpus enrichment:** We added a set of valid PNGs containing metadata chunks such as `tEXt`, `iTXt`, and `zTXt`, ensuring that metadata-parsing logic would be exercised.

2. **Harness enhancement:** We modified the fuzzer harness to explicitly call `png_set_filler()` after image header parsing, enabling transformation logic that is otherwise inactive by default.

Each improvement was tested through a 4-hour fuzzing campaign (x3 trials), and the coverage was compared against the baseline described above.

4.1 Improvement 1: Adding Metadata-Rich PNGs to the Corpus

To improve coverage in regions such as `png_free_data()` in `png.c` and more generally, to target libpng code paths that are only triggered when optional PNG chunks are present in the input, we generated a diverse set of valid PNG images containing well-formed metadata chunks using a custom Python script named `generate_image.py`. This includes `tEXt`, `iTXt`, `pHYs`, `oFFs`, `pCAL`, `sCAL`, `eXIf`, and other rarely encountered chunks.

Implementation. The script is stored under `cs-412-oss-fuzz/part3/improve1/corpus_improve/generate_image.py`. It constructs minimal PNG files with embedded metadata (e.g., `tEXt`, `iTXt`, `eXIf`, `oFFs`, etc.). Each file provides coverage for one or more libpng metadata processing routines.

The output images (e.g., `text_chunk.png`, `offs_chunk.png`) are committed to the `libpng` repository in the `improve1` branch under `contrib/oss-fuzz/seed/`. For reference, a copy is also included in the submission directory.

Repository changes. We committed the images and script to a dedicated branch `improve1`. The associated diff is stored at:

- `submission/part3/improve1/projet.diff`

Build and run. The fuzzing campaign was executed using the following script:

```
chmod +x run.improve1.sh
./run.improve1.sh
```

Listing 11: Run script for Improvement 1

Coverage results. After 4 hours of fuzzing (3 trials), the coverage report confirms that the previously unreachable function `png_get_text()` which was unreachable due to lack of metadata chunks in the corpus. is now executed. This validates the effectiveness of our corpus augmentation as the metadata structures are now allocated and freed during execution. Overall line coverage increased from 24.46% to 25.96%, with local improvements in files such as `pngget.c` and `pngread.c`.

Artifacts.

- Coverage reports: `part3/improve1/coverage_improve1/`
- Run script: `part3/improve1/run.improve1.sh`

Analysis. This improvement highlights a key limitation of the original seed corpus: it lacked diversity in terms of PNG features, notably metadata chunks like `tEXt`, `iTXt`, and `zTXt`. By injecting images that explicitly exercise these features, we uncovered code paths entirely missed by the default configuration. Although the global coverage gain is modest, the improvement demonstrates the importance of a well-designed, feature-targeted corpus to complement mutation-based fuzzing.

Improvement 2 – Harness Modification to Trigger `png_set_filler`

Targeted Region. This improvement targets the previously uncovered function `png_set_filler` in `pngtrans.c`, which configures libpng to insert a filler (or alpha) byte during pixel decoding.

Strategy. We modified the fuzzer harness `libpng_read_fuzzer.cc` to explicitly call `png_set_filler(png_ptr, 0xFF, PNG_FILLER_AFTER)` immediately after reading the image header via `png_read_info(...)`. This activates filler-related transformation logic and enables execution of code that would otherwise remain dormant.

Implementation. The modified harness was committed to a new branch named `improve2` in our `libpng` fork. No changes were made to the `oss-fuzz` repository beyond updating the Dockerfile to track this branch. This ensures the updated harness is properly built and included in the fuzzing workflow.

Relevant diffs:

- `part3/improve2/coverage_improve2/project.diff`
- `part3/improve2/coverage_improve2/oss-fuzz.diff`

Build and run. Fuzzing was executed using the following script:

```
chmod +x run.improve2.sh
./run.improve2.sh
```

Listing 12: Run script for Improvement 2

Coverage Results. After 4 hours of fuzzing (3 trials), our results show a clear increase in global coverage. Overall line coverage increased from 24.46% to 34.62%, function coverage rose to 43.25%, and region coverage reached 26.64%. The targeted file `pngtrans.c` is now partially covered, and the function `png_set_filler` is successfully reached.

Analysis. This result confirms that key transformation paths are unreachable without explicit harness control. Unlike improvements via corpus extension, this improvement shows the critical impact of harness configuration: internal library logic triggered via API calls (rather than input structure) requires manual activation. As with `png_set_filler`, many transformation routines (`png_set_swap_alpha`, `png_set_strip_16`, etc.) likely remain uncovered and could benefit from similar treatment.

Conclusion. The filler configuration logic touches sensitive memory layout areas, which are known hotspots for bugs. By activating this logic through harness changes, we expose libpng to a new class of structural decoding behaviors. Future work should automate or randomize transformation combinations to further increase code coverage and improve test depth in other hard-to-reach transformation paths.

5 Part 4

As our improved fuzzer did not uncover any new bugs, we opted to investigate a previously known vulnerability, **CVE-2019-7317**, reported in `libpng` version `1.6.36`. This issue was originally classified as a **heap-use-after-free** bug occurring in the simplified API under certain memory cleanup conditions.

While attempting to reproduce the historical vulnerability **CVE-2019-7317**, which is documented as a **heap-use-after-free** in the `libpng` simplified API, our proof-of-concept triggered a different bug. Specifically, we observed a **stack-use-after-return** in `libpng` version `1.6.36`, due to a stale stack pointer left in the `png_image` structure's internal state. AddressSanitizer detected and confirmed the issue as a clear case of invalid stack memory access after return. This occurs when cleanup logic in

`png_image_free()` attempts to access a stack-allocated structure that is no longer valid, leading to undefined behavior.

Vulnerable fork: `https://github.com/hectellian/libpng`
Trigger script: `run.poc.sh` (automates build, PoC generation, and crash execution)

Root Cause Analysis The root cause lies in the interaction between the following functions:

- `png_image_begin_read_from_file()` internally creates a `png_control` structure on the stack and stores a pointer to it inside `image->opaque`.
- Later, when `png_image_free(&image)` is called, the function attempts to access `image->opaque`, invoking cleanup routines via `png_safe_execute()`. These routines dereference the pointer to `png_control` that was stack-allocated in the now-returned function, leading to undefined behavior.

Key Code Snippets (Simplified)

To better illustrate the root cause, we provide simplified versions of the relevant functions from `libpng`. These demonstrate how stack memory is misused across API boundaries.

```
/* png_image_begin_read_from_file() (simplified) */
int png_image_begin_read_from_file(png_image* image, const char* file) {
    png_control ctrl; // Stack-allocated control structure
    image->opaque = &ctrl; // DANGER: Stores pointer to stack memory
    // ... image loading logic ...
} // <-- Stack frame ends here, ctrl is no longer valid

/* png_image_free() (simplified) */
void png_image_free(png_image* image) {
    if (image->opaque) {
        png_safe_execute(image, image->opaque->cleanup);
        // CRASH: Dereferences a pointer to invalid stack memory
    }
}
```

Listing 1: Vulnerable memory usage in libpng

In this code, the `png_control` structure is allocated on the stack inside `png_image_begin_read_from_file()`, and a pointer to it is assigned to `image->opaque`. However, since `ctrl` is a stack variable, it becomes invalid after the function returns. When `png_image_free()` is later called, it accesses the now-dangling pointer to `ctrl`, resulting in a **stack-use-after-return** memory error.

This design flaw violates basic memory safety principles by allowing a stack pointer to escape its lifetime and be reused later, leading to undefined behavior when accessed. **Note:** This is **not** the originally documented CVE crash, which described a double `free()` on a heap pointer. Instead, this reproduction uncovered a distinct yet closely related issue stemming from the same mismanagement of object lifetime.

PoC C Snippet (inlined in `run.poc.sh`)

The following C code is embedded directly within the `run.poc.sh` script of part 4 repository. It is written to disk, compiled, and executed automatically as part of the crash reproduction pipeline.

```
int main() {
    printf("libpng version: %s\n", png_get_libpng_ver(NULL));
    png_image image;
    memset(&image, 0, sizeof(image));
    image.version = PNG_IMAGE_VERSION;
    ///// Load image metadata from a PNG file
    if (png_image_begin_read_from_file(&image, "crash_poc.png")) {
        png_bytep buffer = malloc(PNG_IMAGE_SIZE(image));
```

```
        if (buffer != NULL) {
            ////// Fully decode the image
            png_image_finish_read(&image, NULL, buffer, 0, NULL);
            free(buffer); // Free the decoded data
        }
    }
    ////// This second free triggers the crash
    png_image_free(&image);
    return 0;
}
```

Listing 2: PoC C code demonstrating the stack-use-after-return in libpng 1.6.36

The script `run.poc.sh` automates all steps necessary to reproduce the crash:

- Clones the vulnerable `libpng` fork and Checks out the vulnerable tag (`v1.6.36`),
- Builds the library with AddressSanitizer enabled,
- Creates a tiny valid PNG image (`crash_poc.png`) in base64,
- Writes the PoC C code (see Listing 2) to disk and Compiles the PoC with `libpng` statically linked, and Executes the binary to reliably trigger the crash.

Instructions to Run the PoC

To reproduce the crash described above, navigate to the `part4` directory of our submission. Then, execute the following commands:

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

The `run.poc.sh` script will automatically clone the vulnerable version of `libpng`, build it with AddressSanitizer support, generate a minimal PNG image, compile a proof-of-concept (PoC) program, and run it to trigger the crash.

Note: The file `poc.c` is included in the submission for clarity, as it contains the exact C code that is dynamically written and compiled within `run.poc.sh`. However, it is not used directly and does not need to be compiled manually.

Proposed Fix and Justification

To eliminate the **stack-use-after-return** vulnerability, we must ensure that the internal control structure (`png_control`) remains valid until it is explicitly freed by `png_image_free()`. The root problem is that this structure was allocated on the stack in `png_image_begin_read_from_file()`, then accessed via a dangling pointer.

A robust fix consists of allocating the `png_control` structure on the heap instead of the stack, and assigning the heap pointer to `image->opaque`, ensuring its lifetime persists beyond the function call.

Modified code (illustrative):

```
/* Fixed version of png_image_begin_read_from_file() */
int png_image_begin_read_from_file(png_image* image, const char* file) {
    png_control *ctrl = malloc(sizeof(png_control)); // Heap allocation
    if (!ctrl) return 0; // Handle allocation failure
    memset(ctrl, 0, sizeof(png_control));
    image->opaque = ctrl; // Safe: now points to heap memory
    // ...
    return 1;
}

/* png_image_free() remains unchanged */
void png_image_free(png_image* image) {
    if (image->opaque) {
        png_safe_execute(image, image->opaque->cleanup);
        free(image->opaque); // Safe to free now
```

```
        image->opaque = NULL;
    }
}
```

Listing 3: Proposed fix: heap allocation of png$_control$

Justification: This fix preserves the design of the simplified API while ensuring the internal state is allocated safely. Since the lifetime of the heap-allocated structure is now managed explicitly, we avoid accessing invalid memory when freeing the image. Furthermore, this approach does not require modifying the public `png_image` structure, ensuring backward compatibility.

Upstream Fix (libpng 1.6.37). The libpng maintainers adopted a similar solution in version `1.6.37`. In the upstream patch, the `png_control` structure was moved from stack to heap allocation within the simplified API functions. This prevents pointers to invalid stack memory from persisting and ensures that all access to `image->opaque` remains valid until the image is freed.

- This change was minimal, contained, and safe.
- It preserved the simplified API's usability while resolving a critical lifetime management issue.
- It avoids intrusive changes such as embedding the control data in the public struct.

Conclusion. Both our proposed fix and the upstream solution adopt the same core strategy: **use dynamic memory allocation** to ensure internal state lives long enough to be safely cleaned up. This fix fully resolves the bug without affecting users of the simplified API or introducing new invariants.

Severity and Exploitability Analysis

Severity. This vulnerability is a **stack-use-after-return**, triggered when `png_image_free()` accesses a stack-allocated `png_control` structure that was created in `png_image_begin_read_from_file()` and became invalid after that function returned.

Such a bug can be remotely triggered by supplying a crafted PNG to an application using the simplified libpng API. Affected applications (e.g., image processors or web servers) may crash upon parsing such input, leading to a **Denial-of-Service (DoS)**. In hardened builds with sanitizers, this results in a clean crash. In production, the impact is undefined behavior.

Exploitability. Although stack corruption is generally harder to exploit than heap corruption, several attack primitives are theoretically possible:

- **Arbitrary Stack Write:** Cleanup logic may corrupt nearby stack variables.
- **Leakage:** Crashes might expose stack pointers, weakening ASLR.
- **Return Address Overwrite:** Under specific conditions (no stack canaries, predictable layout), control flow hijacking is conceivable.
- **Reliable DoS:** Easily achievable by any malformed PNG using this API sequence.

Real-World Impact. The simplified API is less widely used than the full libpng interface, which limits the exposure somewhat. Still, any application using this interface remains at risk, especially when untrusted image files are accepted.

Resolution. The bug was acknowledged and fixed in `libpng` version `1.6.37`, where the `png_control` structure was moved to the heap. This confirms the security flaw was real and significant enough to require upstream remediation.

Final Assessment.

- **Exploitability:** Low–Medium
- **Impact:** Medium–High
- **CVSS (estimated):** 6.5 – High