Software Security - Fuzzing Report

Group 3

s1148245 s1148243 s1149182 s1148338

February 5, 2025

1 Introduction

Fuzzing is a widely-used technique involving feeding a program with random data to uncover potential vulnerabilities or stability issues (thus effective in identifying memory corruption, crashes and/or security flaws).

These tools represent diverse approaches to fuzzing, ranging from evolutionary to basic mutation-based techniques.

In this project, we conducted our fuzzing experiments on the **libspng** library, using three fuzzing tools: **AFL++** [2], **zzuf** [7]and **radamsa** [6]. We used different tools to compare the outcomes among them and discuss the potential differing results.

We wanted to:

1. Evaluate the effectiveness of different fuzzer/sanitizer combinations.
2. Assess their performance and reliability.
3. Investigate any discovered vulnerabilities and implications.

The following sections detail the experimental setup, results and findings of our experiments.

2 The Fuzzing Experiments

2.1 Experiment Setup

o test the robustness of the libspng library, we used **AFL++**, **zzuf** and **radamsa**. AFL++ was chosen for its maximum code coverage, Zzuf and Radamsa -mutation-based fuzzers, for their simplicity and efficiency in introducing randomness to input data.

To enhance the the fuzzing process, we employed three sanitizers. AddressSanitizer (ASan) [3] was used to detect spatial and temporal memory safety issues. MemorySanitizer (MSan) [4] to identify uninitialized memory usage. Valgrind's Memcheck [5] tool was also used as a secondary diagnostic tool.

The experiments began with a corpus of 175 valid PNG image files obtained from PNGSuite [8], a comprehensive collection of PNG test cases. This diversity, and also the use of their corrupted files, was intended to achieve more code coverage (specially for corner-case bugs).

To streamline the setup, we wrote a custom installation script (`install.sh`) to automate the setup and installation of the required fuzzers and sanitizers, as well as the cloning and building of the libspng library. This should further provide a reproducible environment. By running the script without arguments, users can view a list of available fuzzers and sanitizers for selection.

2.2 Main application

You can find the main repository here: `https://github.com/RaySteak/SS-libspng-fuzz`.

As you can see, we created a main script `generic_test.c` which tried to integrate all the tools, sanitizers and input corpus into a cohesive workflow, so that it would facilitated consistent and repeatable fuzzing campaigns by automating some tasks.

Initially, the script used a random seed obtained from a source like `urandom`. However, to ensure deterministic results, we afterwards decided that we will derive the seed from the central bit of the image buffer (`buf[siz_buf/2]`). In this way, we ensured that specific images consistently produce identical results, paramount if we want reproducibility in testing. Furthermore, some fuzzers benefit from the increased stability/reproducibility in the outputs of the program gained by using a deterministic seed (e.g AFL++[1]).

It is worth mentioning that we created a class, `PNGConfig`, containing all the fields required to configure every parameter based on the image name. In the `read` phase, all functions responsible for retrieving parameters (`spng_get` methods) were thoroughly tested. These functions correspond to the decoding process. Conversely, the `write` phase handled encoding, which required explicit initialization of all parameters — otherwise we got memory issues. Parameters were either correctly initialized (90% of the cases) or left partially random (10% of the times).

2.3 Experiment Results

In this section, we present the results of our experiments. To make it more readable, we will use some tables in order to guide the reader through the results.

Table 1: Summary of Fuzzing Experiments

ID_Experiment	Tool	Time (hrs)	Number of Test Cases	Issues Found
1	AFL++	14.3	220.667.240	0 errors
2	zzuf	41.6	29.000.000	0 errors
3	radamsa	22	9.647.000	0* errors

Without using any sanitizers, all the fuzzing tools did not discover any particular issue or error. This might suggest that, although the fuzzers did generate a variety of test cases, they were not able to identify any vulnerabilities without the added checks provided by sanitizers.

However, an *error worth mentioning was the `ERROR FOUND IN OPTIMIZED COMPILATION`, identified while running radamsa without sanitizers. Due to the `spng_set_exif` function, when we also used compiler optimizations such as `-O1`, `-O2`, or `-O3`, we got segmentation faults in cases where no EXIF data was present and occasionally even when there was. The line that was causing problems in this case was `test(spng_set_exif(ctx, &exif));`. This could indicate that optimizations might introduce issues that are otherwise not evident during regular executions.

Table 2: Summary of Fuzzing Experiments with ASan

ID_Experiment	Tool	Time (hrs)	Number of Test Cases	Issues Found
1	AFL++ with ASan	14.3	22.052.208	0 errors
2	zzuf with ASan*	26.2	2.800.000	1797** errors
3	radamsa with ASan	14.8	1.929.000	39.499** errors

* zzuf had to be used to fuzz through an auxiliary program because of incompatibility with ASan.

Most of the **errors were heap buffer overflows. Specifically for zzuf, 126 came from `memcmp` called by `check_exif` in `spng_set_exif`. In the other 1671, most of the errors (1644 of them) came from the `write_chunks_before_idat` function (13 in `memcpy` called by `deflate` in `compress2` from zlib, and 1631 in `memcpy` called by `write_chunk` in `write_unknown_chunks`), while the other 27 came from a `memcpy` call traced to `write_chunks_after_idat`. In general, for radamsa, the errors occurred in functions related

to compression (deflate), chunk writing (write_chunks_before_idat and write_chunks_after_idat), and row encoding (encode_scanline, encode_row). Additionally, overflows were detected in spng_set_exif during EXIF data processing. The issues suggest improper memory handling across various paths, though they were not tied to specific input files.

Table 3: Summary of Fuzzing Experiments with MSan

ID_Experiment	Tool	Time (hrs)	Number of Test Cases	Issues Found
1	AFL++ with MSan	14.3	15.090.830	135 distinct crashes
2	zzuf with MSan*	18.7	2.800.000	3456** errors
3	radamsa with MSan	34.2	7.284.592	43.616** errors

* zzuf had to be used to fuzz through an auxiliary program because of incompatibility with MSan.

While running radamsa and zzuf, we detected an error of an uninitialized variable at line 2905:

```
text->text_length = strlen(text->text);
```

This issue occurs in the `read_non_idat_chunks` function, which is invoked by `read_chunk`. Both functions are utilized in all `get` and `set` methods, but the error is triggered exclusively in `get` methods and decoding operations for images containing text chunks.

**Specifically, every image in the format `ct*n0g04`, where `*` can be one of the following: `1, e, f, g, h, j, z` (or, in other words, all images containing text).

The error was the following:

```
    Uninitialized bytes in strlen at offset 8 inside [0x701000000006, 9)
==847104==WARNING: MemorySanitizer: use-of-uninitialized-value
    #0 0x7c6fd4a68e54 in read_non_idat_chunks spng.c
    #1 0x7c6fd4a69cf5 in read_chunks spng.c
    #2 0x7c6fd4a6b46b in spng_decode_image (libspng/build/libspng.so+0xb46b)
    (BuildId: 5187f185eac2c97a5bac5545051a8298a7bdd1a7)
    #3 0x572abd3fccf9 in fuzz_spng_read /user/software_security/SS-libspng-
fuzz/
    fuzz/generic_test.c:639:9
    #4 0x572abd3f80cb in main /user/software_security/SS-libspng-fuzz/fuzz/
    generic_test.c:217:19
    #5 0x7c6fd462a1c9 in __libc_start_call_main csu/../sysdeps/nptl/
    libc_start_call_main.h:58:16
```

```
    #6 0x7c6fd462a28a in __libc_start_main csu/../csu/libc-start.c:360:3
    #7 0x572abd35f634 in _start (/user/software_security/SS-libspng-fuzz/fuzz/
    generic_test_msan.fuzz+0x32634) (BuildId: cf7afa532aef04299dfbc5414d0d2616
    e9c18ae7)
SUMMARY: MemorySanitizer:
use-of-uninitialized-value spng.c in read_non_idat_chunks
Exiting
```

Subsequently, these images were skipped, although new problems arose also with images: tbbn2c16, tbgn2c16, tbrn2c08.

Table 4: Summary of Fuzzing Experiments with valgrind

ID_Experiment	Tool	Time (hrs)	Number of Test Cases	Issues Found
1	zzuf with valgrind	62.4	210.000	too many* errors

* The test had to be stopped and rerun to suppress some errors, as the output file was becoming too large. After the errors were suppressed, no other errors showed up. The errors that were suppressed were the following: Use of uninitialised value of size 8 in the `crc32` libz function called by `finish_chunk`, Conditional jump or move depends on uninitialised value(s) (strlen and strstr in `check_png_keyword`, `check_offs` in `spng_set_offs`, and `check_exif` in `spng_set_exif`).

2.4 Analysis of Results

There were several significant patterns for the different fuzzing approaches and configurations.

First of all, there was notable consistency in the results obtained withing individual fuzzing runs. This was due the fact that our testing framework was designed to be deterministic, ensuring that our input corpus for each fuzzer was producing the same outcomes; therefore, making it 100% reproducible. However, when comparing the results and performances of the different fuzzers, we noticed some inconsistencies and some (significant) variations.

For instance, AFL++ with the ASan sanitizer failed to discover any crash in the program, suggesting it might be less effective in triggering address related edge-case behavior. In contrast, when we used radamsa or fuzz under the same conditions, the fuzzers identified 39.499 and 1797 errors each.

It is also interesting to note the different impact of sanitizers. For example, ASan showed very limited utility in detecting any kind of issues, most likely because the detected problems were unrelated to spatial or temporal memory corruption (ASan's specializations). In contrast, MSan was much more useful, revealing significant numbers of issues related to the use of uninitialized memory (see the error about read_non_idat_chunks in the previous section).

2.4.1 Vulnerabilities discovered

Running the fuzzers with the MSan sanitizer lead to the discovery of different possible vulnerabilities. In the case of AFL++ we found five distinct problematic functions and lines in the library stemming mostly from its extensive use of the malloc allocation function without posterior initialization of the allocated data. First of all, we were able to rediscover the error found by radamsa in the function read_non_idat_chunks (see the error in the section 2.3), which on closer examination we discovered it was caused by running the strlen function mentioned above on a string buffer which has uninitialized values. We also would like to note that there is a known vulnerability of the same type (uninitialized access) for this function in the vulert vulnerability database [9]. Although the database doesn't list the complete details of the vulnerability, we hypothesize that it is the same as the one we found as it is relatively common (all of our tested fuzzers found it).

Furthermore, we found another uninitialized value in function spng_decode_scanlin at line 3401 of spng.c:

```
pixel[0] = plte[entry].red;
```

The plte array comes from the spng context (variable ctx) and a specially crafted input can trigger a specific execution path in the code in which it's allocated using malloc without posterior initialization, which leads to undefined behavior when reading from the array.

Another interesting finding is inside the function read_scanline at line 3236:

```
next_filter = ctx->scanline[scanline_width - 1];
if(next_filter > 4) ret = SPNG_EFILTER;
```

Again we found another execution path which leads to reading an uninitialized value. This is because of the initialization of the scanline array, which again comes from the spng context and is allocated using malloc. The error occurs during image decoding:

some specific edge case in images with zlib compressed data can lead to only partial initialization of the array causing the next_filter variable to contain an uninitialized value before performing the comparison on the second line.

Other similar vulnerabilities are found in function gamma_correct_row at line 1752 and in function expand_row at line 1963. As far as we are aware, all vulnerabilities except the one from read_non_idat_chunks are new discoveries.

3 Discussion and Conclusions

3.1 Ease of use and challenges

3.1.1 zzuf

`zzuf` failed first and foremost in being incompatible with ASan and MSan and needs an auxiliary program that fuzzes the input file and outputs it to stdout. The output of this program will then be piped into the stdin of the program that needs fuzzing. This can become unwieldy. Another obvious flaw of `zzuf` is that it's a very simple fuzzer, only fuzzing the program once with a random input. It is then the user's job to write lengthy bash scripts to fuzz using different seeds and inputs or on multiple cores. Making sense of the output is a hassle in itself as well, as errors will have to be classified manually or using parsing methods.

3.1.2 AFL++

AFL++ is a very effective tool for fuzzing complex targets. With multiple optimization configurations, code instrumentation, strong parallelization capabilities, its own compiler and a full-fledged status interface it is a very complete tool which streamlines the fuzzing process very rapidly. The main drawback from this complexity is that it has quite a steep learning curve and, at least in our case, it required multiple hours just to set up the tests correctly. One big challenge we phased at the beginning of the development cycle was compiling and installing the tool and also understanding all the different compilation options available for the instrumentation of the test program. However, once the initial setup phase was completed everything worked as intended and we could seamlessly start fuzzing using different instances which are automatically configured to synchronize between each other and share results. The status screen also provides very useful information in real-time, including coverage, the speed of the fuzzing, the number of crashes

and hang found, and an indication on how much more time you should keep fuzzing the program. It also gives you clues on possible errors or misconfiguration in the fuzzing process or test program.

The most challenging aspect of using AFL++ was analyzing the results. We had to develop our own custom python script (parse_afl_output.py) which takes the discovered crash input files and runs them again with the test program, parses the error output generated by the sanitizers and summarizes the different problematic lines of code found by the fuzzer. A final challenge to mention is the fact that AFL++ uses complex names for the input test files it generates, instead of maintaining the original name of the file. This resulted in us having to parse this complex format using an AFL++ specific function in the test program to obtain the original filename required by the program (as we mentioned before, the file name is used to obtain the format of the image). Furthermore, in some cases the original file name wasn't even inside of the new name created by AFL++, which means the function just returns a random string extracted from the file name.

3.1.3 radamsa

radamsa proved to be very easy to use, its primary role was to mutate our existing images, which were then used as inputs for the target program. That meant that it required only a specification of the seed, the number of mutations and, of course, the image to be mutated. This approach operates offline, which means that it do not directly interact with the executable, making it applicable also in scenarios where the source code is unavailable.

In contrast, the main problem was that it operates as a completely random fuzzer with limited features. It leaves the burden on the developer to determine what went wrong and where. Additionally, it lacks sophistication; because of its random nature, the mutated images might repeatedly trigger the same execution paths thousands of times, without actively exploring new paths as a guided fuzzer like AFL++ does.

3.1.4 ASan and MSan

As they are just compiling options for Clang, ASan and MSan should be reasonably easy to use. It turns out that it is not quite so, as ASan and MSan were the source of a lot of headache on the newer releases of the Linux kernel (6.8.0-49 specifically) by failing to be compatible with high-entropy Address Space Layout Randomization (ASLR). Extensive debugging to trace the cause of the problem pointed to one single culprit: The `vm.mmap_-`

`rnd_bits` Kernel configuration parameter. This parameter specifies the number of bits of entropy to be used for ASLR and has a default of 32. Decreasing the entropy bits to 30 seemed to fix the problem.

3.2 Effectiveness and Performance trade-offs

Most of the tools have proven to be quite effective in finding hard-to-detect flaws. `zzuf` is as effective as it is simple, and it managed to find the same errors as `radamsa`.

Since `zzuf` and `radamsa` are similar, they also share similar performances. When `zzuf` is run in its intended mode (fuzzing file descriptors), it should, in theory, be faster than `radamsa` as it does all the fuzzing in an online way. However, this mode is incompatible with ASan and MSan, and the fix to get it to work adds extra overhead, making it slower than `radamsa`.

Simplicity seems to be the downfall of both `zzuf` and `radamsa`, as they rely on scripting languages (or compiled programs, for that matter) that load and unload the program from memory with every run, while tools like AFL++ integrates in a single program the generation and testing of the input making them ultimately faster. Furthermore, despite being slower, they don't perform test case minimization as `AFL++`.

In combination with the sanitizers, we can say that ASan was very effective in detecting memory corruption (e.g. buffer overflows), while MSan excelled in identifying uninitialized variable usage. And both zzuf and radamsa benefited from the additional usage and diagnostics of these sanitizers. Valgrind was also a very useful tool in detecting vulnerabilities that otherwise would've been overlooked by both ASan and MSan, but it came at the cost of performance, needing a long time to run even for way fewer tests.

References

[1] *AFLplusplus/docs/fuzzing_in_depth.md at stable · AFLplusplus/AFLplusplus — github.com.* `https://github.com/AFLplusplus/AFLplusplus/blob/stable/docs/fuzzing_in_depth.md`. [Accessed 08-12-2024].

[2] AFL++ Contributors. *AFL++ - A superior fork of the original AFL Fuzzer.* `https://github.com/AFLplusplus/AFLplusplus`. Accessed: 2024-12-07.

[3] Google Developers. *AddressSanitizer.* `https://github.com/google/sanitizers/wiki/AddressSanitizer`. Accessed: 2024-12-07.

[4] Google Developers. *MemorySanitizer*. https://github.com/google/sanitizers/wiki/MemorySanitizer. Accessed: 2024-12-07.

[5] Valgrind Developers. *Valgrind*. https://valgrind.org/. Accessed: 2024-12-07.

[6] Aki Helin. *Radamsa*. https://gitlab.com/akihe/radamsa. Accessed: 2024-12-07.

[7] Sam Hocevar. *zzuf*. http://caca.zoy.org/wiki/zzuf. Accessed: 2024-12-07.

[8] Willem van Schaik. *PngSuite - The official set of PNG test images*. http://www.schaik.com/pngsuite/. Accessed: 2024-12-07.

[9] *Use-of-uninitialized-value in read_non_idat_chunks in libspng — vulert.com*. https://vulert.com/vuln-db/oss-fuzz-libspng-11308. [Accessed 08-12-2024].