Software-Security (NWI-ISOFSE)

Radboud Universiteit

Fuzzing libemf2svg

Quartile 2 - 2023 - 2024

Group 7

Full Name	Student ID
A.P. Valen	s1129905
Rick ten Tije	s1005826
Floris Rossel	s1070794

Nijmegen, December 6, 2023

1 Introduction

This project focuses on fuzzing the latest version of the libemf2svg [1] library, an Enhanced MetaFile (EMF) to Scalable Vector Graphics (SVG) converter. We aim to use various EMF images as input and report any potential bugs.

2 libemf2svg

Libemf2svg is a library for converting Microsoft Enhanced Metafile (EMF) to SVG. EMF/EMF+ files are not commonly found on their own but are frequently embedded within other Microsoft file formats. The motivation behind this project was to accurately convert Visio stencils (.VSS) to SVG, enabling the reuse of content in environments beyond Microsoft Office environment. This library also has a command-line interface, therefore enabling automated fuzzing. As an input on the library, EMF files should be provided.

3 Experiments

In the context of this project, we employed fuzzing tools such as AFL++, zzuf, and HonggFuzz to systematically explore the libemf2svg library's behavior under various inputs. These tools generate a large number of test cases, allowing us to identify potential crashes, hangs, or other issues that might arise during the execution of the libemf2svg.

3.1 Corpus Used for Fuzzing

For effective fuzzing, a corpus or initial input is essential to serve as input to the library. Various corpora were employed to detect possible crashes and hangs. The Original EMF corpus, sourced from the GitHub repository of libemf2svg, comprises 186 files ranging from 260 B to 500 KB. In addition to the original corpus, tests included a single text file containing "hello," a single complex EMF file of 141 KB, and, for the final test, crashes discovered during the initial AFL++ fuzzing on the original corpus were used as input.

3.2 Distinctive Fuzzer Approaches

Each fuzzer employed in this project utilizes distinct strategies and techniques to generate test cases and explore the program's execution space. Understanding these differences is pivotal for evaluating the effectiveness of each tool in identifying potential issues.

- **AFL++:** AFL++ (American Fuzzy Lop ++) functions as an evolutionary fuzzer, employing a blend of genetic algorithms and coverage-guided fuzzing. It prioritizes the mutation of existing test cases based on code coverage feedback.
- **zzuf:** zzuf operates as a mutation-based fuzzer, introducing random changes to input files. It proves particularly effective in identifying vulnerabilities arising from unexpected or malicious input.
- **HonggFuzz:** HonggFuzz acts as a feedback-driven fuzzer, utilizing code coverage information to guide the generation of new test cases.

In our experimentation, we utilize **Address Sanitizer** (ASan) and **Undefined Behavior Sanitizer** (UBSan) and **Valgrind** as essential tools for sanitizing the application. ASan helps detect memory-related flaws, such as buffer overflows and memory leaks, enhancing overall security. UBSan identifies undefined behaviors, like type mismatches and division by zero, contributing to a more comprehensive evaluation of code integrity and security. The crash analysis's found by both sanitizers will be reported.

4 Results

The results of our fuzzing experiments are summarized in Table 1. This table provides an overview of the experiments conducted with different fuzzers and initial corpora. The table clearly show that HonggFuzz with ASan & UBsan is better at finding issues than AFL++ with or without these sanitizers and zzuf.

Experiment	Tool	Input	Time	Number of test cases	Issues found
#1	AFL++	Original EMF corpus	65.7 hr	177.5 M	2 crashes, 4 hangs
#2	AFL++	1 text file	17 hrs	85.3 M	14 hangs
#3	AFL++	1 complex EMF	16 hrs	82.3 M	17 hangs
#4	AFL++	Generated EMFs	14 hrs	83.9 M	22 hangs
#6	AFL++ with ASan & UBsan	Crashes #1	77.8 hrs	91.3 M	263 (9 unique) crashes, 13 hangs
#7	zzuf	Original EMF corpus	2.5 hrs	540k	16 crashes
#8	zzuf	1 complex EMF	4.5 hrs	1 M	3 crashes
#9	zzuf	1 text file	4 hrs	1 M	0 crashes
#10	HonggFuzz with ASan & UBsan	Original EMF corpus	41 min	721k	220 (7 unique) crashes
#11	HonggFuzz with ASan & UBsan	1 complex EMF	10 min	221k	677 (1 unique) crashes
#12	HonggFuzz with ASan & UBsan	Generated EMF	26 min	1.56 M	5952 (1 unique) crashes
#13	HonggFuzz with ASan & UBsan	1 text file	55 min	7 M	0 crashes, 1242 hangs
#14	HonggFuzz with ASan & UBsan	crashes #1	35 min	551k	256 (11 unique) crashes

Table 1: Results of different fuzzers applied to libemf2svg with start corpora

4.1 Crash Analysis

4.1.1 AFL++

The main number of crashes detected using AFL++ are illegal instructions, these could be platform bounded. However AFL++ also found several crashes using Address sanitizer (ASan)

Crashes found with ASan that where already reported on GitHub:

```
SUMMARY: Address-sanitizer: heap-buffer-overflow ...
    /home/arianvaIen/final/libemf2svg/src/lib/emf2svg_utils.c: 1222:30 in fontindex_to_utf8
```

```
SUMMARY: AddressSanitizer: stack-buffer-overflow ...
    /home/arianvalen/final/libemf2svg/src/lib/emf2svg.c:50:20 in U_emf_onerec_analyse
```

```
    SUMMARY: AddressSanitizer: heap-buffer-overflow ...
        /home/arianvalen/final/libemf2svg/src/lib/emf2svg_rec_control.c:92:67 in ...
        U_EMRHEADER_draw
```

```
    SUMMARY: AddressSanitizer: heap-buffer-overflow ...
        /home/arianvalen/final/libemf2svg/src/lib/uemf_utf.c:164:16 in wchar16len
```

New crashes found with ASan which is not reported on GitHub:

```
SUMMARY: AddressSanitizer: heap-buffer-overflow ...
    (/home/arianvalen/final/libemf2svg/emf2svg-conv+0x43822b) in MemcpInterceptorCommon
```

```
SUMMARY: AddressSanitizer: heap-buffer-overflow ...
    /home/arianvalen/final/libemf2svg/src/lib/uemf.c:1213:25 in DIB_to_RGBA
```

Using undefined behaviour sanitizer (UBsan) the illegal instruction errors can be traced back to a known crash on GitHub:

```
SUMMARY: Undefined BehaviorSanitizer: SEGV ...
    /home/arianvalen/libemf2svg/src/lib/emf2svg_utils.c:1368:17 in reverse_utf8
```

4.1.2 zzuf

All the errors found with zzuf were segmentation faults. We used Valgrind here to get more information about the errors. Most of the errors were caused by invalid readings by the program as shown in the output below.

```
==145== Invalid read of size 4
==145==    at 0x48B6D35: fontindex_to_utf8 (in /usr/lib/libemf2svg.so.1.1.0)
==145==    by 0x48B7605: text_convert (in /usr/lib/libemf2svg.so.1.1.0)
==145==    by 0x48B78FA: text_draw (in /usr/lib/libemf2svg.so.1.1.0)
==145==    by 0x48BCE7A: U_EMREXTTEXTOUTA_draw (in /usr/lib/libemf2svg.so.1.1.0)
==145==    by 0x48CD327: U_emf_onerec_draw (in /usr/lib/libemf2svg.so.1.1.0)
==145==    by 0x48CD9A2: emf2svg (in /usr/lib/libemf2svg.so.1.1.0)
==145==    by 0x10BA28: main (in /usr/bin/emf2svg-conv)
==145==  Address 0x5410430 is 8,512 bytes inside an unallocated block of size 4,058,352 in ...
    arena "client"
==145==
```

Valgrind gave the following output for one of the complex EMF test case crashes.

```
==153== Invalid read of size 4
==153==    at 0x48B6D35: fontindex_to_utf8 (in /usr/lib/libemf2svg.so.1.1.0)
==153==    by 0x48B7605: text_convert (in /usr/lib/libemf2svg.so.1.1.0)
==153==    by 0x48B78FA: text_draw (in /usr/lib/libemf2svg.so.1.1.0)
==153==    by 0x48BCEF8: U_EMREXTTEXTOUTW_draw (in /usr/lib/libemf2svg.so.1.1.0)
==153==    by 0x48CD343: U_emf_onerec_draw (in /usr/lib/libemf2svg.so.1.1.0)
==153==    by 0x48CD9A2: emf2svg (in /usr/lib/libemf2svg.so.1.1.0)
==153==    by 0x10BA28: main (in /usr/bin/emf2svg-conv)
==153==  Address 0x53e1f7c is not stack'd, malloc'd or (recently) free'd
==153==
==153==
==153== Process terminating with default action of signal 11 (SIGSEGV)
==153==  Access not within mapped region at address 0x53E1F7C
==153==    at 0x48B6D35: fontindex_to_utf8 (in /usr/lib/libemf2svg.so.1.1.0)
==153==    by 0x48B7605: text_convert (in /usr/lib/libemf2svg.so.1.1.0)
==153==    by 0x48B78FA: text_draw (in /usr/lib/libemf2svg.so.1.1.0)
==153==    by 0x48BCEF8: U_EMREXTTEXTOUTW_draw (in /usr/lib/libemf2svg.so.1.1.0)
==153==    by 0x48CD343: U_emf_onerec_draw (in /usr/lib/libemf2svg.so.1.1.0)
==153==    by 0x48CD9A2: emf2svg (in /usr/lib/libemf2svg.so.1.1.0)
==153==    by 0x10BA28: main (in /usr/bin/emf2svg-conv)
==153==  If you believe this happened as a result of a stack
==153==  overflow in your program's main thread (unlikely but
==153==  possible), you can try to increase the size of the
==153==  main thread stack using the --main-stacksize= flag.
==153==  The main thread stack size used in this run was 8388608.
```

4.1.3 HonggFuzz

HongFuzz identified numerous crashes using Address Sanitizer (Asan) and Undefined Behavior Sanitizer (UBSan). These crashes ranged from memory-related issues to invalid memory accesses, highlighting potential vulnerabilities. The following unique crashes where reported.

Crashes found with ASan that where already reported on GitHub:

```
SUMMARY: AddressSanitizer: SEGV ...
    /home/arianvalen/libemf2svg/src/lib/emf2svg_utils.c:1368:17 in reverse_utf8
```

```
SUMMARY: AddressSanitizer: heap-buffer-overflow ...
    /home/arianvalen/libemf2svg/src/lib/emf2svg.c:50:20 in U_emf_onerec_analyse
```

```
SUMMARY: AddressSanitizer: SEGV /home/ricktt/libemf2svg/src/lib/emf2svg_utils.c:1222:30 in ...
    fontindex_to_utf8
```

```
SUMMARY: AddressSanitizer: heap-buffer-overflow ...
    /home/ricktt/libemf2svg/src/lib/emf2svg_rec_control.c:35:15 in U_EMRHEADER_draw
```

```
SUMMARY: AddressSanitizer: heap-buffer-overflow ...
    /home/ricktt/libemf2svg/src/lib/uemf_utf.c:164:16 in wchar16len
```

```
SUMMARY: AddressSanitizer: stack-buffer-overflow ...
    /home/arianvalen/libemf2svg/src/lib/emf2svg_rec_control.c:35:15 in U_EMRHEADER_draw
```

New crash found with ASan which is not reported on GitHub:

```
SUMMARY: AddressSanitizer: heap-buffer-overflow ...
    /home/arianvalen/libemf2svg/src/lib/emf2svg_rec_control.c:92:67 in U_EMRHEADER_draw
```

Crashes found with UBsan that where already reported on GitHub:

```
SUMMARY: UndefinedBehaviorSanitizer: undefined-behavior ...
    /home/ricktt/libemf2svg/src/lib/emf2svg.c:50:20
```

```
SUMMARY: UndefinedBehaviorSanitizer: SEGV emf2svg_utils.c in fontindex_to_utf8
```

New crashes found with UBsan which is not reported on GitHub:

```
SUMMARY: UndefinedBehaviorSanitizer: undefined-behavior ...
    /home/arianvalen/libemf2svg/src/lib/emf2svg.c:59:21
```

```
SUMMARY: UndefinedBehaviorSanitizer: undefined-behavior ...
    /home/ricktt/libemf2svg/src/lib/emf2svg.c:723:30
```

```
SUMMARY: UndefinedBehaviorSanitizer: undefined-behavior ...
    /home/arianvalen/libemf2svg/src/lib/emf2svg_utils.c:1221:26
```

```
SUMMARY: UndefinedBehaviorSanitizer: undefined-behavior ...
    /home/arianvalen/libemf2svg/src/lib/emf2svg_utils.c:1360:34
```

```
SUMMARY: UndefinedBehaviorSanitizer: undefined-behavior ...
    /home/arianvalen/libemf2svg/src/lib/emf2svg_utils.c:1368:23
```

5 Reflection

5.1 Effectiveness of Fuzzing Tools

The experiment involved using three different fuzzing tools, namely AFL++, zzuf, and HonggFuzz, to assess their ability to discover flaws in the libemf2svg application. These tools exhibited varying degrees of effectiveness in identifying vulnerabilities.

AFL++ and HonggFuzz demonstrated a higher level of effectiveness in uncovering a larger number of flaws, including crashes and hangs. This suggests that these tools excel in systematically exploring the application's execution space, making them proficient at finding vulnerabilities.

In contrast, zzuf, while functional, proved to be less effective in discovering flaws. This could indicate that mutation-based fuzzers like zzuf may not be as adept at detecting certain types of vulnerabilities compared to evolutionary and feedback-driven fuzzers like AFL++ and HonggFuzz.

5.2 Comparison of Fuzzing Tools

In our evaluation, AFL++ and HonggFuzz emerged as superior options for finding more flaws and providing more detailed information about them. Their ease of use also stood out as a significant advantage over zzuf. Additionally, HonggFuzz's multithreading capabilities offered a substantial increase in throughput. The primary advantage of zzuf was its ability to easily reproduce found bugs.

5.3 Overheads of Instrumentation Approaches

Instrumentation approaches, such as Address Sanitizer (Asan) and Undefined Behavior Sanitizer (UBSan), introduced overhead in terms of execution speed. AFL++, for instance, experienced a nearly 50% reduction in speed due to instrumentation. However, this trade-off was justified by the improved detection rate of flaws. Early detection and enhanced security outweighed the temporary slowdown. Furthermore, it was demonstrated that crashes found without instrumentation could be used as input for fuzzing with sanitizing, potentially leading to the discovery of more unique crashes.

HonggFuzz, when used with sanitization, found more errors in a shorter amount of time compared to AFL++ with sanitizing. However, the use of sanitizing also resulted in more non-unique errors (reported as unique by both tools). Both AFL++ and HonggFuzz still uncovered a considerable number of unique errors when analyzed afterward by running all the crashes provided with the application compiled with sanitization.

5.4 Ease of Setup for Fuzzing and Instrumentation Tools

Setting up zzuf was relatively cumbersome compared to AFL++ and HonggFuzz. While installing the fuzzer was straightforward, running multiple files with different mutations required creating a custom bash loop, which added complexity. In contrast, AFL++ and HonggFuzz offered one-liner solutions for this task. Additionally, AFL++ and HonggFuzz provided detailed output in the terminal, while zzuf required manual result saving and post-inspection.

Compiling AFL++ and HonggFuzz, especially when integrating them seamlessly, could be somewhat challenging. However, in the tested application, the use of CMake proved to be convenient, making it relatively easy to incorporate various compile options, such as Address Sanitizer (ASan) and Undefined Behavior Sanitizer (UBSan).

5.5 Test Case Generation Speed

When considering the speed of test case generation, zzuf appeared significantly slower compared to AFL++ and HonggFuzz. HonggFuzz also exhibited a slight speed advantage over AFL++ in this aspect.

5.6 Sensitivity to Initial Inputs

The sensitivity of fuzzing tools to the initial set of valid inputs or the initial random number seed was observed in the experiment results. Depending on the input corpus used, variations were noted in the number of crashes detected. For instance, using the original EMF corpus resulted in the discovery of the most crashes (not counting found crashes as input). In contrast, using a small text file as input failed to produce any crashes, indicating the challenge of generating a file resembling the EMF structure in a reasonable amount of time. This suggested that many code paths remained untouched in such cases.

6 Conclusion

In conclusion, AFL++ and HonggFuzz stand out as highly effective fuzzing tools, providing superior flaw detection capabilities, ease of use, and efficient performance. The use of instrumentation approaches, while introducing some overhead, proved valuable in enhancing security through early flaw detection. However, the choice of fuzzing tool and the selection of initial inputs remain critical factors in the success of a fuzzing campaign.

References

[1] kakwa, "libemf2svg," 2021. [Online]. Available: https://github.com/kakwa/libemf2svg#libemf2svg