Fuzzing results

Cristian Daniele
Patrick Lodeweegs
Erik Poll
Digital Security group
Radboud University Nijmegen

Fuzzing – case studies

Group	Case study	input format
1	LunaSVG	SVG
3	libspng	PNG
4	svg2gcode	SVG
5	impr	BMP
8	simd	PNG
10	svg2ass	SVG
12	bmp2jpeg	BMP
15	stenography (encoding and decoding)	PNG and ZIP
16	OpenTTD	OpenTTD save file
19	PNGcrush	PNG
20	PDFALTO	PDF
21	PDF-Parser-C	PDF
23	ROC Toolkit	ROC audiostream

Fuzzing – tools used

Group	Case study	fuzzers	sanitisers
1	LunaSVG	afl++, Honggfuzz, zzuf	ASan
3	libspng	afl++, zzuf, Radamsa	ASan, MSan, valgrind
4	svg2gcode	afl++, Honggfuzz, zzuf, Radamsa, LibFuzzer	ASan, UBSan
5	impr	afl++, Honggfuzz, zzuf	ASan
8	simd	afl++, Honggfuzz	ASan, MSan
10	svg2ass	afl++, Honggfuzz, zzuf, Radamsa,	ASan, MSan
12	bmp2jpeg	afl++, HonggFuzz, zzuf	Asan
15	stenography	afl++, HonggFuzz, zzuf, Radamsa	Asan, valgrind
16	OpenTTD	afl++, Honggfuzz, zzuf	ASan, MSan
19	PNGcrush	afl++, HongFuzz, zzuf	ASan
20	PDFALTO	afl, Zzuf, Hongfuzz, Radamsa	ASan
21	PDF-Parser-C	afl, HonggFuzz, zzuf	ASan, MSan
23	ROC Toolkit	afl++, Libfuzzer, Radamsa, (Hongfuzz, Angora, zzuf)	ASan, UBSan

Fuzzing – bugs found?

Group	Case study	fuzzers	bugs found?
1	LunaSVG	afl++, Honggfuzz, zzuf	yes (in dependency)
3	libspng	afl++, zzuf, Radamsa	yes
4	svg2gcode	afl++, Honggfuzz, zzuf, Radamsa, LibFuzzer	yes
5	impr	afl++, Honggfuzz, zzuf	yes
8	simd	afl++, Honggfuzz	yes
10	svg2ass	afl++, Honggfuzz, zzuf, Radamsa,	yes
12	bmp2jpeg	afl++, HonggFuzz, zzuf	yes
15	stenography	afl++, HonggFuzz, zzuf, Radamsa	yes
16	OpenTTD	afl++, Honggfuzz, zzuf	no
19	PNGcrush	afl++, HongFuzz, zzuf	yes
20	PDFALTO	afl, Zzuf, Hongfuzz, Radamsa	yes
21	PDF-Parser-C	afl, HonggFuzz, zzuf	yes
23	ROC Toolkit	afl++, (Hongfuzz), Libfuzzer, (Angora), Radamsa, (zzuf)	yes

Did anyone report bugs?

Or even commit bug fixes?

Typical performance numbers (group 3)

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

afl++ is typically faster than the other tools

(though group 12 reported Honggfuzz was faster than afl++)

Typical performance numbers (group 4)

ID_Experiment	Tool + Sanitizer	Time	# Test Cases	Issues Found
#1	AFL++	3 hrs 37 mins	19.6 million	120 crashes, 0 hangs
#2	zzuf	3 hrs	4 million	856 crashes, 0 hangs
#3	Radamsa	3 hrs	4 million	124 crashes, 0 hangs
#4	HonggFuzz	4 hrs	1.68 million	693 crashes, 0 hangs
#5	AFL++ with ASan	3 hrs 9 mins	5.5 million	131 crashes, 0 hangs
#6	Radamsa with ASan	3 hrs	4 million	159 crashes, 0 hangs
#7	HonggFuzz with ASan	4 hrs	1.05 million	90 crashes, 0 hangs
#7	LibFuzzer with ASan	3 hrs	7 million	251 crashes, 0 hangs
#8	AFL++ with UBSan	10 hrs	15.3 million	124 crashes, 0 hangs
#9	Radamsa with UBSan	3 hrs	4 million	112 crashes, 0 hangs

Rule of thumb: if dumb fuzzers like zzuf and Radamsa find bugs then the code is pretty bad.

Is ASan/MSan overhead worth it?

ID_Experiment	Tool + Sanitizer	Time	# Test Cases	Issues Found
#1	AFL++	3 hrs 37 mins	19.6 million	120 crashes, 0 hangs
#2	zzuf	3 hrs	4 million	856 crashes, 0 hangs
#3	Radamsa	3 hrs	4 million	124 crashes, 0 hangs
#4	HonggFuzz	4 hrs	1.68 million	693 crashes, 0 hangs
#5	AFL++ with ASan	3 hrs 9 mins	5.5 million	131 crashes, 0 hangs
#6	Radamsa with ASan	3 hrs	4 million	159 crashes, 0 hangs
#7	HonggFuzz with ASan	4 hrs	1.05 million	90 crashes, 0 hangs
#7	LibFuzzer with ASan	3 hrs	7 million	251 crashes, 0 hangs
#8	AFL++ with UBSan	10 hrs	15.3 million	124 crashes, 0 hangs
#9	Radamsa with UBSan	3 hrs	4 million	112 crashes, 0 hangs

Sanitizers had a huge impact on afl++ performance - either with ASan or UBsan it performed almost 4 times slower with not a huge number of additional bugs found.

Typical ASan & MSan overhead (group 10)

Fuzzer	sanitizer	Time	# of execs	# of cycles done	# of crashes	# of hangs
afl++	none	9h 48min	8.43M	295	18	16
afl++	ASan	10h 48min	2.90M	6	26	13
afl++	MSan	8h 19min	573K	19	11	3

Valgrind (group 3)

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

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

Valgrind is probably too slow (compared to ASan & MSan) to use while fuzzing

It may still be useful for post-hoc analysis of bugs

Surprising dumb fuzzer successes (group 3)

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

Sometimes - surprisingly - zzuf and Radamsa beat afl++

Unsurprisingly, they then find many instances of the same bug

Still, I think group 12 is right to say that

> Another observation is that the 'dumb' fuzzers such as Zzuf are no longer worth using. Evolution-based fuzzers can find more bugs faster. Obviously, given enough time, Zzuf would probably find a random mutation that does indeed crash the program, but this may take a long time.

Unexplained mysteries? [group 19]

For honggfuzz, the average speed on the runs with ASan was higher than without it and honggfuzz did manage to find a crash. Adding a dictionary significantly decreased the speed and did not add anything to honggfuzz's ability to find crashes. For honggfuzz it seems that the speed and quality of the fuzzing is at its best when using a sanitizer without a dictionary.

ASan overhead [group 20]

Surprising lack of ASan overhead

ID_experiment	Tool	Time	Number of test cases	Issues found
#1	AFL	12hr	12k	no errors, 114 crashes, 51 hangs
#3	AFL + ASan	3hr	4k	no errors, 1 crashes, no hangs
#2	zzuf	15min	5k	2.319k errors, no crashes, no hangs
#5	zzuf + ASan	15min	5k	2.319k errors, no crashes, no hangs

ASan overhead [group 19]

Even more surprising: ASan speeding things up

For honggfuzz, the average speed on the runs with ASan was higher than without it and hongg-fuzz did manage to find a crash. Adding a dictionary significantly decreased the speed and did not add anything to honggfuzz's ability to find crashes. For honggfuzz it seems that the speed and quality of the fuzzing is at its best when using a sanitizer without a dictionary.

Spot the security flaw [CVE-2024-320, group 8]

```
while (_data[_pos] == value && _pos < _size)
    _pos++;
return _pos < _size;
```

Security fix

```
while (_pos < _size && _data[_pos] == value )
    _pos++;
return _pos < _size;
```

Seed selection [group 4]

We ran fuzzers on four small (max. 370 bytes) `.svg` files and in our opinion they were enough. We tried to introduce a more complex file (515 Kb), but then fuzzers were performing very poorly (afl++ sometimes even refused to work, because a single test case took more than a second). Changing initial seeds didn't do much - eventually fuzzers found the same bugs.

It is to be expected that a file of 500 Kb is too big for fuzzing

It is nice to see that different seeds still reveal the same bugs

Seed selection: using ChatGTP? [group 19]

Interesting idea, but not sure if it will be any help

We ran all of the fuzzers on the latest version of pngcrush (1.8.1) and the earliest (on Github's releases) which is 1.7.27. The reason for using this specific version is because there is a CVE on this version with a off-by-one error [1]. Unfortunately, we could not find or reproduce a PNG to trigger the CVE. We used 17 files as input or seed files. These files are valid PNGs, specific test cases and potential test cases. We got these files using two methods. We asked ChatGPT to try and generate files which could result in problems for our program, this resulted in 5 files. We also searched GitHub repositories for images used for testing and images that were reported to have interesting outcomes.

Checksum issues? [group 19. And other PNG groups?]

As an optimizer for PNG images, pngcrush requires a valid PNG file as input to process it as it will otherwise be rejected. However, utilizing fuzzing methods to generate a valid input is far from a trivial task, as the PNG format is known for its complexity and strict adherence to its specification. Every PNG file is organized into a series of chunks and each chunk includes a checksum for error detection. Mutating the chunk data without updating the checksum will thus cause the entire chunk, and potentially the entire file, to be rejected. The PNG dictionary of AFL++ is not of help here as it can only enforce the test cases to adhere to the general png chunk structure, but cannot recompute the checksums.

As a benchmark to gauge how much of the generated PNG's by AFL++ pass pngcrush's validation checks, we adapted the codebase with additional logging output triggered at every execution of the main loop. We ran this set-up for 20880 iterations, of which the main loop was reached 7 times, yielding an efficiency of 0.034%. It is important to note that this percentage is only a rough indicator of the test case generation efficiency as the selection of seed files has a lot of impact on the effectiveness of evolutions and only one set of seed files was used for this experiment.

Coverage (group 16)

Fuzzer	#Tests	Crashes	Hangs	Map coverage
AFL++	1.52M	0	12	4.56%
AFL++ with ASan	831K	0	23	0.71%

It is also somewhat surprising that AFL++ without a sanitizer reported a significantly higher map coverage than AFL++ with ASan (4.56% and 0.71% respectively). We think this is related to the lower number of executions with ASan, but it is also possible that using a sanitizer changes the way new inputs are selected (preferring inputs that AFL++ thinks may show issues for the given sanitizer), leading the fuzzer to getting stuck in exploring a smaller part of the code base. Another possibility is that the sanitizer adds extra branches for its checks, that are never triggered, thus lowering the coverage.

Even if we account for the fact that we focussed on only a small part, it is still surprising that there is such a large difference between the two ways of fuzzing. We should also note that the fuzzers ran for a relatively short period of time, and that the low coverage may also be (partially) caused by the short run time.

Crash/error = security issue or not? [group 23]

The problem we had with HonggFuzz is that every error handle or normal return would be marked as a crash while we would not mark it as a crash. Take for example figure 6, here we can see `roc-send` handling the invalid input given by Honggfuzz and marking it as an error. This is not a crash and is intended behavior. But Honggfuzz sees this as a crash.

```
root@12e0bd9d8c1b:/roc-toolkit# roc-send --source=rtp://127.0.0.1:2345 --input=file:///fil
edump/HongFuzz/findings/SIGABRT.PC.7efc7a4a3f1c.STACK.1acfac32bc.CODE.-6.ADDR.0.INSTR.mov_
___%eax,%r14d.fuzz
14:20:09.631 [14] [err] roc_address: parse io uri: invalid path
14:20:09.632 [14] [err] roc_send: invalid --input file or device URI
```

Crash on assertion error = security issue? [group 8]

AFL also find quite some duplicates crashes, however it did also start with the same set of files each run. This lead to the same crashes multiple times. It did also sometimes find some assertion errors, however we did not find that relevant enough to investigate since those were not possible to turn off in the configuration.

4.1 Flaws Detected

- **Bug 1**: In the function `nsvg__parseFloat`, a `NULL` string was passed to `sscanf`, resulting in a segmentation fault. This occurred due to insufficient validation of input before parsing.
- **Bug 2**: Also in `nsvg__parseFloat`, malformed strings caused `sscanf` to misbehave. This was due to improper handling of unexpected input formats, leading to undefined behavior.
- **Bug 3**: In the function `nsvg__parseAttr`, a `NULL` value was passed as the `value` parameter in comparisons or calls to functions like `strcmp`. This caused segmentation faults due to invalid memory access.
- **Bug 4**: In `nsvg__parseSVG`, attempting to parse `NULL` or malformed attribute values (e.g., empty strings) without validation led to segmentation faults during operations such as `sscanf` and `strstr`.
- **Bug 5**: In `nsvg__parseFloat`, the lack of validation for the format of input strings allowed invalid characters to pass through, leading to misbehavior and potential crashes during parsing.

4.1 Flaws Detected

- **Bug 1**: In the function `nsvg__parseFloat`, a `NULL` string was passed to `sscanf`, resulting in a segmentation fault. This occurred due to insufficient validation of input before parsing.
- **Bug 2**: Also in `nsvg__parseFloat`, malformed strings caused `sscanf` to misbehave. This was due to improper handling of unexpected input formats, leading to undefined behavior.
- **Bug 3**: In the function `nsvg__parseAttr`, a `NULL` value was passed as the `value` parameter in comparisons or calls to functions like `strcmp`. This caused segmentation faults due to invalid memory access.
- **Bug 4**: In `nsvg__parseSVG`, attempting to parse `NULL` or malformed attribute values (e.g., empty strings) without validation led to segmentation faults during operations such as `sscanf` and `strstr`.
- **Bug 5**: In `nsvg__parseFloat`, the lack of validation for the format of input strings allowed invalid characters to pass through, leading to misbehavior and potential crashes during parsing.

4.1 Flaws Detected

- **Bug 1**: In the function `nsvg__parseFloat`, a `NULL` string was passed to `sscanf`, resulting in a segmentation fault. This occurred due to insufficient validation of input before parsing.
- **Bug 2**: Also in `nsvg__parseFloat`, malformed strings caused `sscanf` to misbehave. This was due to improper handling of unexpected input formats, leading to undefined behavior.
- **Bug 3**: In the function `nsvg__parseAttr`, a `NULL` value was passed as the `value` parameter in comparisons or calls to functions like `strcmp`. This caused segmentation faults due to invalid memory access.
- **Bug 4**: In `nsvg__parseSVG`, attempting to parse `NULL` or malformed attribute values (e.g., empty strings) without validation led to segmentation faults during operations such as `sscanf` and `strstr`.
- **Bug 5**: In `nsvg__parseFloat`, the lack of validation for the format of input strings allowed invalid characters to pass through, leading to misbehavior and potential crashes during parsing.

Lack of Input Validation & Writing Style

- Talk about 'lack of validation before parsing' is potentially misleading

  Validation is an *integral part of* parsing.
  If you first validate data & then only parse valid data
  you end up with two parsers

  Hence the *'Parse, don't validate'* slogan

- General writing tip: don't try to say the same thing in different ways
  - This only confuses the reader
  - Forget what they told you in high school about varying words
  - Keep prose boring & repetitive, consistently using the same terminology

Uniqueness

Are 'unique' bugs (as claimed by afl++ or Honggfuzz) really   unique?

Often not!

Hangs / time-outs

Are hangs/time-outs really hangs?

Often not!

Beware of different goals of instrumentation

Instrumentation is used for two very different purposes in fuzzing:

1) to provide feedback to guide the mutation process

   eg afl's or Honggfuzz's standard instrumentation to observe branch coverage

2) to detect bugs

   eg the instrumentation added by sanitisers such as ASan, MSan, UBSan

Watch your prose

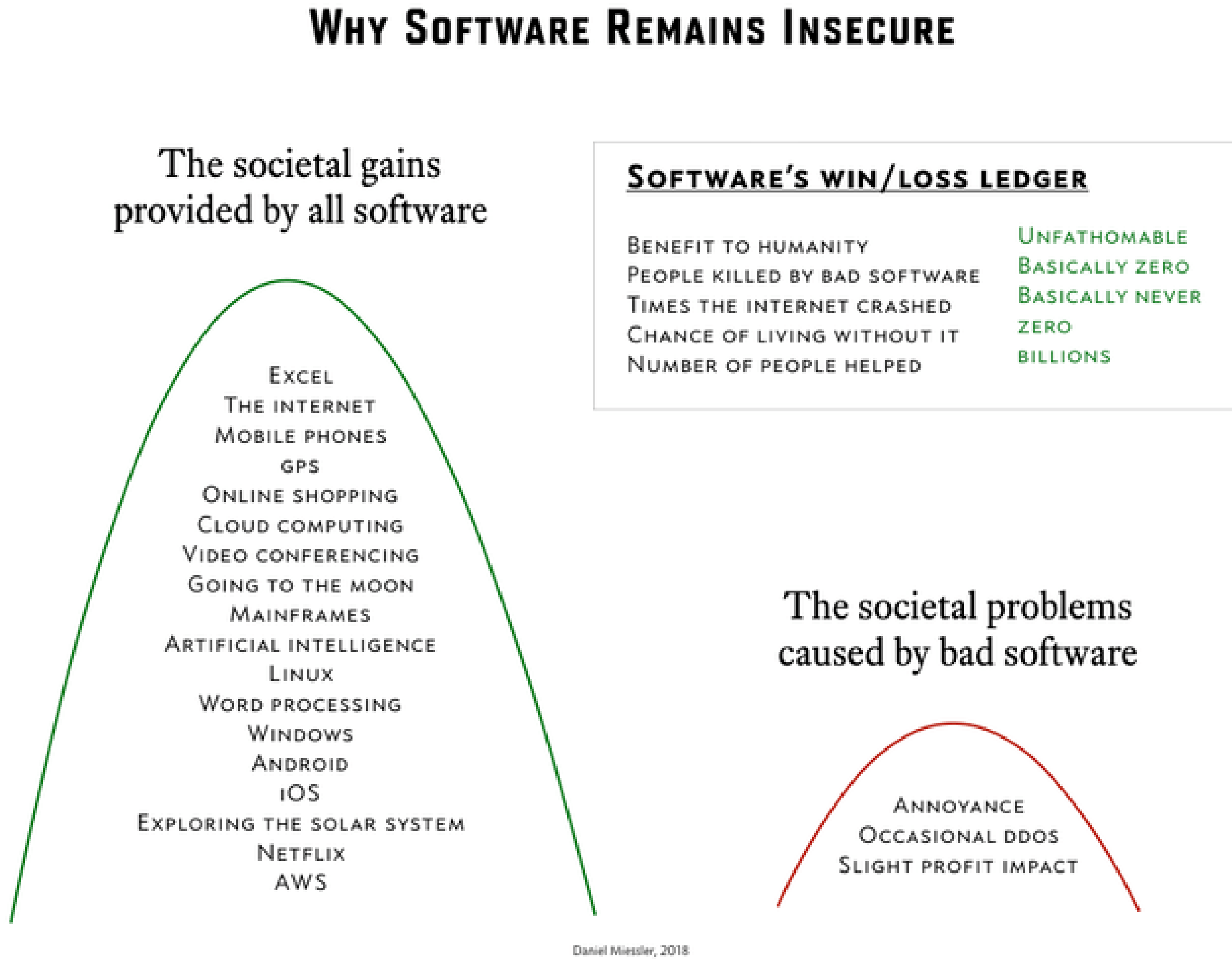
As the reader progresses through the following sections, they will gain valuable insights into the unique strengths and limitations of each fuzzing tool, in addition to gaining a comprehensive understanding of the security posture of the [redacted] tool. The findings in this report have the potential to contribute significantly to the overarching discourse on software security, enabling efforts aimed at increasing the resilience of critical software components.

By leveraging the formidable capabilities of fuzzing techniques and tools, this report represents a critical step in strengthening open-source applications like [redacted] and [redacted] and in strengthening the security and reliability of file processing within today's dynamic digital landscape.

Some end of year reflection

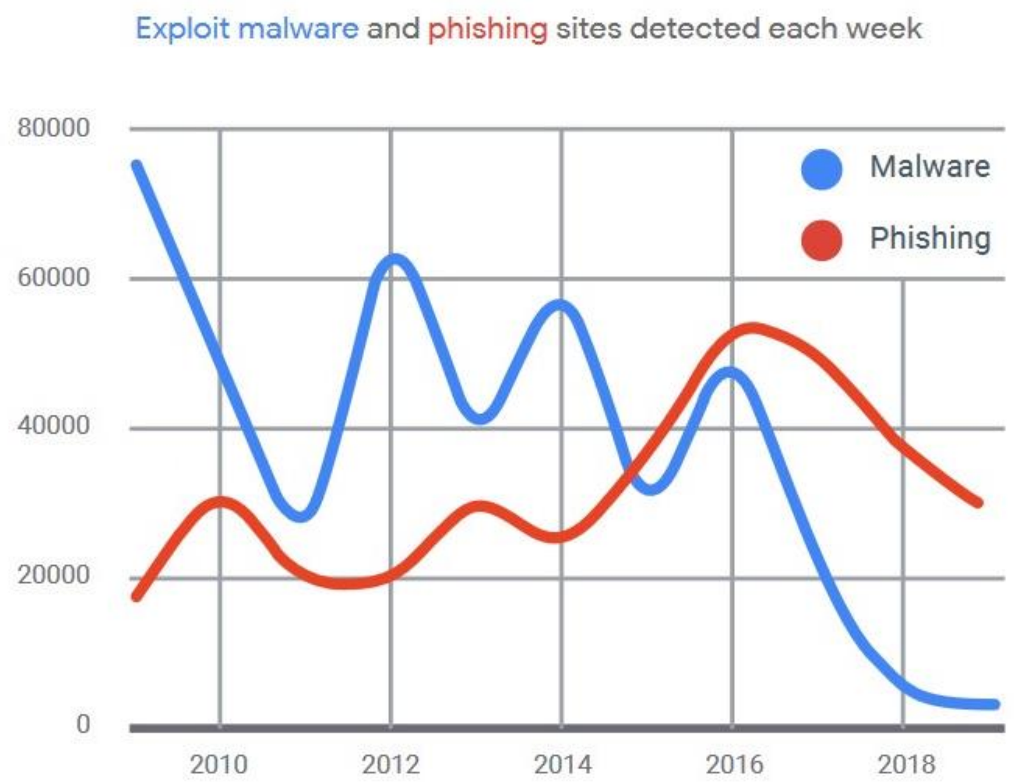
-

not exam material



Daniel Miessler, https://danielmiessler.com/p/the-reason-software-remains-insecure/

Phishing overtook exploit-based malware in 2016



Source: Safe Browsing (Google Transparency Report)