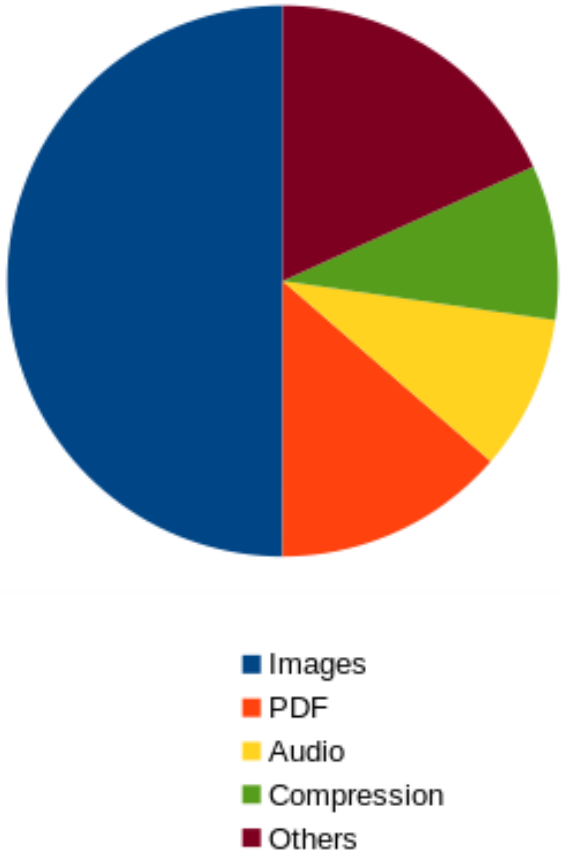
Fuzzing results

Patrick Lodeweegs
Faiz Muhammad
Erik Poll
Digital Security group
Radboud University Nijmegen

Case studies & input formats

	Case study	Formats
1	gifsicle	gif
1	gifview	gif
2	TinyEXIF	jpg
3	Lorina	Verilog
4	PoDoFo	pdf
6	Zlib	gz,
6	TinyTiff	tiff
7	Mini-XML	XML, ZST
7	TInyZZZ	LPAQ8, LZMA, LZ4, ZSTD
8	HTMLDOC	HTML
9	tinyexr	EXR
10	GIL	jpg png
11	tPNG	PNG
12	jpegoptim	jpeg
13	LibGD	BMP, GIF, WBMP
16	pdfresurrect	pdf
17	stb_vorbis	ogg
19	DjVuLibre	DjVu
20	libnsgif	gif
23	mupdf	pdf
24	Dav1d	ivf
25	Lib3mf	3mf



Tools used

Group	Fuzzers	Sanitizers	Afl-options
1	AFL++, Radamsa, Honggfuzz	ASan	
2	AFL++, Radamsa, Honggfuzz	ASan	
3	AFL++, Honggfuzz, zzuf	ASan	
4	AFL++, Honggfuzz, zzuf	ASan, UBSan	CmpLog, MOPT
6	AFL++, Radamsa, zzuf	ASan	
7	AFL++, Libfuzzer, MOPT	ASan, UBSan	
8	AFL++, Honggfuzz, zzuf	ASan	CmpLog, CompCov
9	AFL++,HongFuzz, zzuf	Asan	
10	AFL++, zzuf, Honggfuzz, FormatFuzzer	ASan	
11	AFL++, HongFuzz, zzuf	ASan, UBSan	
12	AFL++, Radamsa	ASan, MSan	deterministic mode
13	AFL++, Honggfuzz, zzuf	ASan	
16	AFL++, AFL, Honggfuzz, zzuf	ASan, Msan, UBSan	
17	AFL++, zzuf, Radamsa, Hongfuzz	ASan, MSan, UBSan	
19	AFL++, zzuf, Honggfuzz	ASan	
20	AFL++, zzuf, Honggfuzz	ASan, MSan	
23	AFL++, Honggfuzz, zzuf	ASan	
24	AFL++,HongFuzz, zzuff	ASan	
25	AFL++, zzuf, Hongfuzz	ASan	

New/known bugs found?

Group	Case study	In Oss-Fuzz?	New bugs?	Known bugs?
1	gifsicle	No	No	Yes
2	TinyEXIF	No	Yes	Yes
3	Lorina	No	?	
4	PoDoFo	No	Not sure	
6	Zlib	Yes	No	Yes
	TinyTiff		Maybe	Maybe
7	Mini-XML	No	Maybe	Maybe
	TinyZZZ		Yes	
8	HTMLDOC	No	Yes	Maybe
9	tinyexr	No	Probably	
10	GIL	No	Maybe	Yes
11	tPNG	No	Yes	
12	jpegoptim	Yes	No	No?
13	LibGD	Yes	Yes	Yes
16	pdfresurrect	No	No	Yes
17	stb_vorbis	No	Yes	Yes
19	DjVuLibre	No	No	Yes
20	libnsgif	No	No	No
23	mupdf	Yes	Yes	Maybe
24	Dav1d	Yes	Yes	No
25	Lib3mf	No	Yes	No

Did anyone report bugs?

Or commit bug fixes?

Crashes vs bugs

Were the reported crashes always bugs?

Hangs / time-outs

Are hangs/time-outs really hangs?

Hangs / time-outs

Are hangs/time-outs really hangs?

Often not!

Is ASan worth using?

Are ASan, MSan, UBSan worth using?

Bad for performance, in #tests/minute,
good for detection

- ***How big was the performance hit?***
- ***Was the performance hit worth it?***

Typical ASan, MSan, UBSan performance hit (group 16)

ID	Tool	Sanitizer	Input files	Time	Number of generated test cases	Issues Found (unique crashes)
3	AFL++	-	big.pdf	2h	30.04M	942K (11 unique)
4	AFL++	ASan	big.pdf	2h	13.57M	568K (15 unique)
5	AFL++	MSan	small.pdf	2h	1.14M	5.78k (8 unique)
6	AFL++	UBSan	small.pdf	2h	26.75M	1.06M (139 unique paths)

Note: this performance hit is also a rough indication of the cost of memory-safety checks

ASan overhead can pay off (group 25)

More crashes with ASan

ID_experiment	Tool	Time	Number of test cases	Issues found
#2	AFL without ASAN	2 hours	2.68M	0 crashes
#3	AFL with ASAN	2 hours	876K	1 crashes

But… fuzzing is random, so results can be a coincidence, as shown by,

ID_experiment	Tool	Time	Number of test cases	Issues found
#1	AFL with ASAN	1800 seconds	191452	3 crashes
#3	AFL with ASAN	2 hours	876K	1 crashes

(Style side note: consistent conventions for times, numbers… are nice)

No success with ASan (group 19)

With the addition of ASan in the experiments, we hoped for more crashes and a better detection rate. But it only slowed the fuzzing down by approximately 4 times compared to the runs without ASan. And the saddest part is that it didn't even result in more crashes or a better detection rate.

Tool	Time	Testcases	Iterations	Issues found
HonggFuzz	2.5 hrs	264	6.37M	36 unique crashes
HonggFuzz with ASan	13 hrs	266	9.84M	4 unique crashes
AFL++	21hrs	1189	19.1M	27 saved crashes
AFL++ with ASan	22 hrs	1077	13.6M	22 saved crashes

But …

- **Location of error will be more precise with ASan**
- **Several crashes without ASan may map to one crash with ASan (that is the root cause)**

ASan impact on coverage? (group 13)

We ran HonggFuzz both with and without ASan and fuzzed the latest version of LibGD as well as the older release 2.2.4. We used initial seed files from the 'small' GitHub repository [2], which contains minimal seed files for various formats, and created a few small WBMP files ourselves. HonggFuzz expanded these into a corpus of 50 files. For all formats, the coverage quickly got stuck between 40–50% (within about two minutes) without ASan, and with ASan we reached around 65% coverage. We let the fuzzer run for a few more hours, but the coverage did not improve and no issues were found. The low coverage may be due to the fact that we fuzzed only a small part of the input formats, which in turn may explain why no issues were detected.

Interesting that ASan improved coverage for Honggfuzz

Not sure why/how that could happen?

Are dumb fuzzers
(zzuf, Radamsa,...)
still worth using?

Success with zzuf (group 13)

When we started our analysis, we failed to find LibGD in projects covered by OSS-Fuzz. However, we discovered after the fact that LibGD has been partly covered by OSS-Fuzz with 34% code coverage [3]. OSS-Fuzz reports to have used AFL, Honggfuzz and libFuzzer. This may explain why zzuf managed to find issues in the latest version of LibGD, whereas our tests with AFL++ and Honggfuzz did not.

Our test results indicate that basic mutation-based fuzzers such as zzuf are still good enough to find flaws in some applications, albeit by running many more test cases over a much longer period of time than we would with a modern evolutionary fuzzer like AFL++.

The bug found involved an integer overflow:

> If we save an image as a BMP file with RLE compression while the width is more than half the maximum integer value, the function will multiply the width by two and cause an integer overflow. This integer is then used in a memory allocation call and causes unpredictable behavior.

Success with Radamsa and zzuf (group 17)

stb_vorbis operating on Ogg Vorbis format

Line	Row	Radamsa	Zzuf	HonggFuzz	AFL++
960	16		✓	✓	✓
966		✓	✓		
977	11		✓	✓	✓
1244	8	✓			✓
1248	8				✓
1586	24	✓	✓		✓
1587	24		✓		✓
3061	8	✓			
4214	21	✓			✓
5171	15	✓			
5285	21	✓			

Success with Radamsa + UBSan (group 17)

Radamsa + UBSan finds plenty of bugs

Crash type	Line	Times Found	CVE
Left shift of negative value (1)	3061:8	4294	No
Left shift of negative value (2)	1586:24	591	No
Integer overflow (3)	5171:15	214	No
Invalid `free()` pointer (4)	966	6	2023-47212
Integer overflow (5)	5285:21	147	No
Load of null pointer	4214:21	196	2023-47212
`SEGV` on unknown address	966	7	2023-47212
Double free or corruption	966	2	2023-47212
Integer overflow (9)	1244:8	5	No

Success with Radamsa + MSan or ASan (group 17)

2.1.2 msan

Time: 7796s
Iterations: 50693

Crash type	Line	Times Found	CVE
Too large requested allocation	3670	290	2023-47212
SEGV on unknown address	4214	1	2023-47212

2.1.3 asan

Time: 7826s
Iterations: 118269

Crash type	Line	Times Found	CVE
Memory leak	N/A	2324	No
Too large requested allocation (2)	3670	713	2023-47212
SEGV on unknown address	4214	1	2023-47212

Random nature of fuzzing (group 2)

ID	Tool	Seeds	Time	Testcases/ Mutations	Crashes	Unique Vulnerabilities
#5	Honggfuzz + ASan	34	1hrs	2,330,059	137	4
#6	Honggfuzz + ASan	34	2hrs	1,120,748	123	3
#7	Honggfuzz + ASan	34	4hrs	1,280,253	96	2
#8	Honggfuzz + ASan	34	261 hrs	5,074,002	102	3

Sometimes short runs find bugs that long ones miss

(I think the 34 seeds were identical, right?)

Experiment #8 used different (cloud) server, Digital Ocean

Sweet spot for zzuf mutation rate (group 4)

ID	Corpus	Rate	# unique error lines	# total crashes
2.6	Mozilla	0.01	199	0
2.7		0.001	997	9
2.8		0.0001	703	22
2.9		0.00001	338	11
2.10		0.000001	137	1

Advantage of coverage guided (aka greybox) fuzzers

- Dumb mutational fuzzers, but also grammar-based fuzzers, tend to produce many instances of the same bug

- Coverage-guided fuzzers can weed out (some? many?) duplicates than to the observation of execution paths/code coverage

CmpLog & CompCov
(group 8 & 10)

AFL++ flag CmpLog (group 8)

New crash only found by afl++, when using CmpLog option

COMPARISON ACROSS FUZZERS (HTMLDOC V1.9.21).

Metric	AFL++	AFL++ (ASan)	AFL++ (CompCov)	AFL++ (CmpLog)	honggfuzz	zzuf
Initial seeds (count)	8	8	8	8	8	1
Total runtime (hrs)	479	563	563	563	168	139
Total test cases (approx)	136m	95.6m	162m	115m	1.92m	46.4m
Average execs/hr (approx)	284k	170k	288k	204k	11k	334k
Unique crashes (count)	0	0	0	26	0	0
Hangs (count)	2	3	1	7	2	0
Reproducible crashes	0	0	0	1	0	0
ASan reports	-	0	-	-	0	-
Crashes per million tests	0	0	0	0.000000226	0	0

Also,

- **Typical slowdown with ASan**
- **Extreme slowdown with Honggfuzz**
- **Does afl++ still find new paths after 500+ hrs?**

HTMLDOC

New crash found by afl++ with CmpLog

- HTML input with very deeply nested tags

  `<a><a><a><a><a><a>. . . . . .`

  causes a crash, because deep recursion leads stack overflow

- Such a test case is interesting for any program processing HTML or XML

  A problem might be unavoidable in such cases,
  but gracefully exiting may mitigate DoS issues

  (Eg, does your entire browser crash on such a file
  or will the rendering in one tab be aborted because of time-out?)

CmpLog in afl++

- Values used in comparisons, eg

```
if (input == 2025) {...}
```

  are recorded & used in further mutations

- The comparison value recorded & used was probably <
- Adding < or <a> to the dictionary of afl++ might also help the fuzzer to find the problematic input?

- The way afl(++) observes branch coverage means each nested call is seen as new branch coverage; this is important to discover this problem with deep nestings

CompCov

Another afl++ option is CompCov aka laf-intel

This splits comparisons.

For instance

```
if (input <= 2025) {...}
```

becomes

```
if (input < 2025) { if (input == 2025 {..}}
```

AFL++ docs suggests CmpLog works better than CompCov, as was the case for group 8

CompCov was used by group 4

- Impact of corpus: only Mozilla corpus reveals crashes
- Weirdly consistent edge coverage
- MOpt seems less effective compared to CompCov

ExperimentId	JobId	Corpus	Fuzzer	Time	Edge Coverage	Crashes	Timeouts
1.1	corpus-clean	PDF 2.0	Master	92h	4.71%	0	0
1.2	Run	Mozilla	Master	53h	6.13%	435	4
1.3	corpus-clean	PDF 2.0	MOpt_Cmp	92h	4.71%	0	0
1.4	Run	Mozilla	MOpt_Cmp	53h	6.13%	406	1
1.5	corpus-clean	PDF 2.0	ASAN	92h	4.71%	0	3
1.6	Run	Mozilla	ASAN	53h	6.13%	188	2
1.7	corpus-clean	PDF 2.0	CompCov	92h	4.71%	0	0
1.8	Run	Mozilla	CompCov	53h	6.13%	464	2
1.9	corpus-clean	PDF 2.0	PS Slaves	92h	4.71%	0	1 (rare) / 1 (coe)
1.10	Run	Mozilla	PS Slaves	53h	6.13%	5.5k (440 each)	15 in sum

Some exotic fuzzing options

Formatfuzzer group 10

FormatFuzzer is very good at the generation and parsing of binary inputs according to some template. This template can then be used to make a custom mutator. Thus, if we give FormatFuzzer the template for `png` and `jpeg` files, we can try to see what happens if we use a custom mutator in AFL++ and Honggfuzz . FormatFuzzer in itself is also a fuzzer, but we only tried combining it with AFL++ and Honggfuzz .

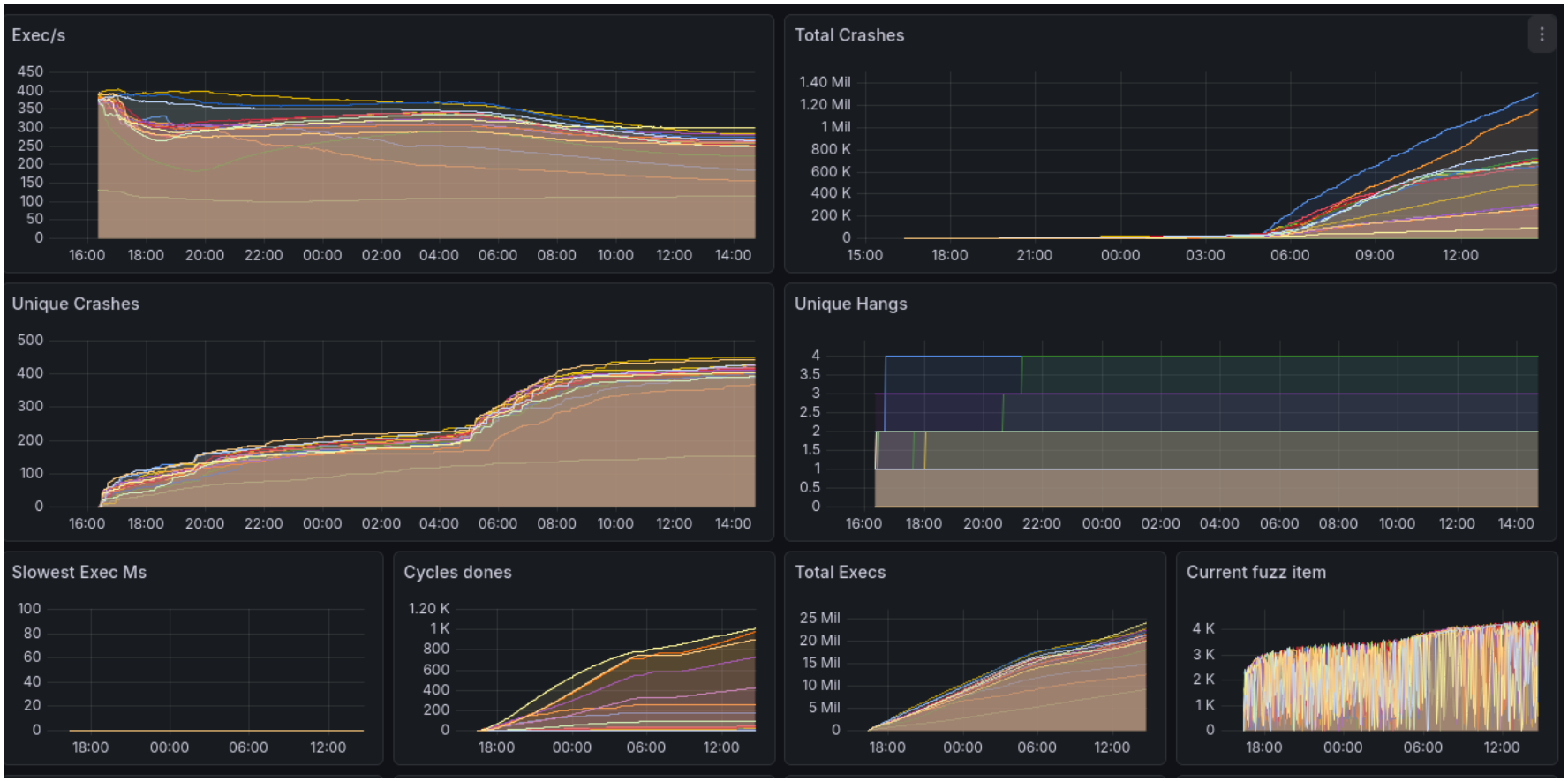
In experiment #4, we used FormatFuzzer to generate random `png`images as the input for the Honggfuzz experiment. Note that this is not a mutation based fuzzing strategy; each of the inputs is generated randomly. This was a bit of a shot in the dark, since it does not start from known edge-cases, but led to us finding some segmentation faults.

So it seems adding this grammar-based FormatFuzzer paid off?

- **Not quite clear from report if some (one?) seg-faults were also found without any use of FormatFuzzer.**

Maybe a better seed collection might have helped?

group 4 used dashboard to monitor progress



group 1

Only group to encounter a GUI, an work a way around it

2.5 Gifview

Moreover, a specific CVE occurs in gifview, which depends on a graphical interface. We've learned that fuzzing anything depending on a graphical interface is hard, since there's no way for the fuzzer to understand what to do when something is visualised (it might not be able to continue after a visualisation is supplied). We created a small testing harnas by editing the source code of gifview to make the gifsicle immediatly terminate after supplying the visual representation. This is done by commenting out `if (viewers) loop();` at line 1416 in version 1.89. This disables the looping of the program and ensures that gifview terminates after displaying the GIF image.

Some findings

TinyEXR (group 9)

ID experiment	Tool	Time	Number of test cases	Issues found	Mutations
1	AFL++	7 hours	48 million	12.9k	364
2	AFL++ with ASan	8 hours	19.8 million	5111	331
3	zzuf	12 hours	2.9 million	0	2.9 million
4	zzuf with ASan	12 hours	2.9 million	0	2.9 million
5	HongFuzz	9 hours	24 million	0	257
6	HongFuzz with ASan	9 hours	8 million	0	1796

Unusual to find so many issues;

(Were these unique issues? Or are 'mutations' the issues?)

TinyEXR (group 9)

Bug revealed by afl++ is a double-free issue

Proposed fix to make the code more defensive

```
int FreeEXRImage(EXRImage *exr_image) {
  if (exr_image == NULL) {
    return TINYEXR_ERROR_INVALID_ARGUMENT;
  }
  if (exr_image->next_level) {
    FreeEXRImage(exr_image->next_level);
    delete exr_image->next_level;
    exr_image->next_level = NULL; // fix
  }
  for (int i = 0; i < exr_image->num_channels; i++) {
    if (exr_image->images && exr_image->images[i]) {
      free(exr_image->images[i]);
      exr_image->images[i] = NULL; // fix
    }
  }
  if (exr_image->images) {
    free(exr_image->images);
    exr_image->images = NULL; // fix
  }
```

zlib (group 6)

Experiment	Tool	Test Cases	Time	Hangs	Crashes	Version
1	Zzuf (no ASan)	105k	10 min	0	0	1.3.1.1
2	Radamsa (no ASan)	105k	30 min	0	0	1.3.1.1
3	AFL++ (no ASan)	1.1M	5 min	0	0	1.3.1.1
4	Zzuf + ASan	105k	15 min	0	0	1.3.1.1
5	Radamsa + ASan	105k	25 min	0	0	1.3.1.1
6	AFL++ + ASan	1.1M	40 min	0	0	1.3.1.1
7	Zzuf + Asan	200k	2hr 30min	0	0	1.1.3
8	Radamsa + Asan	200k	2hr 25min	0	0	1.1.3
9	AFL++ + Asan	1.05M	8 min	0	1	1.1.3
10	Zzuf + ASan	200k	2hr 55 min	0	0	1.2.11
11	Radamsa + ASan	200k	2hr 50 min	0	0	1.2.11

- **Typical performance differences with AFL beating others**
- **Known double free bug found in old version 1.1.3**

We had success in finding VU#368819 [3], a double-free memory bug in zlib 1.1.3. Our finding it seemed heavily dependent on the corpus used; a simple corpus of a dummy gzip file seemed most successful in comparison to larger ones, and whether we incorporated specially crafted small files seemed inconsequential.

zlib (group 6)

Analysed bug + found the solution in the next version

Regardless of the success of `inflate_trees_dynamic`, the relevant data would be freed. Due to the built-in free within the inflate trees that triggers upon failure, this overall results in a double-free vulnerability. As documented in the vulnerability disclosure report, this is rather difficult to exploit meaningfully, and is most practical as a way to cause a Denial of Service in zlib applications.

```
1         t = inflate_trees_dynamic(257 + (t & 0x1f), 1 + ((t >> 5) & 0x1f),
2                                   s->sub.trees.blens, &bl, &bd, &tl, &td,
3                                   s->hufts, z);
4         ZFREE(z, s->sub.trees.blens);
5         if (t != Z_OK)
6         {
7           if (t == (uInt)Z_DATA_ERROR)
8             s->mode = BAD;
9           r = t;
10          LEAVE
11        }
```

```
1         t = inflate_trees_dynamic(257 + (t & 0x1f), 1 + ((t >> 5) & 0x1f),
2                                   s->sub.trees.blens, &bl, &bd, &tl, &td,
3                                   s->hufts, z);
4         if (t != Z_OK)
5         {
6           if (t == (uInt)Z_DATA_ERROR)
7           {
8             ZFREE(z, s->sub.trees.blens);
9             s->mode = BAD;
10          }
11          r = t;
12          LEAVE
13        }
```

group 12

Mac OS much slower than Ubuntu (with WLS)

#11	AFL with ASan	22 h	7,56M	0 crash, 1 hang	WSL2 Ubuntu	Set #2
#12	AFL++	74 h	23k	0 crash, 0 hang	macOS 15.7 Sequoia	Set #3 (3 random images)
#13	Radamsa	17 h	1M	0 crash, 0 hang	WSL2 Ubuntu	Set #2

2.8 Experiment 11

This experiment was completed on MAC OS platform running AFL++ using the same repository on its latest version for MAC OS, providing a seed of 3 different images with different sizes, different resolutions. Using MAC OS 15.7 Sequoia, allowing flags for errors on applications while disabling automatic crash reports sent to Apple. (Fig 5)

After 3 continuous days, 2 hours and 38 minutes of fuzzing, it was aborted by us, reporting an estimated 21K cases, stating 25.71% coverage achieved, from which it found 0 crashes and 0 timeouts. This has a contrasted result compared to the experiments run on the Ubuntu platform. On fig 6 we can see the coverage reports for each case and the results after abortion.

Different goals of instrumentation

Instrumentation is used for two very different purposes

1) to provide feedback to guide the mutation process

   eg afl's or Honggfuzz's instrumentation for coverage guidance of the mutation process

2) to detect bugs

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

Two benefits over coverage guidance

1) improving the mutation process to find new paths

2) less duplication of bugs

   unique bugs might not be quite unique, but alf will produce far fewer duplicates than Radamsa or zzuf

How to ensure covering enough code?

The code tested by zzuf focused on opening and saving files, but LibGD contains many functions to manipulate an image once it is loaded in memory as well. It may be interesting to fuzz code utilizing these functions to determine whether these can cause issues too.

Different features are likely to use the same parser,

buit some features may not need to parse all data.

Writing tips

- Always number pages & sections

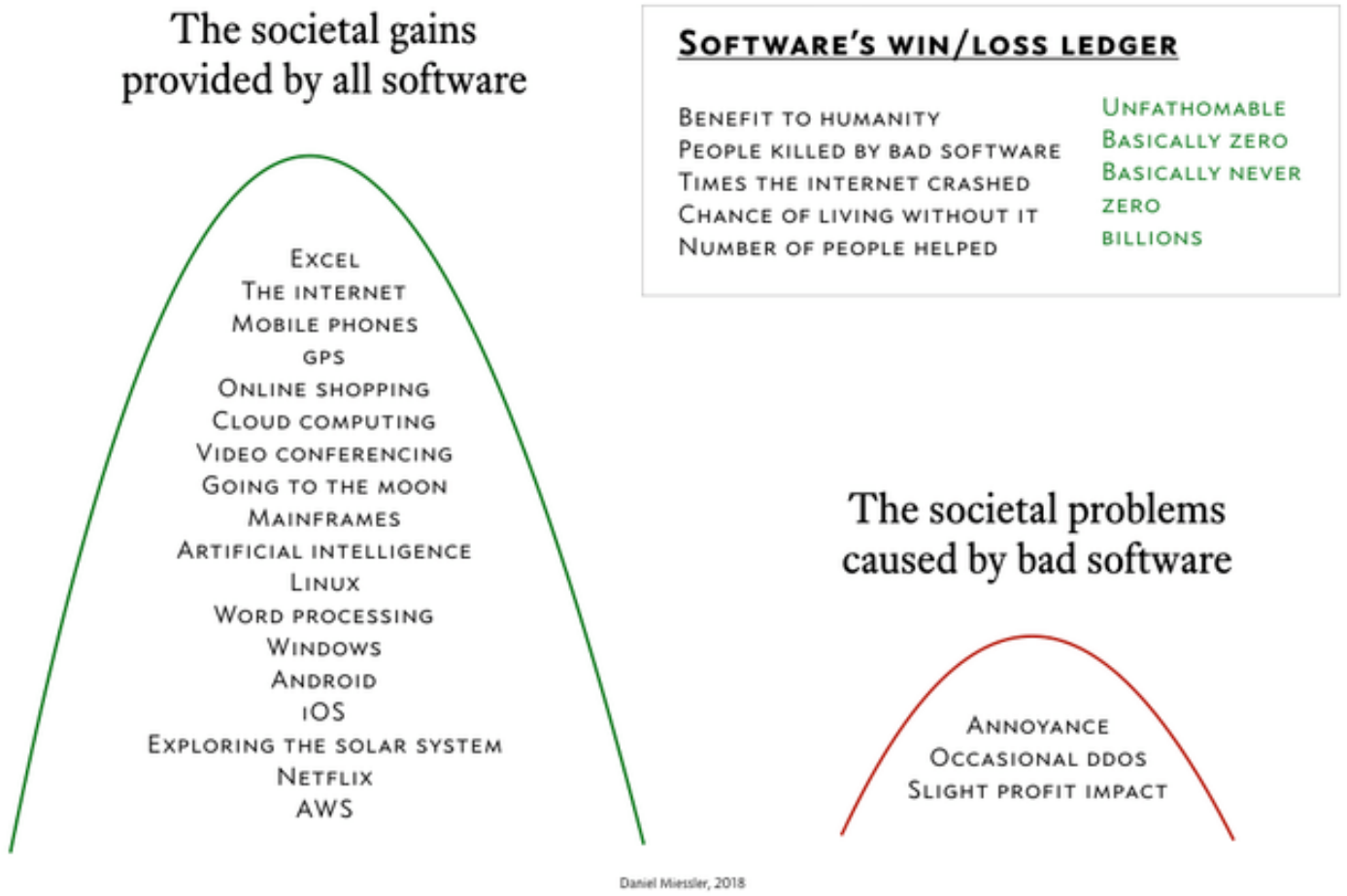
- Think about your audience
  - What do the people who will read this want to know?

Some end of year reflection

-

not exam material

Why software remains insecure

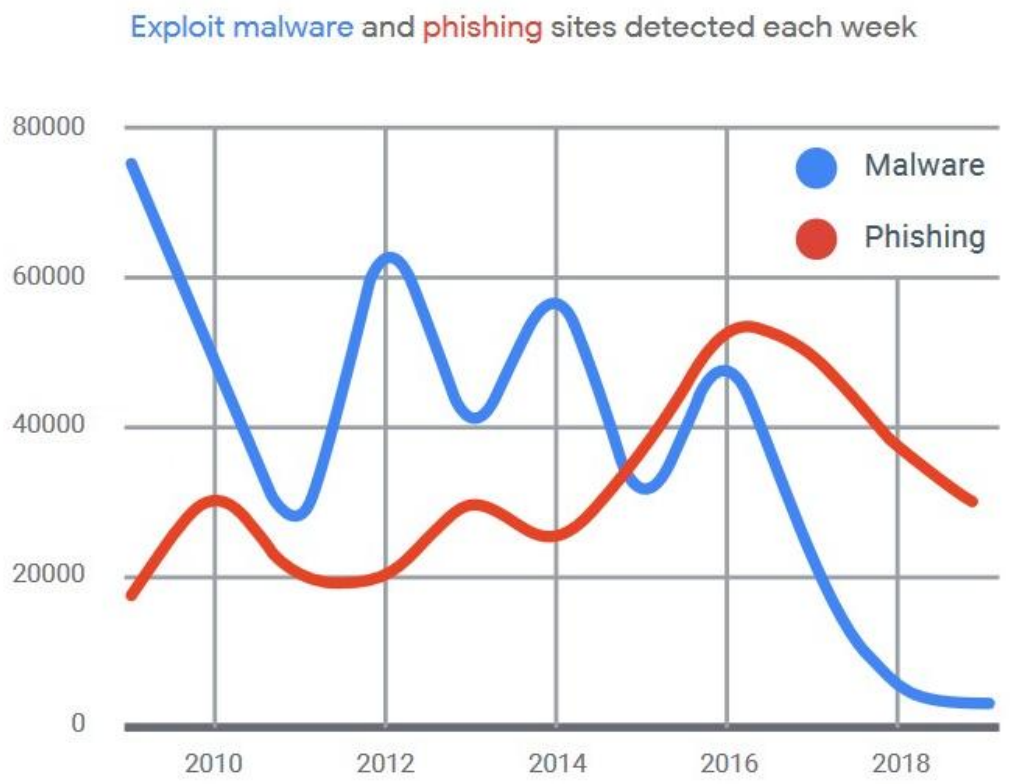


Daniel Miessler,

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

Maybe software is secure enough?

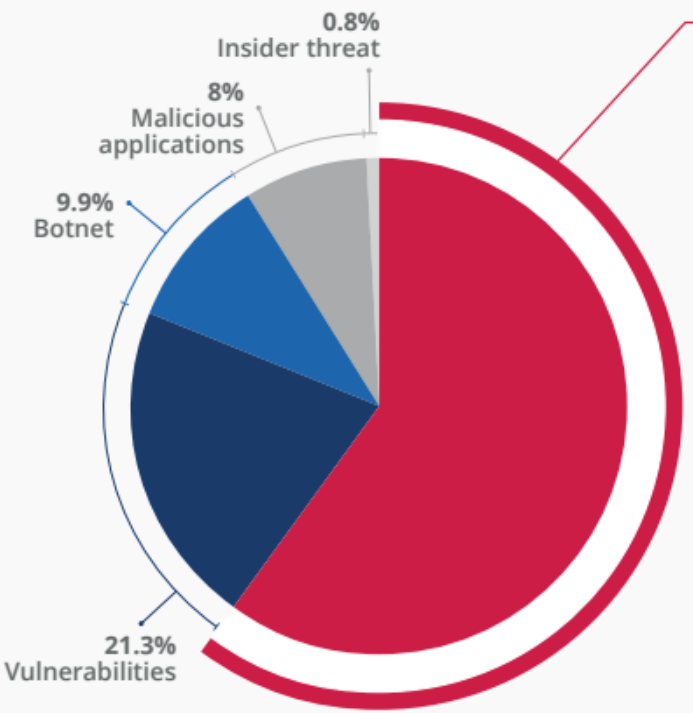
Phishing overtook exploit-based malware in 2016



Source: Safe Browsing (Google Transparency Report)

ENISA 2025 Threat Landscape

Tactics, Techniques and Procedures overview



Most identified initial infection vector

Source: ENISA dataset

60%
Phishing

Phishing was the dominant intrusion vector, accounting for approx. 60% of cases, including malspam, vishing, and malvertising. Vulnerability exploitation represented 21.3% of initial access vectors, with 68% leading to malware deployment as a follow-up activity.

Mobile devices and Internet- exposed services and devices, particularly Operational Technology (OT) systems remain high value targets across all types of threats.

State-aligned intrusion sets and cybercriminal operators increasingly leverage AI for productivity and optimisation of their malicious activities.

https://www.enisa.europa.eu/publications/enisa-threat-landscape-2025