LLVM Fuzzing Audit

Fuzzing Audit Report

Adam Korczynski, David Korczynski

2024-01-11

About Ada Logics

Ada Logics is a software security company founded in Oxford, UK, 2018 and is now based in London. We are a team of pragmatic security engineers and security researchers that work hands-on with code auditing, security automation and security tool development.

We are committed open source contributors and we routinely contribute to state of the art security tooling in the fuzzing domain such as advanced fuzzing tools like Fuzz Introspector and continuous fuzzing with OSS-Fuzz. For example, we have contributed to fuzzing of hundreds of open source projects by way of OSS-Fuzz. We regularly perform security audits of open source software and make our reports publicly available with findings and fixes, and we have audited many of the most widely used cloud native applications.

Ada Logics contributes to solving the challenge of securing the software supply-chain. To this end, we develop the tooling and infrastructure needed for ensuring a secure software development lifecycle, and we deploy these tools to critical software packages. On the tooling and infrastructure side, we contribute to projects such as the OpenSSF Scorecard project as well as the Sigstore projects like SLSA and Cosign.

Ada Logics helps some of the most exposed organisations secure their software, analyse their code and increase security automation and assurance, and if you would like to consider working with us please reach out to us via our website.

We write about our work on our blog and maintain a Youtube channel with educational videos. You can also follow Ada Logics on Linkedin, X.

Ada Logics ltd
71-75 Shelton Street,
WC2H 9JQ London,
United Kingdom

Contents

1 Project dashboard

Contact	Role	Organisation	Email
Adam Korczynski	Auditor	Ada Logics Ltd	adam@adalogics.com
David Korczynski	Auditor	Ada Logics Ltd	david@adalogics.com
Amir Montazery	Facilitator	OSTIF	amir@ostif.org
Derek Zimmer	Facilitator	OSTIF	derek@ostif.org
Helen Woeste	Facilitator	OSTIF	helen@ostif.org

2 Executive Summary

Ada Logics conducted a fuzzing audit of LLVM at the end of November and December 2023. The goal of the audit was to generally improve the fuzzing set up of LLVM with a particular focus on its continuous fuzzing by way of OSS-Fuzz. The audit was facilitated by the Open Source Technology Improvement Fund (OSTIF) and funded by the Sovereign Tech Fund.

Ada Logics has extensive experience in fuzzing and throughout the initial assessment of LLVM's fuzzing set up we identified and prioritised the tasks needed to have impact on the fuzzing of LLVM. To this end, throughout the engagement we fixed the existing OSS-Fuzz LLVM fuzzing set up, extended existing fuzzers as well as added new fuzzers, patched issues found by fuzzers as well and developed a strategy on how to move the LLVM fuzzing set up forward.

The LLVM project has extensive fuzzing, however, it lacks efficiency in certain areas that means the existing set up does not reach its full potential in terms of memory corruption issues. In order to improve the chance of the fuzzers finding memory corruption issues in LLVM we recommend addressing efficiency issues in the fuzzing set up, and estimate once this has been done a significant amount of the LLVM codebase will be covered by fuzzing.

In summary, during the engagement we:

- Fixed the OSS-Fuzz set up of LLVM that had been broken for more than a year.
- Expanded coverage from 1.1 million to 2.4 million LoC, making it the project on OSS-Fuzz with most lines of code covered by fuzzing.
- Extended existing fuzzing suite on OSS-Fuzz and developed 3 new fuzzers, increasing the fuzzers on OSS-Fuzz with 15 fuzzers.
- Fixed 11 issues reported by OSS-Fuzz, including 8 memory corruption issues.
- Developed strategy for next steps of fuzzing LLVM, with a focus on improving fuzzing efficiency

3 LLVM Fuzzing Audit

3.1 Engagement overview

The goal of this engagement was to improve the fuzzing set up of LLVM and to this end we performed several different tasks all aimed at improving the LLVM fuzzing set up. In this section we detail the various high-level tasks performed and the results of them. We summarise the engagement in the following tasks:

1. LLVM OSS-Fuzz setup analysis and repair
2. Fixing issues reported by OSS-Fuzz
3. Expanding fuzzing coverage
4. Identifying areas for improvement and future work

In the following sections we go through each of these tasks.

3.2 LLVM OSS-Fuzz setup and repair

LLVM has been integrated into OSS-Fuzz since August 2017. At this point in time there were around 90 projects in OSS-Fuzz (in contrast to more than 1200 now), which makes it one of the projects that has been in OSS-Fuzz for the longest period of time.

In total, OSS-Fuzz has reported more than 2770 issues in LLVM and there are around 400 open issues at the moment. The LLVM OSS-Fuzz project is public by having no view restrictions which means that anyone can (1) view the issues reported by the OSS-Fuzz setup, and (2) download the reproducer test cases to reproduce any of the reported findings. As such, anyone can monitor and reproduce the issues discovered without any limitations on deadlines, i.e. issues are made public when they are found and do not have any embargo on them.

For example, the following steps reproduce the following issue llvm/clang-fuzzer: Null-dereference READ in clang::Lexer::Lex:

```
#!/bin/bash

mkdir workdir
cd workdir

# Download the "Reproducer Testcase" (https://oss-fuzz.com/download?
    testcase_id=5665748027965440)
# and storeit in ./clusterfuzz-testcase-minimized-clang-fuzzer
    -5665748027965440 (name of the file)


git clone https://github.com/google/oss-fuzz
cd oss-fuzz
python3 infra/helper.py build_fuzzers llvm
python3 infra/helper.py reproduce \
                        llvm \
                        clang-fuzzer \
                        ./../clusterfuzz-testcase-minimized-clang-
                            fuzzer-5665748027965440
```

The above assumes the issue has not been fixed and that the build is working, which is the case as of the release of this report.

LLVM Build status

LLVM is one of the projects in OSS-Fuzz that has been there for the longest time, however, the health of the LLVM OSS-Fuzz set up has not been ideal in recent years. Looking at the monorail bug tracker, we can find the following fuzzing-build issues for LLVM, which shows the project has been failing to build throughout:

- Dec 15 2019 : Dec 19 2019

- Aug 19, 2020 : Aug 21, 2020
- Nov 6, 2020 : Nov 7, 2020
- Nov 20, 2020 : Jan 25, 2022
- May 13, 2022 : Aug 13, 2022
- Oct 7, 2022 : Dec 2, 2023

In this sense there had been long failing builds for LLVM between the periods Nov 20, 2020 to late 2023, and when we started the engagement the project had been failing to build for more than a year. As the build was broken, LLVM had not been fuzzing the latest up-to-date code, and had not generated any code coverage reports as well.

The build issue was, however, that one of the issues triggered an issue in the first run of the fuzzer, and OSS-Fuzz then considers the build broken since the fuzzer will not do any form of exploration. This was initially fixed by removing the fuzzer from the OSS-Fuzz build while simultaneously submitting a fix for the fuzzer 4.10.

Getting coverage working

The LLVM coverage build failed to pass in the OSS-Fuzz infrastructure even after fixing the fuzzing build. The difference in this case is that the "fuzzing" build refers to building and running the fuzzers using bug-finding sanitizers (e.g. ASAN) whereas the coverage build refers to building LLVM with lcov and generating html reports showing the code coverage of the source of LLVM.

The main problem is that coverage builds take up more memory when building the fuzzers, and this was exhausting the resources on the OSS-Fuzz cloud machines causing the build to be aborted.

To solve this issue the first step was to reduce the amount of parallelism during the LLVM build process for coverage builds. However, even when no parallelism is used (i.e. compiling with a single job), the memory would be exhausted. The issue is that when building certain files in the LLVM codebase, the build will simply exhaust the memory available. To overcome this, we added a minor tool for the LLVM build that patches the build set up of LLVM for two files to *not* include coverage instrumentation.

The following PRs on OSS-Fuzz are focused on getting the code coverage working again:

- llvm: limit resources for build
- llvm: fix coverage build
- llvm: limit coverage builds to 2 processes
- llvm: reorder fuzzer builds
- llvm: try getting coverage to work
- llvm: fix coverage build
- llvm: fix coverage build

3.3 Fixing issues reported by OSS-Fuzz.

Following the initial analysis and the build fixing, the next step was to start fixing the issues reported by OSS-Fuzz. The most important in this context are the issues labelled as security relevant, and at the beginning of the engagement there were several open issues reported by OSS-Fuzz and labelled "Security-issue". For reference, in general issues are labelled security issues by OSS-Fuzz if they are memory corruption issues, such as buffer overflows, use-after-frees and alike. Throughout the report we will refer to these as security issues for this reason, although the specific security relevance is dependent on the individual LLVM component's threat model. This engagement focused on fuzzing and we consider it out of scope for this audit to develop such threat models. The list of open such issues can be found using the following query on Monorail: open security issues on LLVM OSS-Fuzz's monorail. In addition to the security-relevant issues there are several open issues for e.g. memory leaks and NULL-pointer dereferences. We also consider these important to fix.

The issues themselves vary in nature in terms of complexity, furthermore, some of these issues are not triggered in a single iteration of a fuzzer, but need 2 iterations. We considered issues that were triggerable in a single iteration as the most important because these correspond more to the use case of LLVM/Clang where the operations the fuzzers perform are usually performed in an ethemeral manner, e.g. you use an individual process of clang for each run of the compiler.

The issues vary significantly in complexity, and for some of the issues it can be tricky to understand the root-cause as well as the fix. Ideally, fixing the issues should be delegated to those who know the code, although this is logistically difficult in LLVM's case since many of the maintainers are not familiar with the OSS-Fuzz set up.

The issues that we proposed fixes for are in the list of issues below, and in this category of issue type the following are relevant:

- Heap-use-after-free in clang::Parser::isCXXDeclarationSpecifier
- Heap-use-after-free in clang::Sema::GetNameFromUnqualifiedId
- Heap-buffer-overflow in llvm::object::WasmObjectFile::parseCodeSection
- Out of bounds write in llvm::DWARFUnitIndex::paseImpl
- Null-dereference READ in llvm::object::WasmObjectFile::parseLinkingSectionSymtab
- Global-buffer-overflow in llvm::hashing::detail::hash_short
- Heap-buffer-overflow in llvm_regcomp
- Heap-buffer-overflow in WasmObjectFile::parseLinkingSectionSymtab
- NULL-dereference READ in processTypeAttrs
- NULL-dereference READ in GetFullTypeForDeclarator

In addition to the security-labelled issues, OSS-Fuzz has also reported more than 180 issues that are related to false `asserts` (list here). These are less relevant from a code security perspective and more relevant from a fuzzing-health perspective, since these issues create a significant hurdle for the fuzzing of LLVM. We will discuss more about this in later sections.

Several of these issues were present prior to the engagement, and some were discovered following the fixing of the build as well as new fuzzers occuring during the audit.

3.4 Expanding fuzzing coverage

The next step of the engagement that was relevant was expanding the fuzzing performed by OSS-Fuzz. The code coverage of LLVM had been broken for a while at the beginning of the engagement, and the most recent coverage report that we could find from before the engagement was from 10th May, 2022. At that point in time LLVM had around 100K LoC analysed. However, this is not a perfect example of correct code coverage since some fuzzers on OSS-Fuzz were disabled. For example, the report from two years earlier shows 1.1 million LoC covered by the fuzzers LLVM Code coverage report May 10th, 2020.

To expand the fuzzing coverage of LLVM we did two primary tasks:

1. Expand on existing fuzzers to cover additional code
2. Develop new fuzzers that target unexplored code
3. Fix issues/fuzz blockers that break fuzzers

1. Expand on existing fuzzers to cover additional code

There are two fuzzers in LLVM that are written in a way where they can easily be adjusted to cover certain parts of the code: llvm-isel-fuzzer and llvm-opt-fuzzer.

llvm-isel-fuzzer generates LLVM IR modules and will run the LLVM (legacy) pass manager on the modules and will also emit these modules. In order to emit the modules several steps need to be handled by LLVM, e.g. code generation steps. The idea behind this fuzzer is to emit files of various architectures in order to trigger code generation steps for the various architectures. To this end we extended the architectures that OSS-Fuzz would analyse with `hexagon`, `riscv64`, `mips64`, `arm`, `ppc64`, `nvptx`, `ve`, `bpf`. This is in addition to the existing architectures: `aarch64`, `x86_64`, `wasm32`, `aarch64-gisel`.

The llvm-opt-fuzzer is similar in nature to llvm-isel-fuzzer in that it relies on creating LLVM modules seeded with fuzz-data and run the LLVM processing on these modules. The llvm-opt-fuzzer, however, is not focused on code generation but rather on running the LLVM pass pipeline on the generated modules. To this end, the focus is to analyse various different LLVM passes and we extended with 6 new passes: `dse`, `loop_idiom`, `reassociate`, `lower_matrix_intrinsics`, `memcpyopt`, `sroa`. This is in addition to around 15 existing LLVM passes being analysed.

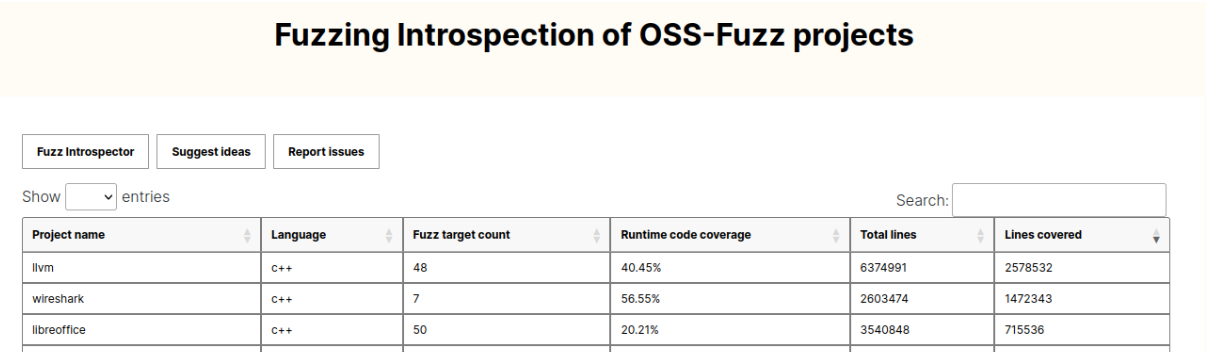
2. Develop new fuzzers that target unexplored code

Next, we developed a set of new fuzzers that target new parts of the LLVM codebase. In total, we added three new fuzzers:

- llvm-parse-assembly-fuzzer
- llvm-object-yaml-fuzzer

- llvm-symbol-reader-fuzzer

Following the fixing of the OSS-Fuzz set up the LLVM build and coverage build, the total lines of code coverage was slightly more than 1.1 million LoC. The extensions described in this section increased the lines oc code analysed to around 2.6 million lines of code, and, interestingly the LLVM is now the project with most lines of code covered on OSS-Fuzz as shown in Figure 1.


Fuzzing Introspection of OSS-Fuzz projects

Fuzz Introspector | Suggest ideas | Report issues

Show entries Search:

Project name	Language	Fuzz target count	Runtime code coverage	Total lines	Lines covered
llvm	c++	48	40.45%	6374991	2578532
wireshark	c++	7	56.55%	2603474	1472343
libreoffice	c++	50	20.21%	3540848	715536

Figure 1: Coverage overview of OSS-Fuzz projects, showing LLVM has highest amount of lines covered https://introspector.oss-fuzz.com/projects-overview

Additionally the fuzz count number on OSS-Fuzz increased from 30 to 48, and the correlation between fuzz count increasing and code coverage increasing is shown in Figure 2.

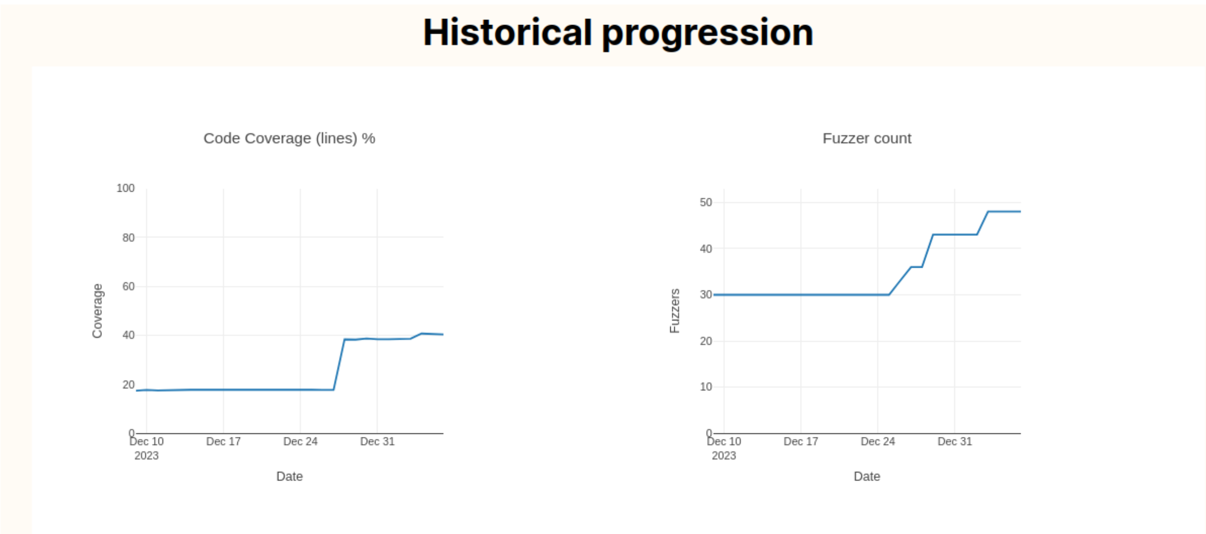


Figure 2: LLVM historical progression since build was fixed https://introspector.oss-fuzz.com/project-profile?project=llvm

3.5 Identifying areas of improvement and future work

As the final objective of our engagement we focused on identifying directions for where LLVM should focus on fuzzing efforts. There are several areas of improvement and tasks that can be done for future work, and we consider the three primary tasks to be:

1. Ensure that fuzzers are running correctly
2. Fix issues to ensure fuzzers run
3. Limit the use of `abort` and hard exits

There are other possible tasks, although we consider these secondary to the above listed ones. These include

- Expand with new fuzzers
- Ensure proper seeds for the fuzzers

We consider these secondary because the three first items are likely to cover a lot of the code in the LLVM codebase, but are currently blocked for progress. In total, at 9th December 2023 when the coverage build was fixed, the lines of code coverage by fuzzers was 1,111,412 and at the end of this engagement a total of 2.588.921.

It is very likely that once the first batch of issues are found then further blockers of the same kind will occur. As such, the primary issues listed above are likely time-consuming and long-term tasks.

Once the three issues listed below have been solved, we estimate that the LLVM fuzzing setup will (1) have found and discovered a fair number of new memory corruption issues and (2) that the fuzzing set up will cover a significant part of the LLVM codebase.

In the following we will go into more details with the three primary areas suggested above. The three areas are all related to each other, in that they revolve around the fuzzers running without being crashed by existing issues regularly. We have split this overall topic into three issues, by and large due to the possible solutions at hand.

1. Ensure that fuzzers are running correctly:

The fuzzers of LLVM are facing issues in terms of encountering code points that cause the fuzzers to be stopped. This makes the fuzzing inefficient, and currently our estimate is that the LLVM fuzzers have a significant potential in terms of exploring many more parts of the LLVM codebase, but are currently blocked from doing this by the early exits.

In this case, we would like to reference the OSS-Fuzz "Fuzzer Statistics" page, which is accessible to the emails listed in the LLVM project.yaml by way of oss-fuzz.com. This page shows various metrics for the performance of the fuzzers, and Figure 3 shows a screenshot as of early Jan, 2024 of the page with fuzzers sorted by the column "fuzzing_time_percent". This column shows the *"Percent of expected* *fuzzing time actually spent fuzzing"*. Several of the fuzzers have less than 1% efficiency and many of the fuzzers have less than 25% fuzzing time. Ideally, this should be closer to 100% from the perspective of ensuring fuzzers spend time exploring new code.

In general, our guidance in this next step is to focus on using the "Fuzzer Statistics" page to ensure fuzzers run efficiently, and in particular by way of the "fuzzing_time_percent" column.

fuzzer	tests_executed	new_crashes	edge_coverage	cov_report	corpus_size	avg_exec_per_sec	fuzzing_time_percent
libFuzzer_llvm_llvm-isel-fuzzer--bpf-O2	6,950	2	--	--	326 (26 MB)	104.4	0.1
libFuzzer_llvm_llvm-isel-fuzzer--mips64-O2	45,269	0	2.45% (42149/1720272)	Coverage	--	101.5	0.1
libFuzzer_llvm_llvm-dis-fuzzer	290,863	0	5.22% (5957/114182)	Coverage	--	520.7	0.8
libFuzzer_llvm_llvm-object-yaml-fuzzer	8,454	1	1.26% (2245/178688)	Coverage	--	29.4	0.8
libFuzzer_llvm_llvm-dwarfdump-fuzzer	312,002	2	--	--	16517 (182 MB)	124.5	0.9
libFuzzer_llvm_llvm-isel-fuzzer--nvptx-O2	12,803	2	3.54% (60885/1720272)	Coverage	--	28.5	1.2
libFuzzer_llvm_llvm-isel-fuzzer--arm-O2	93,415	0	5.03% (86570/1720272)	Coverage	--	17	7.4
libFuzzer_llvm_llvm-opt-fuzzer--x86_64-instcombine	414,316	1	5.10% (87236/1712095)	Coverage	--	33.9	9.3
libFuzzer_llvm_llvm-isel-fuzzer--aarch64-gisel	153,315	0	10.42% (179206/1720272)	Coverage	--	62.3	9.4
libFuzzer_llvm_llvm-isel-fuzzer--riscv64-O2	275,145	0	10.71% (184161/1720272)	Coverage	--	21.8	15.2
libFuzzer_llvm_llvm-isel-fuzzer--wasm32-O2	62,881	0	5.48% (94342/1720272)	Coverage	--	58.5	18.8
libFuzzer_llvm_llvm-parse-assembly-fuzzer	36,680,426	2	10.90% (13013/119352)	Coverage	--	603.1	20.8
libFuzzer_llvm_llvm-isel-fuzzer--ppc64-O2	229,198	0	5.89% (101350/1720272)	Coverage	--	26.6	21.7
libFuzzer_llvm_llvm-isel-fuzzer--hexagon-O2	147,185	0	5.63% (96809/1720272)	Coverage	9064 (37 MB)	30.1	21.8
libFuzzer_llvm_llvm-opt-fuzzer--x86_64-guard_widening	3,540,783	1	2.50% (42808/1712095)	Coverage	--	135.5	22
libFuzzer_llvm_llvm-isel-fuzzer--aarch64-O2	130,729	0	10.37% (178399/1720272)	Coverage	--	39.3	23.2
libFuzzer_llvm_llvm-isel-fuzzer--x86_64-O2	214,943	0	11.29% (194276/1720272)	Coverage	--	12.9	25

Figure 3: Fuzzing statistics for LLVM on OSS-Fuzz

The next to suggestions for future work are related to this task as well, in that the following two suggestions are pragmatic ways to improve the fuzzing efficiency, and likely those that will have most impact.

2. Fix issues to ensure fuzzers run

In general, the key way to ensure fuzzers run efficiently is ensuring there are no open issues on OSS-Fuzz. This means that the list of open issues should be 0, and currently it has more than 380. However, we suggest prioritising the issues in the following order:

1. Fix the issues that are labelled security-relevant: list.
2. Fix the NULL-derefence issues: list.
3. Fix the issues that are related to failed `asserts`: list.
4. Fix issues related to leaks and OOMs
5. Fix the remainder issues.

The above list is a rough-guideline and not a hard prioritisation based on which code issues are likely most relevant to the security of LLVM. Another important metric for prioritising which issues to fix is how fast the fuzzers run into the given issue. For example the fuzzers with less than 1 percent fuzzing efficiency are running into specific issues instantly in the execution, and fixing these should be prioritised as well.

3. Limit the use of abort and hard exits

The group of issues with the biggest number of issues is the failed `asserts`. Failed `asserts` are used across LLVM to catch error states and a failed assert does not mean that a bug exists in the LLVM code. This use of `asserts` makes it difficult for the fuzzers to explore code, as the fuzzers will consistently run into failed asserts during execution.

In the `llvm/lib/` folder, there are more than 1100 calls to `report_fatal_error` which causes hard exits once a fuzzer triggers a call to this function:

```
$ git clone https://github.com/llvm/llvm-project --depth=1
$ cd llvm-project/llvm/lib
$ grep -rn "report_fatal_error" ./ | wc -l
1146
```

A general recommendation to maximise fuzzing efficiency is to limit the use of fatal errors. LLVM already has extensive use of passing non-fatal errors, which can be handled by the calling code. From a fuzzing perspective, soft errors that can be caught or handled by the fuzzers will maximise the efficiency of the fuzzers and, consequently, optimize the chance of finding security vulnerabilities.

An example of this is the `llvm-dwarfdump-fuzzer` which exercises a lot of code in the `llvm/lib/Object/WasmObjectFile.cpp` module. This module, however, uses various functions that reads numerical values from data provided by the fuzzer, and if the numerical value does not match certain criteria a hard exit is performed. Some of these criteria are difficult for the fuzzer to get right when it's aborted all the time. For example `readVaruint1` reads a numerical value from the fuzz data, and unless the numerical value is above 1 or below 0, a fatal exit will happen:

```
static uint8_t readVaruint1(WasmObjectFile::ReadContext &Ctx) {
  int64_t Result = readLEB128(Ctx);
  if (Result > VARUINT1_MAX || Result < 0)
    report_fatal_error("LEB is outside Varuint1 range");
  return Result;
}
```

Another example from the llvm-dwarfdump-fuzzer is issue20708. This issue was discovered on 15th Febuary 2020, and on June 18th, 2020 OSS-Fuzz added a note that the crash occurs frequently, limiting the potential progress of the fuzzer. The issue shows a stacktrace with a call into `readLimits` causes an `abort`:

```
static wasm::WasmLimits readLimits(WasmObjectFile::ReadContext &Ctx) {
```

```
  wasm::WasmLimits Result;
  Result.Flags = readVaruint32(Ctx);
  Result.Minimum = readVaruint64(Ctx);
  if (Result.Flags & wasm::WASM_LIMITS_FLAG_HAS_MAX)
    Result.Maximum = readVaruint64(Ctx);
  return Result;
}
```

This function uses two utility functions for reading numerical values out of data provided by the fuzzer: `readVaruint32` and `readVaruint64`. `readVaruint32` is defined as follows:

```
static uint32_t readVaruint32(WasmObjectFile::ReadContext &Ctx) {
  uint64_t Result = readULEB128(Ctx);
  if (Result > UINT32_MAX)
    report_fatal_error("LEB is outside Varuint32 range");
  return Result;
}
```

Furthermore, there `readULEB128` is defined as follows:

```
113 static uint64_t readULEB128(WasmObjectFile::ReadContext &Ctx) {
114   unsigned Count;
115   const char *Error = nullptr;
116   uint64_t Result = decodeULEB128(Ctx.Ptr, &Count, Ctx.End, &Error);
117   if (Error)
118     report_fatal_error(Error);
119   Ctx.Ptr += Count;
120   return Result;
121 }
```

The problem is that both `readVaruint32` and `decodeULEB128` has a chance of calling `report_fatal_error` in the event the integer read from the fuzzer-provided data is not within a certain range or corresponds to a certain format.

It is very likely that the fuzzer will not produce accurate numerical values in the majority of fuzz iterations, and causing a crash here significantly blocks the fuzzer from doing further analysis as the fuzzer relies on in-process fuzzing.

Instead of aborting with a fatal issue, it would be much better for the fuzzing if the error on line 116 is propagated further up the stack so it can be softly handled and the fuzzer can continue running without the process being crashed. However, in this case, the problem is that a refactoring of this requires significant adjustments as there are e.g. more than 70 use cases of `readVaruint32` and there are several other similar uses across `WasmObjectFile.cpp` that are non-trivial to adjust. In this sense, the right approach would be to refactor the `WasmObjectFile.cpp` so that fatal errors are not used, and in particular in places where some input data does not correspond to some expected structure.

4 Issues found and fixed

In this section we will go through the issues found and fixed throughout the audit.

4.1 Heap-buffer-overflow in llvm::xxh3_64bits

id	ADA-2023-LLVM-1
Monorail ID and URL	65114
Date reported by OSS-Fuzz	2023-12-16
Fix PR	[llvm-dwarfdump-fuzzer] fix out of bounds potential

A heap overflow was reported to exist within `llvm::xxh3_64bits`. However, after fixing the `llvm-dwarfdump-fuzzer` by ensuring the input data is properly wrapped this issue is fixed.

The original fuzzer is as follows:

```
extern "C" int LLVMFuzzerTestOneInput(uint8_t *data, size_t size) {
  std::unique_ptr<MemoryBuffer> Buff = MemoryBuffer::getMemBuffer(
      StringRef((const char *)data, size), "", false);
```

The fixed fuzzer is as follows:

```
extern "C" int LLVMFuzzerTestOneInput(uint8_t *data, size_t size) {
  std::string Payload(reinterpret_cast<const char *>(data), size);
  std::unique_ptr<llvm::MemoryBuffer> Buff = llvm::MemoryBuffer::
     getMemBuffer(Payload);
```

The problem is that the current fuzzer relies on MemoryBuffer to hold the fuzz data. However, the fuzzer runs into an OOB instantly because the MemoryBuffer interface guarantees that "In addition to basic access to the characters in the file, this interface guarantees you can read one character past the end of the file, and that this character will read as '\0'.", which the fuzzer fails to satisfy.

4.2 Out of bounds write in llvm::DWARFUnitIndex::paseImpl

id	ADA-2023-LLVM-2
Monorail ID and URL	30308
Date reported by OSS-Fuzz	2021-02-05
Fix PR	[DWARFLibrary] Add bounds check to Contrib index

An out of bounds write exists in the `llvm::DWARFUnitIndex::parseImpl` at the following lines:

```
146   auto Contribs =
147       std::make_unique<Entry::SectionContribution *[]>(Header.NumUnits);
148   ColumnKinds = std::make_unique<DWARFSectionKind[]>(Header.NumColumns);
149   RawSectionIds = std::make_unique<uint32_t[]>(Header.NumColumns);
150
151   // Read Hash Table of Signatures
152   for (unsigned i = 0; i != Header.NumBuckets; ++i)
153     Rows[i].Signature = IndexData.getU64(&Offset);
154
155   // Read Parallel Table of Indexes
156   for (unsigned i = 0; i != Header.NumBuckets; ++i) {
157     auto Index = IndexData.getU32(&Offset);
158     if (!Index)
159       continue;
160     Rows[i].Index = this;
161     Rows[i].Contributions =
162         std::make_unique<Entry::SectionContribution[]>(Header.NumColumns);
163     Contribs[Index - 1] = Rows[i].Contributions.get();
```

The problem is that the write on line 163 depends on `Index`, which is read on line 157 from arbitrary data, and there is no bounds checking on the value.

The proposed fix is to add bounds checking when reading `Index`:

```
    auto Index = IndexData.getU32(&Offset);
    if (!Index)
      continue;
    // Fix: ensure proper bounds
    if (Index > Header.NumColumns)
```

```
      return false;
    Rows[i].Index = this;
    Rows[i].Contributions =
        std::make_unique<Entry::SectionContribution[]>(Header.
            NumColumns);
    Contribs[Index - 1] = Rows[i].Contributions.get();
```

4.3 Heap-buffer-overflow in llvm::object::WasmObjectFile::parseCodeSection

id	ADA-2023-LLVM-3
Monorail ID and URL	28856
Date reported by OSS-Fuzz	2020-12-21
Fix PR	[WebAssembly] Add bounds check in parseCode-Section

An overflow was reported to exist in `decodeULEB128` with the following stacktrace:

```
    =================================================================
==6507==ERROR: AddressSanitizer: heap-buffer-overflow on address 0
   x6070000000fc at pc 0x0000009061c4 bp 0x7fff87432890 sp 0
   x7fff87432888
READ of size 1 at 0x6070000000fc thread T0
    #0 0x9061c3 in decodeULEB128 llvm-project/llvm/include/llvm/Support
       /LEB128.h:144:22
    #1 0x9061c3 in readULEB128 llvm-project/llvm/lib/Object/
       WasmObjectFile.cpp:116:21
    #2 0x9061c3 in readVaruint32 llvm-project/llvm/lib/Object/
       WasmObjectFile.cpp:158:21
    #3 0x9061c3 in llvm::object::WasmObjectFile::parseCodeSection(llvm
       ::object::WasmObjectFile::ReadContext&) llvm-project/llvm/lib/
       Object/WasmObjectFile.cpp:1469:21
    #4 0x8f85c1 in llvm::object::WasmObjectFile::parseSection(llvm::
       object::WasmSection&) llvm-project/llvm/lib/Object/
       WasmObjectFile.cpp:376:12
    #5 0x8f767d in llvm::object::WasmObjectFile::WasmObjectFile(llvm::
       MemoryBufferRef, llvm::Error&) llvm-project/llvm/lib/Object/
       WasmObjectFile.cpp:340:16
    #6 0x8f5c01 in make_unique<llvm::object::WasmObjectFile, llvm::
       MemoryBufferRef &, llvm::Error &> /usr/local/include/c++/v1/
       __memory/unique_ptr.h:724:32
    #7 0x8f5c01 in llvm::object::ObjectFile::createWasmObjectFile(llvm
       ::MemoryBufferRef) llvm-project/llvm/lib/Object/WasmObjectFile.
       cpp:69:21
    #8 0x8d47c6 in llvm::object::ObjectFile::createObjectFile(llvm::
       MemoryBufferRef, llvm::file_magic, bool) llvm-project/llvm/lib/
       Object/ObjectFile.cpp:195:12
    #9 0x5775aa in createObjectFile llvm-project/llvm/include/llvm/
       Object/ObjectFile.h:375:12
    #10 0x5775aa in LLVMFuzzerTestOneInput llvm-project/llvm/tools/llvm
       -dwarfdump/fuzzer/llvm-dwarfdump-fuzzer.cpp:27:7
```

After further analysis, the error was deemed to exist higher in the stacktrace, specificcally inside of `llvm::object::WasmObjectFile::parseCodeSection`:

```
1458 Error WasmObjectFile::parseCodeSection(ReadContext &Ctx) {
1459   CodeSection = Sections.size();
1460   uint32_t FunctionCount = readVaruint32(Ctx);
1461   if (FunctionCount != Functions.size()) {
1462     return make_error<GenericBinaryError>("invalid function count",
1463                                           object_error::parse_failed);
1464   }
1465
1466   for (uint32_t i = 0; i < FunctionCount; i++) {
1467     wasm::WasmFunction& Function = Functions[i];
1468     const uint8_t *FunctionStart = Ctx.Ptr;
1469     uint32_t Size = readVaruint32(Ctx);
1470     const uint8_t *FunctionEnd = Ctx.Ptr + Size;
1471
1472     Function.CodeOffset = Ctx.Ptr - FunctionStart;
1473     Function.Index = NumImportedFunctions + i;
1474     Function.CodeSectionOffset = FunctionStart - Ctx.Start;
1475     Function.Size = FunctionEnd - FunctionStart;
1476
1477     uint32_t NumLocalDecls = readVaruint32(Ctx);
1478     Function.Locals.reserve(NumLocalDecls);
1479     while (NumLocalDecls--) {
1480       wasm::WasmLocalDecl Decl;
1481       Decl.Count = readVaruint32(Ctx);
1482       Decl.Type = readUint8(Ctx);
1483       Function.Locals.push_back(Decl);
1484     }
1485
1486     uint32_t BodySize = FunctionEnd - Ctx.Ptr;
1487     Function.Body = ArrayRef<uint8_t>(Ctx.Ptr, BodySize);
1488     // This will be set later when reading in the linking metadata
          section.
1489     Function.Comdat = UINT32_MAX;
1490     Ctx.Ptr += BodySize;
1491     assert(Ctx.Ptr == FunctionEnd);
```

The problem is that `Size` read on line 1469 is read from data and denotes the size of `Function` inside of the memory owned by `Ctx`. However, there is no checking on whether the `Size` (of the function) extends beyond buffer owned by `Ctx`. Adding a check on the size fixes the issue:

```
1469     uint32_t Size = readVaruint32(Ctx);
1470     const uint8_t *FunctionEnd = Ctx.Ptr + Size;
1471
1472     Function.CodeOffset = Ctx.Ptr - FunctionStart;
1473     Function.Index = NumImportedFunctions + i;
1474     Function.CodeSectionOffset = FunctionStart - Ctx.Start;
1475     Function.Size = FunctionEnd - FunctionStart;
```

```

    uint32_t NumLocalDecls = readVaruint32(Ctx);
    Function.Locals.reserve(NumLocalDecls);
    while (NumLocalDecls--) {
      wasm::WasmLocalDecl Decl;
      Decl.Count = readVaruint32(Ctx);
      Decl.Type = readUint8(Ctx);
      Function.Locals.push_back(Decl);
    }

    uint32_t BodySize = FunctionEnd - Ctx.Ptr;
    Function.Body = ArrayRef<uint8_t>(Ctx.Ptr, BodySize);
    // This will be set later when reading in the linking metadata section.
    Function.Comdat = UINT32_MAX;

    // Fix: Check that Function start + size is within Ctx's buffer bounds.
    if (Ctx.Ptr + BodySize > Ctx.End) {
        return make_error<GenericBinaryError>("Function points beyond buffer",
                                              object_error::parse_failed);
    }
    Ctx.Ptr += BodySize;
    assert(Ctx.Ptr == FunctionEnd);
```

4.4 Null-dereference READ in llvm::object::WasmObjectFile::parseLinkingSectionSymtab

id	ADA-2023-LLVM-4
Monorail ID and URL	30789
Date reported by OSS-Fuzz	2021-02-01
Fix PR	[WasmObjectFile] fix NULL-dereference

A NULL-dereference was found with the following stack trace:

```
==24837==ERROR: AddressSanitizer: SEGV on unknown address 0
    x000000000558 (pc 0x000000550ae0 bp 0x7ffc829e7af0 sp 0x7ffc829e72b0
     T0)
==24837==The signal is caused by a READ memory access.
==24837==Hint: address points to the zero page.
SCARINESS: 10 (null-deref)
    #0 0x550ae0 in __sanitizer::internal_memmove(void*, void const*,
       unsigned long) llvm-project/compiler-rt/lib/sanitizer_common/
       sanitizer_libc.cpp:68:16
    #1 0x5397b5 in __asan_memmove llvm-project/compiler-rt/lib/asan/
       asan_interceptors_memintrinsics.cpp:30:3
    #2 0x9145dc in llvm::object::WasmObjectFile::
       parseLinkingSectionSymtab(llvm::object::WasmObjectFile::
       ReadContext&) llvm-project/llvm/lib/Object/WasmObjectFile.cpp
       :758:17
    #3 0x90f042 in llvm::object::WasmObjectFile::parseLinkingSection(
       llvm::object::WasmObjectFile::ReadContext&) llvm-project/llvm/
       lib/Object/WasmObjectFile.cpp:552:23
    #4 0x8f90fe in llvm::object::WasmObjectFile::parseCustomSection(
       llvm::object::WasmSection&, llvm::object::WasmObjectFile::
       ReadContext&) llvm-project/llvm/lib/Object/WasmObjectFile.cpp
       :1091:21
    #5 0x8f861c in llvm::object::WasmObjectFile::parseSection(llvm::
       object::WasmSection&) llvm-project/llvm/lib/Object/
       WasmObjectFile.cpp:354:12
```

The root-cause was determined to be in `llvm::object::WasmObjectFile::parseLinkingSectionSymta` at the following lines:

```
755       Info.ElementIndex = readVaruint32(Ctx);
756       // Use somewhat unique section name as symbol name.
757       StringRef SectionName = Sections[Info.ElementIndex].Name;
758       Info.Name = SectionName;
```

```
        break;
      }
```

The problem is that `Info.ElementIndex` is read from untrusted data and is then used as an index into the array. There is no bounds checking as to whether it's a valid index.

The proposed fix:

```
        Info.ElementIndex = readVaruint32(Ctx);
        if (Info.ElementIndex >= Sections.size()) {
          return make_error<GenericBinaryError>("invalid section index index",
                                                object_error::parse_failed);
        }
        // Use somewhat unique section name as symbol name.
        StringRef SectionName = Sections[Info.ElementIndex].Name;
        Info.Name = SectionName;
        break;
      }
```

4.5 Heap-use-after-free in clang::Parser::isCXXDeclarationSpecifier

id	ADA-2023-LLVM-5
Monorail ID and URL	23204
Date reported by OSS-Fuzz	2020-06-08
Fix PR	[clang][parse] Fix UAF in MaybeDestroyTemplates

Heap-use-after-free was discovered with the following stack trace:

```
==41917==ERROR: AddressSanitizer: heap-use-after-free on address 0
    x60600000b380 at pc 0x0000055596fe bp 0x7ffe882edcb0 sp 0
    x7ffe882edca8
READ of size 4 at 0x60600000b380 thread T0
    #0 0x55596fd in hasInvalidName llvm-project/clang/include/clang/
        Sema/ParsedTemplate.h:230:42
    #1 0x55596fd in clang::Parser::isCXXDeclarationSpecifier(clang::
        ImplicitTypenameContext, clang::Parser::TPResult, bool*) llvm-
        project/clang/lib/Parse/ParseTentative.cpp:1592:22
    #2 0x555659f in clang::Parser::isCXXSimpleDeclaration(bool) llvm-
        project/clang/lib/Parse/ParseTentative.cpp:162:18
    #3 0x5555dbc in clang::Parser::isCXXDeclarationStatement(bool) llvm
        -project/clang/lib/Parse/ParseTentative.cpp:112:12
    #4 0x54dd954 in isDeclarationStatement llvm-project/clang/include/
        clang/Parse/Parser.h:2497:14
    #5 0x54dd954 in clang::Parser::
        ParseStatementOrDeclarationAfterAttributes(llvm::SmallVector<
        clang::Stmt*, 32u>&, clang::Parser::ParsedStmtContext, clang::
        SourceLocation*, clang::ParsedAttributes&, clang::
        ParsedAttributes&) llvm-project/clang/lib/Parse/ParseStmt.cpp
        :239:10
```

where the memory was freed by at:

```
0x60600000b380 is located 32 bytes inside of 56-byte region [0
    x60600000b360,0x60600000b398)
freed by thread T0 here:
    #0 0x5d9472 in __interceptor_free llvm-project/compiler-rt/lib/asan
        /asan_malloc_linux.cpp:52:3
    #1 0x511126d in Destroy llvm-project/clang/include/clang/Sema/
        ParsedTemplate.h:219:7
    #2 0x511126d in clang::Parser::DestroyTemplateIds() llvm-project/
        clang/lib/Parse/Parser.cpp:581:9
```

```
    #3 0x54dbdfd in MaybeDestroyTemplateIds llvm-project/clang/include/
       clang/Parse/Parser.h:296:7
    #4 0x54dbdfd in clang::Parser::ParseStatementOrDeclaration(llvm::
       SmallVector<clang::Stmt*, 32u>&, clang::Parser::
       ParsedStmtContext, clang::SourceLocation*) llvm-project/clang/
       lib/Parse/ParseStmt.cpp:120:3
```

The root-cause was found to be that `clang::Parser::MaybeDestroyTemplateIds` is too permissive with the following code:

```
void MaybeDestroyTemplateIds() {
  if (!TemplateIds.empty() &&
      (Tok.is(tok::eof) || !PP.mightHavePendingAnnotationTokens()))
    DestroyTemplateIds();
}
```

Specifically, the issue found was discovered to trigger a condition where `Tok.is(tok::eof)` is true by `!PP.mightHavePendingAnnotationTokens()` is false.

The fix is to adjust `clang::Parser::MaybeDestroyTemplateIds` to narrow the check to:

```
void MaybeDestroyTemplateIds() {
  if (!TemplateIds.empty() &&
      (!PP.mightHavePendingAnnotationTokens()))
    DestroyTemplateIds();
}
```

4.6 Heap-use-after-free in clang::Sema::GetNameFromUnqualifiedId

id	ADA-2023-LLVM-6
Monorail ID and URL	52018
Date reported by OSS-Fuzz	2022-10-01
Fix PR	[clang][parse] Fix UAF in MaybeDestroyTemplates

Heap-use-after-free was discovered with the following stack trace:

```
==7843==ERROR: AddressSanitizer: heap-use-after-free on address 0
    x60600000b498 at pc 0x0000062eb0fe bp 0x7ffefb4c9f90 sp 0
    x7ffefb4c9f88
READ of size 8 at 0x60600000b498 thread T0
SCARINESS: 51 (8-byte-read-heap-use-after-free)
    #0 0x62eb0fd in get llvm-project/clang/include/clang/Sema/Ownership
       .h:81:41
    #1 0x62eb0fd in clang::Sema::GetNameFromUnqualifiedId(clang::
       UnqualifiedId const&) llvm-project/clang/lib/Sema/SemaDecl.cpp
       :6049:52
    #2 0x62ebdde in GetNameForDeclarator llvm-project/clang/lib/Sema/
       SemaDecl.cpp:5937:10
    #3 0x62ebdde in clang::Sema::HandleDeclarator(clang::Scope*, clang
       ::Declarator&, llvm::MutableArrayRef<clang::
       TemplateParameterList*>) llvm-project/clang/lib/Sema/SemaDecl.
       cpp:6360:34
    #4 0x62eb816 in clang::Sema::ActOnDeclarator(clang::Scope*, clang::
       Declarator&) llvm-project/clang/lib/Sema/SemaDecl.cpp:6216:15
    #5 0x51c85c0 in clang::Parser::
       ParseDeclarationAfterDeclaratorAndAttributes(clang::Declarator&,
        clang::Parser::ParsedTemplateInfo const&, clang::Parser::
       ForRangeInit*) llvm-project/clang/lib/Parse/ParseDecl.cpp
       :2517:24
    #6 0x51c13ec in clang::Parser::ParseDeclGroup(clang::
       ParsingDeclSpec&, clang::DeclaratorContext, clang::
       ParsedAttributes&, clang::SourceLocation*, clang::Parser::
       ForRangeInit*) llvm-project/clang/lib/Parse/ParseDecl.cpp
       :2337:21
    #7 0x51bca40 in clang::Parser::ParseSimpleDeclaration(clang::
       DeclaratorContext, clang::SourceLocation&, clang::
       ParsedAttributes&, clang::ParsedAttributes&, bool, clang::Parser
       ::ForRangeInit*, clang::SourceLocation*) llvm-project/clang/lib/
       Parse/ParseDecl.cpp:2030:10
```

where the memory was freed by at:

```
0x60600000b498 is located 24 bytes inside of 56-byte region [0x60600000b480,0x60600000b4b8)
freed by thread T0 here:
    #0 0x5d9472 in __interceptor_free llvm-project/compiler-rt/lib/asan/asan_malloc_linux.cpp:52:3
    #1 0x5112d2d in Destroy llvm-project/clang/include/clang/Sema/ParsedTemplate.h:219:7
    #2 0x5112d2d in clang::Parser::DestroyTemplateIds() llvm-project/clang/lib/Parse/Parser.cpp:581:9
    #3 0x54dd8bd in MaybeDestroyTemplateIds llvm-project/clang/include/clang/Parse/Parser.h:296:7
    #4 0x54dd8bd in clang::Parser::ParseStatementOrDeclaration(llvm::SmallVector<clang::Stmt*, 32u>&, clang::Parser::ParsedStmtContext, clang::SourceLocation*) llvm-project/clang/lib/Parse/ParseStmt.cpp:120:3
    #5 0x550a607 in clang::Parser::ParseCompoundStatementBody(bool) llvm-project/clang/lib/Parse/ParseStmt.cpp:1236:11
    #6 0x52f954b in clang::Parser::ParseBlockLiteralExpression() llvm-project/clang/lib/Parse/ParseExpr.cpp:3748:19
    #7 0x52d7567 in clang::Parser::ParseCastExpression(clang::Parser::CastParseKind, bool, bool&, clang::Parser::TypeCastState, bool, bool*) llvm-project/clang/lib/Parse/ParseExpr.cpp:1782:11
```

The root-cause was found to be that `clang::Parser::MaybeDestroyTemplateIds` is too permissive with the following code:

```
  void MaybeDestroyTemplateIds() {
    if (!TemplateIds.empty() &&
        (Tok.is(tok::eof) || !PP.mightHavePendingAnnotationTokens()))
      DestroyTemplateIds();
  }
```

Specifically, the issue found was discovered to trigger a condition where `Tok.is(tok::eof)` is true by `!PP.mightHavePendingAnnotationTokens()` is false.

The fix is to adjust `clang::Parser::MaybeDestroyTemplateIds` to narrow the check to:

```
  void MaybeDestroyTemplateIds() {
    if (!TemplateIds.empty() &&
        (!PP.mightHavePendingAnnotationTokens()))
      DestroyTemplateIds();
  }
```

4.7 Global-buffer-overflow in llvm::hashing::detail::hash_short

id	ADA-2023-LLVM-7
Monorail ID and URL	65283
Date reported by OSS-Fuzz	2023-12-22
Fix PR:	[BitcodeReader] Add bounds checking on Strtab

A global buffer overflow was reported with the following stacktrace:

```
==47690==ERROR: AddressSanitizer: SEGV on unknown address 0x00000001fff2 (pc 0x000000720be4 bp 0x7fffc3c056f0 sp 0x7fffc3c056c0 T0)
==47690==The signal is caused by a READ memory access.
SCARINESS: 20 (wild-addr-read)
    #0 0x720be4 in hash_1to3_bytes llvm-project/llvm/include/llvm/ADT/Hashing.h:198:15
    #1 0x720be4 in llvm::hashing::detail::hash_short(char const*, unsigned long, unsigned long) llvm-project/llvm/include/llvm/ADT/Hashing.h:260:12
    #2 0xb92be2 in getHashValue llvm-project/llvm/include/llvm/ADT/DenseMap.h:472:12
    #3 0xb92be2 in bool llvm::DenseMapBase<llvm::DenseMap<llvm::StringRef, llvm::detail::DenseSetEmpty, llvm::DenseMapInfo<llvm::StringRef, void>, llvm::detail::DenseSetPair<llvm::StringRef> >, llvm::StringRef, llvm::detail::DenseSetEmpty, llvm::DenseMapInfo<llvm::StringRef, void>, llvm::detail::DenseSetPair<llvm::StringRef> >::LookupBucketFor<llvm::StringRef>(llvm::StringRef const&, llvm::detail::DenseSetPair<llvm::StringRef> const*&) const llvm-project/llvm/include/llvm/ADT/DenseMap.h:653:25
    #4 0xb931cb in LookupBucketFor<llvm::StringRef> llvm-project/llvm/include/llvm/ADT/DenseMap.h:689:9
    #5 0xb931cb in llvm::detail::DenseSetPair<llvm::StringRef>* llvm::DenseMapBase<llvm::DenseMap<llvm::StringRef, llvm::detail::DenseSetEmpty, llvm::DenseMapInfo<llvm::StringRef, void>, llvm::detail::DenseSetPair<llvm::StringRef> >, llvm::StringRef, llvm::detail::DenseSetEmpty, llvm::DenseMapInfo<llvm::StringRef, void>, llvm::detail::DenseSetPair<llvm::StringRef> >::InsertIntoBucketImpl<llvm::StringRef>(llvm::StringRef const&, llvm::StringRef const&, llvm::detail::DenseSetPair<llvm::StringRef>*) llvm-project/llvm/include/llvm/ADT/DenseMap.h:609:7
    #6 0x122f198 in InsertIntoBucket<const llvm::StringRef &, llvm::detail::DenseSetEmpty &> llvm-project/llvm/include/llvm/ADT/DenseMap.h:574:17
```

```
    #7 0x122f198 in try_emplace<llvm::detail::DenseSetEmpty &> llvm-
       project/llvm/include/llvm/ADT/DenseMap.h:271:17
    #8 0x122f198 in insert llvm-project/llvm/include/llvm/ADT/DenseSet.
       h:208:19
    #9 0x122f198 in llvm::UniqueStringSaver::save(llvm::StringRef) llvm
       -project/llvm/lib/Support/StringSaver.cpp:29:19
    #10 0xa00afe in llvm::GlobalValue::setPartition(llvm::StringRef)
       llvm-project/llvm/lib/IR/Globals.cpp:220:35
    #11 0x64a60a in parseGlobalIndirectSymbolRecord llvm-project/llvm/
       lib/Bitcode/Reader/BitcodeReader.cpp:4221:12
    #12 0x64a60a in (anonymous namespace)::BitcodeReader::parseModule(
       unsigned long, bool, llvm::ParserCallbacks) llvm-project/llvm/
       lib/Bitcode/Reader/BitcodeReader.cpp:4511:23
    #13 0x58a4f7 in parseBitcodeInto llvm-project/llvm/lib/Bitcode/
       Reader/BitcodeReader.cpp:4548:10
    #14 0x58a4f7 in llvm::BitcodeModule::getModuleImpl(llvm::
       LLVMContext&, bool, bool, bool, llvm::ParserCallbacks) llvm-
       project/llvm/lib/Bitcode/Reader/BitcodeReader.cpp:8014:22
    #15 0x5a10ce in llvm::BitcodeModule::parseModule(llvm::LLVMContext
       &, llvm::ParserCallbacks) llvm-project/llvm/lib/Bitcode/Reader/
       BitcodeReader.cpp:8215:10
```

The problem was assessed to be within the `BitcodeReader::parseGlobalIndirectSymbolRecord` function where a `StringRef` is constructed to point to a buffer that extends beyond allocated memory:

```
  // Check whether we have enough values to read a partition name.
  if (OpNum + 1 < Record.size()) {
    NewGA->setPartition(
        StringRef(Strtab.data() + Record[OpNum], Record[OpNum + 1]));
    OpNum += 2;
  }
```

The problem is that the generated is meant to point inside of the `Strtab` buffer. However, there is no bounds checking on whether `Record[OpNum + Record[OpNum+1]` extends beyond the buffer of `Strtab`, which means that a `Stringref` may be created that extends beyond the allocated data of ‘Strtab.

Interestingly, in other parts of the same module there are boundary checkings in place:

```
  // Check whether we have enough values to read a partition name. Also
      make
  // sure Strtab has enough values.
  if (Record.size() > 18 && Strtab.data() &&
      Record[17] + Record[18] <= Strtab.size()) {
    Func->setPartition(StringRef(Strtab.data() + Record[17], Record
       [18]));
  }
```

The proposed fix for the issue:

```
  // Check whether we have enough values to read a partition name.
  if (OpNum + 1 < Record.size()) {
    if (Record[OpNum] + Record[OpNum + 1] > Strtab.size()) {
        return ze{6}{8}error("Malformed partition, too large.");
    }
    NewGA->setPartition(
        StringRef(Strtab.data() + Record[OpNum], Record[OpNum + 1]));
    OpNum += 2;
  }
```

4.8 Heap-buffer-overflow in llvm_regcomp

id	ADA-2023-LLVM-8
Monorail ID and URL	65423
Date reported by OSS-Fuzz	2023-12-30
Fix PR	[Support] Fix buffer overflow in regcomp

`OQUEST_` and `OCH_` causes the scan pointer to skip elements in `g`'s `strip` buffer. However, the terminating character of `g->strip` may be within the skipped elements, and there is currently no checking of that. This adds a check on the skipped elements to ensure no overflow happens.

The `findmust` function has the following code:

```
    /* find the longest OCHAR sequence in strip */
    newlen = 0;
    scan = g->strip + 1;
    do {
        s = *scan++;
        switch (OP(s)) {
        case OCHAR:     /* sequence member */
            if (newlen == 0)        /* new sequence */
                newstart = scan - 1;
            newlen++;
            break;
        case OPLUS_:        /* things that don't break one */
        case OLPAREN:
        case ORPAREN:
            break;
        case OQUEST_:       /* things that must be skipped */
        case OCH_:
            scan--;
            do {
                scan += OPND(s);
                s = *scan;
                /* assert() interferes w debug printouts */
                if (OP(s) != O_QUEST && OP(s) != O_CH &&
                            OP(s) != OOR2) {
                    g->iflags |= REGEX_BAD;
                    return;
                }
            } while (OP(s) != O_QUEST && OP(s) != O_CH);
            LLVM_FALLTHROUGH;
        default:        /* things that break a sequence */
```

```
1639                if (newlen > g->mlen) {     /* ends one */
1640                    start = newstart;
1641                    g->mlen = newlen;
1642                }
1643                newlen = 0;
1644                break;
1645            }
1646        } while (OP(s) != OEND);
1647  ~
```

The **do**-**while** loop terminates whenever `OP(s)== OEND)`. However, in the switch statement within the **do** body, in the event `OP(s)` is either `OCQUEST_` or `OCH_` the `scan` pointer will increase by a given amount, and it may be that the elements within the skipped amount contains the `OEND` element. This must be checked, as otherwise future dereferences, such as the dereference on line 1629, will lead to buffer overflows on `g->strip`.

The fix proposed is to add a loop that checks if any of the skipped elements contain the `OEND` element.

4.9 Heap-buffer-overflow in WasmObjectFile::parseLinkingSectionSymtab

id	ADA-2023-LLVM-9
Monorail ID and URL	65432
Date reported by OSS-Fuzz	2023-12-16
Fix PR	[WebAssembly] Limit increase of Ctx.End

The following code in `WasmObjectFile.cpp` leads to possible buffer overflows:

```
531 Error WasmObjectFile::parseLinkingSection(ReadContext &Ctx) {
532   HasLinkingSection = true;
533
534   LinkingData.Version = readVaruint32(Ctx);
535   if (LinkingData.Version != wasm::WasmMetadataVersion) {
536     return make_error<GenericBinaryError>(
537         "unexpected metadata version: " + Twine(LinkingData.Version) +
538             " (Expected: " + Twine(wasm::WasmMetadataVersion) + ")",
539         object_error::parse_failed);
540   }
541
542   const uint8_t *OrigEnd = Ctx.End;
543   while (Ctx.Ptr < OrigEnd) {
544     Ctx.End = OrigEnd;
545     uint8_t Type = readUint8(Ctx);
546     uint32_t Size = readVaruint32(Ctx);
547     LLVM_DEBUG(dbgs() << "readSubsection type=" << int(Type) << " size=" << Size
548                       << "\n");
549     Ctx.End = Ctx.Ptr + Size;
```

The problem is that `Ctx.End` is potentially increased beyond the `OrigEnd` on line 549. Since `Ctx.End` represents the end of a heap-allocated buffer, this can cause memory buffer overflows later on, as the `Ctx.End` is used as the limit of the Ctx's memory buffer.

4.10 [llvm-special-case-list-fuzzer] fix off-by-one read

id	ADA-2023-LLVM-10
Date reported by OSS-Fuzz	2023-08-01
Fix PR	[llvm-special-case-list-fuzzer] fix off-by-one read

The LLVM build had been failing to build and this was due to a broken fuzzer. Reading the build logs, such as this one, we can see the `llvm-special-case-list-fuzzer` runs into an ASAN issue in the first fuzz run:

```
0000000000000000000000000000000000000000\n0xa,\n\\012\nartifact_prefix='./'; Test unit written to ./crash-adc83b19e793491b1c6ea0fd8b46cd9f32e592fc\nBase64: Cg==\n", stderr=b''))
Step #13 - "build-check-libfuzzer-address-x86_64": BAD BUILD: /tmp/not-out/tmptsh8iy49/llvm-special-case-list-fuzzer seems to have either startup crash or exit:
Step #13 - "build-check-libfuzzer-address-x86_64": /tmp/not-out/tmptsh8iy49/llvm-special-case-list-fuzzer -rss_limit_mb=2560 -timeout=25 -seed=1337 -runs=4 < /dev/null
Step #13 - "build-check-libfuzzer-address-x86_64": INFO: Running with entropic power schedule (0xFF, 100).
Step #13 - "build-check-libfuzzer-address-x86_64": INFO: Seed: 1337
Step #13 - "build-check-libfuzzer-address-x86_64": INFO: Loaded 1 modules   (37010 inline 8-bit counters): 37010 [0xa8b098, 0xa9412a),
Step #13 - "build-check-libfuzzer-address-x86_64": INFO: Loaded 1 PC tables (37010 PCs): 37010 [0xa94130,0xb24a50),
Step #13 - "build-check-libfuzzer-address-x86_64": INFO: -max_len is not provided; libFuzzer will not generate inputs larger than 4096 bytes
Step #13 - "build-check-libfuzzer-address-x86_64": INFO: A corpus is not provided, starting from an empty corpus
Step #13 - "build-check-libfuzzer-address-x86_64": =================================================================
Step #13 - "build-check-libfuzzer-address-x86_64": ==408==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x602000000111 at pc 0x0000007ebd7b bp 0x7ffdff69d590 sp 0x7ffdff69d588
Step #13 - "build-check-libfuzzer-address-x86_64": READ of size 1 at 0x602000000111 thread T0
Step #13 - "build-check-libfuzzer-address-x86_64": SCARINESS: 12 (1-byte-read-heap-buffer-overflow)
Step #13 - "build-check-libfuzzer-address-x86_64":     #0 0x7ebd7a in line_iterator /src/llvm-project/llvm/lib/Support/LineIterator.cpp:48:5
```

```
Step #13 - "build-check-libfuzzer-address-x86_64":     #1 0x7ebd7a in llvm::line_iterator::line_iterator(llvm::MemoryBuffer const&, bool, char) /src/llvm-project/llvm/lib/Support/LineIterator.cpp:36:7
Step #13 - "build-check-libfuzzer-address-x86_64":     #2 0x580722 in llvm::SpecialCaseList::parse(llvm::MemoryBuffer const*, std::__1::basic_string<char, std::__1::char_traits<char>, std::__1::allocator<char> >&) /src/llvm-project/llvm/lib/Support/SpecialCaseList.cpp:161:22
Step #13 - "build-check-libfuzzer-address-x86_64":     #3 0x57faa3 in createInternal /src/llvm-project/llvm/lib/Support/SpecialCaseList.cpp:127:8
Step #13 - "build-check-libfuzzer-address-x86_64":     #4 0x57faa3 in llvm::SpecialCaseList::create(llvm::MemoryBuffer const*, std::__1::basic_string<char, std::__1::char_traits<char>, std::__1::allocator<char> >&) /src/llvm-project/llvm/lib/Support/SpecialCaseList.cpp:93:12
Step #13 - "build-check-libfuzzer-address-x86_64":     #5 0x56da34 in LLVMFuzzerTestOneInput /src/llvm-project/llvm/tools/llvm-special-case-list-fuzzer/special-case-list-fuzzer.cpp:22:3
```

The root-cause was determined to be due to the fuzzer relying on MemoryBuffer to hold the fuzz data. However, the fuzzer runs into an OOB instantly because the MemoryBuffer interface guarantees that "In addition to basic access to the characters in the file, this interface guarantees you can read one character past the end of the file, and that this character will read as '\0'." (see this documentation), which the fuzzer fails to satisfy. As such, it runs into an OOB on line 48 in `llvm/lib/Support/LineIterator.cpp`:

```
45    // Ensure that if we are constructed on a non-empty memory buffer that it is
46    // a null terminated buffer.
47    if (Buffer.getBufferSize()) {
48      assert(Buffer.getBufferEnd()[0] == '\0');
49      // Make sure we don't skip a leading newline if we're keeping blanks
```

4.11 NULL-dereference READ in processTypeAttrs

id	ADA-2023-LLVM-11
Monorail ID and URL	20938
Date reported by OSS-Fuzz	2020-02-28
Fix PR	[Clang][Sema] Fix NULL dereferences for invalid references

A NULL-dereference was found with the following stacktrace:

```
    ==21077==ERROR: MemorySanitizer: SEGV on unknown address 0
        x000000000008 (pc 0x00000d0c4f21 bp 0x7ffce9374ae0 sp 0
        x7ffce9374720 T21077)
==21077==The signal is caused by a READ memory access.
==21077==Hint: address points to the zero page.
    #0 0xd0c4f21 in getKind llvm-project/clang/include/clang/Sema/
        ParsedAttr.h:608:43
    #1 0xd0c4f21 in processTypeAttrs((anonymous namespace)::
        TypeProcessingState&, clang::QualType&, TypeAttrLocation, clang
        ::ParsedAttributesView const&, clang::Sema::CUDAFunctionTarget)
        llvm-project/clang/lib/Sema/SemaType.cpp:8743:18
    #2 0xd08af0b in GetFullTypeForDeclarator((anonymous namespace)::
        TypeProcessingState&, clang::QualType, clang::TypeSourceInfo*)
        llvm-project/clang/lib/Sema/SemaType.cpp:5788:5
    #3 0xd0690f8 in clang::Sema::GetTypeForDeclarator(clang::Declarator
        &, clang::Scope*) llvm-project/clang/lib/Sema/SemaType.cpp
        :6082:10
    #4 0x9e246cf in clang::Sema::HandleDeclarator(clang::Scope*, clang
        ::Declarator&, llvm::MutableArrayRef<clang::
        TemplateParameterList*>) llvm-project/clang/lib/Sema/SemaDecl.
        cpp:6436:27
    #5 0x9e234ef in clang::Sema::ActOnDeclarator(clang::Scope*, clang::
        Declarator&) llvm-project/clang/lib/Sema/SemaDecl.cpp:6216:15
    #6 0x83eb185 in clang::Parser::
        ParseDeclarationAfterDeclaratorAndAttributes(clang::Declarator&,
         clang::Parser::ParsedTemplateInfo const&, clang::Parser::
        ForRangeInit*) llvm-project/clang/lib/Parse/ParseDecl.cpp
        :2517:24
```

The proposed fix is to adjust `isInvalid` in `clang/include/clang/Sema/ParsedAttr.h`:

```
  bool isInvalid() const {
    if (&Info == NULL) {
      Invalid = true;
```

```
    }
    return Invalid;
}
```

The problem is that `Info` may end up referencing a NULL pointer, so a check for this should be in place. Ideally the reference should never reference a NULL pointer, however, due to timing constraints we were unable to identify the fix that makes this possible.

4.12 NULL-dereference READ in GetFullTypeForDeclarator

id	ADA-2023-LLVM-12
Monorail ID and URL	20946
Date reported by OSS-Fuzz	2020-02-28
Fix PR	[Clang][Sema] Fix NULL dereferences for invalid references

A NULL-dereference was found with the following stacktrace:

```
==38279==ERROR: MemorySanitizer: SEGV on unknown address 0x000000000008
    (pc 0x00000d0789ae bp 0x7ffdba4dc350 sp 0x7ffdba4db490 T38279)
==38279==The signal is caused by a READ memory access.
==38279==Hint: address points to the zero page.
    #0 0xd0789ae in getKind llvm-project/clang/include/clang/Sema/
       ParsedAttr.h:608:43
    #1 0xd0789ae in hasNullabilityAttr llvm-project/clang/lib/Sema/
       SemaType.cpp:4243:12
    #2 0xd0789ae in GetFullTypeForDeclarator((anonymous namespace)::
       TypeProcessingState&, clang::QualType, clang::TypeSourceInfo*)
       llvm-project/clang/lib/Sema/SemaType.cpp:5229:12
    #3 0xd0690f8 in clang::Sema::GetTypeForDeclarator(clang::Declarator
       &, clang::Scope*) llvm-project/clang/lib/Sema/SemaType.cpp
       :6082:10
    #4 0x9e246cf in clang::Sema::HandleDeclarator(clang::Scope*, clang
       ::Declarator&, llvm::MutableArrayRef<clang::
       TemplateParameterList*>) llvm-project/clang/lib/Sema/SemaDecl.
       cpp:6436:27
    #5 0x9e234ef in clang::Sema::ActOnDeclarator(clang::Scope*, clang::
       Declarator&) llvm-project/clang/lib/Sema/SemaDecl.cpp:6216:15
    #6 0x83eb185 in clang::Parser::
       ParseDeclarationAfterDeclaratorAndAttributes(clang::Declarator&,
        clang::Parser::ParsedTemplateInfo const&, clang::Parser::
       ForRangeInit*) llvm-project/clang/lib/Parse/ParseDecl.cpp
       :2517:24
```

The proposed fix includes the logic from 4.11 and also adjusts `hasNUllabilityAttr` to include a check on each attribute iterated, to test if it is valid or not:

```
static bool hasNullabilityAttr(const ParsedAttributesView &attrs) {
  for (const ParsedAttr &AL : attrs) {
    if (AL.isInvalid()) {
      continue;
```

```
    }
    if (AL.getKind() == ParsedAttr::AT_TypeNonNull ||
        AL.getKind() == ParsedAttr::AT_TypeNullable ||
        AL.getKind() == ParsedAttr::AT_TypeNullableResult ||
```