ADALOGICS

Node.js fuzzing audit

In collaboration with Open Source Technology Improvement Fund and Sovereign Tech Fund

Adam Korczynski, David Korczynski

2024-06-29

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 dedicated, pragmatic security engineers and security researchers that work hands-on with code auditing, security automation and security tooling.

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. You can also follow Ada Logics on Linkedin, Twitter and Youtube.

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

Contents

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

Executive Summary

Ada Logics conducted a security assurance audit of Node.js at the end of November and December 2023. The audit's goal was to improve the continuous fuzzing setup of the Node.js ecosystem. 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. Throughout the engagement, we first analyzed the existing Node.js fuzzing setup, fixed an OSS-Fuzz build that had been broken for several months, added new ClusterFuzzLite integrations for Node.js dependencies, and added more than 45 new fuzzers to the existing fuzzing suite, which only had a single fuzzer. These now run as part of the Node.js OSS-Fuzz setupsetup.

In summary, during the engagement, we:

- Fixed a broken Node.js OSS-Fuzz build, including fuzzing and code coverage.
- Created three new ClusterFuzzLite integrations to core Node.js dependencies.
- Extended the OSS-Fuzz fuzzing setup of Node.js with 48 new fuzzers.
- Documented four security findings found by the fuzzers.

Node.js threat model

Node.js is interesting from the perspective of fuzzing in that it consists of both a Javascript layer and an underlying layer implemented in C++ and C. At the Javascript layer are Node.js's Javascript APIs for the internal modules such as buffer, fs, tls, etc. This layer is located and maintained in the `lib` directory in the Node.js source tree. At the lower C++ and C layer are Node.js C++ bindings and Node.js core dependencies such as V8, LibUV, C-Ares, Zlib, etc. Node.js C++ bindings are maintained in the `src` directory, and its core dependencies are maintained in its `deps` directory. At a high level, Node.js's architecture can be visualized as shown in Figure 1.

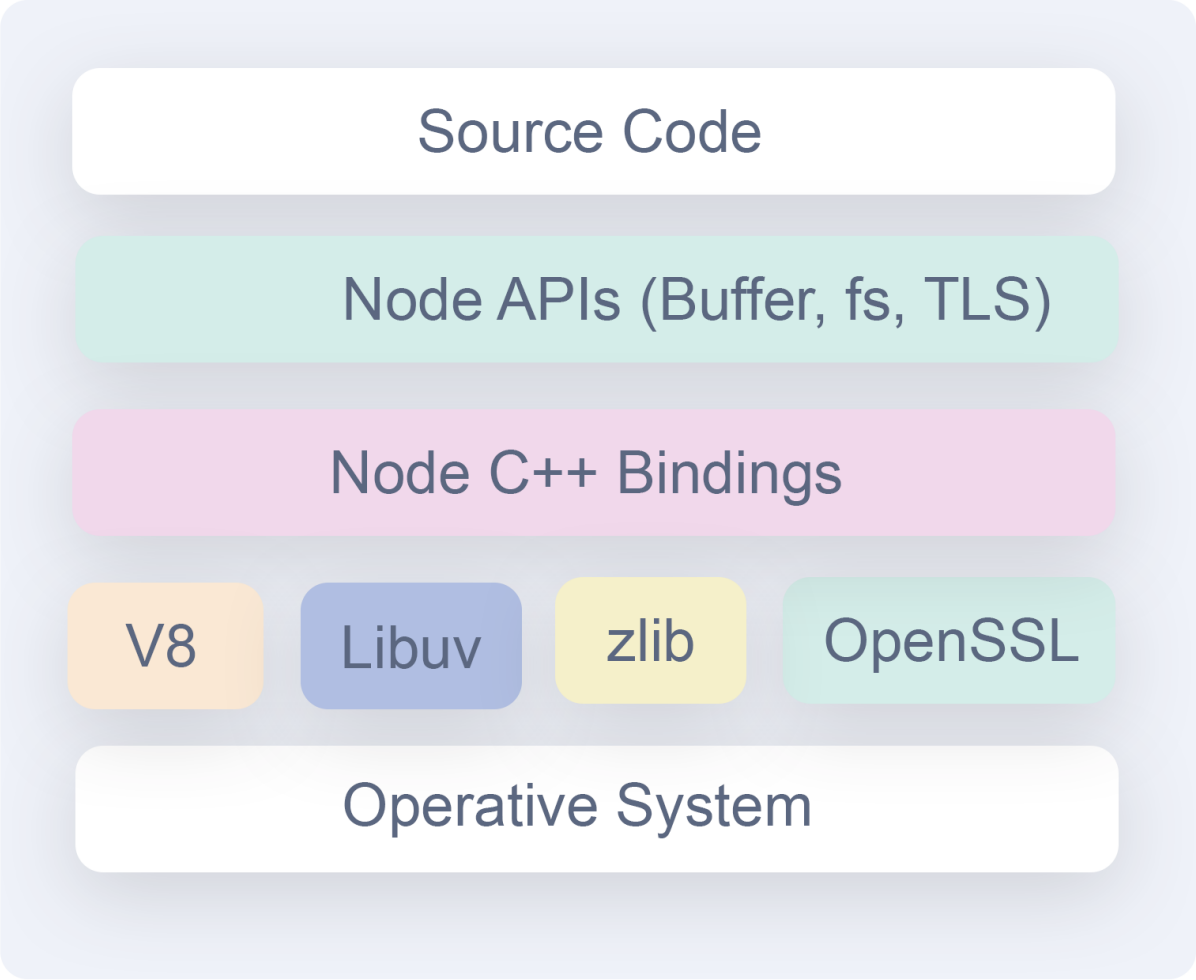

Figure 1: Node.js high-level architecture.

In the above diagram, the "Source Code" layer at the top represents the code Node.js users write and deploy as their Node.js applications. As Node.js executes the users source code, invocation travels

downwards into the Node.js .js APIs to the Node.js C++ bindings to the core dependencies and finally to the operating system; Node.js first invokes the APIs that the user has adopted, then its C++ bindings, and then its dependencies which rely on the operating system depending on Node.js module. For example, Node.js will invoke OpenSSL if the user consumes crypto-related Node.js modules.

When users expose Node.js to untrusted input, this input enters Node.js system via the source code layer at the very top. From the perspective of Nodes threat model, this includes any way of accepting input, such as forms, raw http requests, or something third.

Threats

In this section, we consider the threats that Node.js faces and abstract them into high-level observations that we can use to audit Node.js against.

When we consider threats in Node.js's threat model, we first of all want to think of threats that apply to all or almost all use cases. Node.js can be used in many different ways, in many different environments and can be exposed to users of different threat levels. Some negative behaviour may be a big issue to some users but might not have any criticality at all to others. As a general rule of thumb, when assessing Node.js's security, we consider the use cases that accept untrusted input. The input can be highly untrusted or less untrusted. An application that is exposed to anyone on the Internet accepts fully untrusted input, whereas an application that a teacher uses to collect the homework from their students accepts less untrusted input.

Node.js maintains a public threat model that describes the scope of what is considered a security issue in Node.js, which is available here.

Node.js threat model explicitly does not trust the following data inputs as shown in Figure 2.

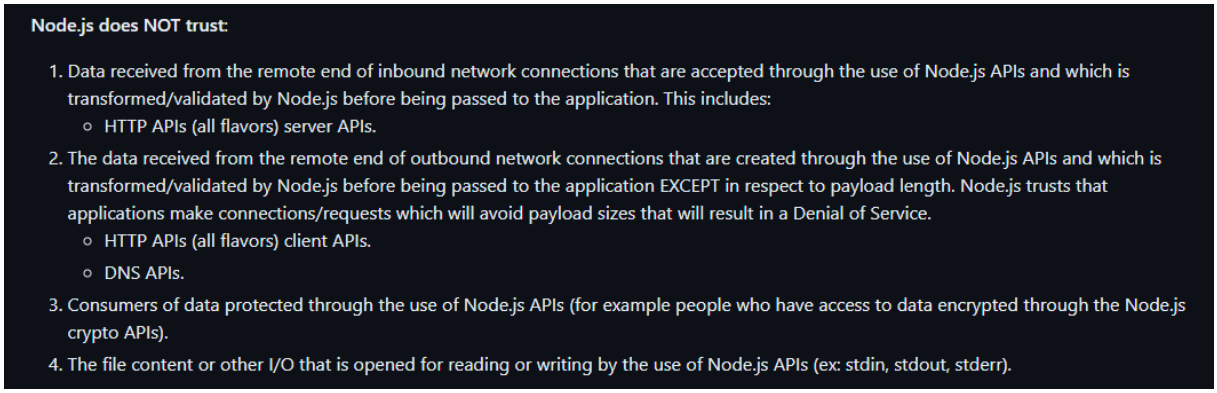

Figure 2: Snippet of the Node.js threat model describing untrusted data.

Node.js is susceptible to the vulnerability class of memory corruption issues because of its large

dependency on memory-unsafe languages. The majority of Node.js code base is implemented in C++ or C if we include its core dependencies into our consideration. Much of the input from untrusted users travels through Node.js memory-unsafe layers to its core dependencies. Node.js's threat model does not explicitly mention memory corruption or other vulnerability classes in memory-unsafe languages, but it does include a section outlining issues that can often be triggered by memory corruption issues, as shown in Figure 3.

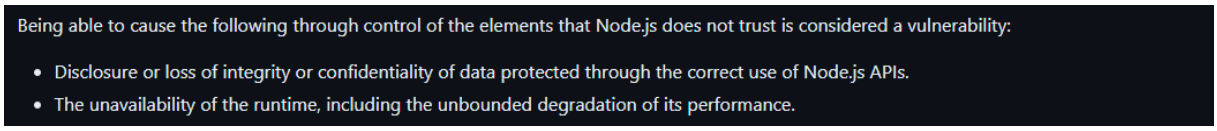

Figure 3: Snippet of the Node.js threat model outlining specific inclusions of issues.

Memory corruptions and vulnerability classes specific to Node.js's underlying C/C++ layers have the potential to achieve all of these objectives. In cases in Node.js's execution flow, where user-supplied data travels from the .js layer to Node.js's C/C++ layers, the C/C++ are also exposed to untrusted input. The issue is exploitable if such input can trigger issues in the C/C++ layers. We consider any case where an untrusted user can trigger memory corruptions, UAFs (use-after-free), and integer overflows in Node.js's C/C++ layers to be undesirable and should be audited for security criticality. Some examples of this could be:

1. A Node.js application is exposed to the internet, and users can register and log in. The username is an email address and the application checks that the email contains exactly one "@" symbol. The application thereby processes incoming text strings. A user registering their account in the application can create a string that triggers an issue in Node.js's underlying C/C++ code that can escalate privileges.
2. A Node.js application is exposed internally to handle document storage. The organization needs to maintain strict access controls against the documents. A temporary worker is able to upload a document with a name that corrupts memory.

In the next section, we discuss how to design the fuzzing workflow so that the fuzzers can test scenarios where untrusted users send data to Node.js.

Fuzzing Node.js

In this section, we describe the fuzzing efforts part of the engagement. We will go over how to approach fuzzing Node.js, the fuzzing setup that is in place for Node.js, and the efforts we carried out to expand the fuzzing of Node.js.

A significant part of the Node.js ecosystem is written in memory-unsafe languages, specifically C and C++. Furthermore, a substantial portion of this codebase comprises dependencies used by Node.js and not first-party code written by Node.js authors. We will consider the fuzzing setup of Node.js from a whole-ecosystem perspective, including considerations around the third-party dependencies that compose most of Node.js's memory-unsafe code.

The overall steps of the fuzzing part of this engagement were as follows:

1. Analyze the existing fuzzing ecosystem of Node.js.
2. Repair the existing Node.js OSS-Fuzz setup.
3. Outline potential strategies for fuzzing Node.js continuously.
4. Extend the existing fuzzing suite.

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

Existing Node.js fuzzing and repair

Continuous fuzzing background

Continuity is an essential element of every robust fuzzing suite. The reason for this is that fuzzers need to explore the execution paths of the application they test, and this takes time. A fuzzer with a sizeable potential call tree may require weeks or months to generate test cases that reach the furthest parts of its call tree. If such a fuzzer only runs for a couple of hours, it will not have enough time to explore the code, let alone test the entire call tree. An essential part of open-source fuzzing infrastructure is the OSS-Fuzz and ClusterFuzzLite projects, which solve the problem of continuous fuzzing for open source software.

OSS-Fuzz is an open-source project by Google that offers large amounts of computing power and implements the necessary infrastructure to automate continuous fuzzing at scale for critical open-source projects. In practice, critical open-source projects integrate into OSS-Fuzz and set up their fuzzers so they can run in the OSS-Fuzz environment. Once done, OSS-Fuzz will run the project's fuzzes periodically with thousands of CPUs. ClusterFuzzLite is a lighter version of OSS-Fuzz that enables continuous fuzzing to run as part of the CI, such as by way of GitHub actions.

Continuous fuzzing of Node.js dependencies

Node.js's core dependencies are established independent projects. Most of these are maintained by third parties, with a single dependency maintained by the Node.js project. Node.js's security posture inherits the security posture of its third-party dependencies. For Node, it is essential that its core dependencies - with an emphasis on the dependencies implemented in memory-unsafe languages - maintain their own fuzzing infrastructure. Below is a table that gives an overview of the fuzzing efforts of Nodes memory-unsafe dependencies:

#	Name	Continuous fuzzing	OSS-Fuzz Code coverage
1	ada-url	oss-fuzz/projects/ada-url	75.59% coverage report
2	base64	No, but OSS-Fuzz proposed	-
3	brotli	oss-fuzz/projects/brotli	80.98% coverage report
4	c-ares	oss-fuzz/projects/c-ares	31.73% coverage report
5	histogram	None	-
6	icu-small	oss-fuzz/projects/icu	43.43% coverage-report
7	llhttp	oss-fuzz/projects/llhttp	65.44% coverage-report
8	nghttp2	oss-fuzz/projets/nghttp2	54.67% coverage-report
9	ngtcp2	ClusterFuzzLite: ngtcp2/.clusterfuzzlite	x
10	nghttp3	ClusterFuzzLite: nghttp3/.clusterfuzzlite	x
11	OpenSSL	oss-fuzz/projects/openssl	38.45% coverage-report
12	SimdJson	oss-fuzz/projects/simdjson	49.04% coverage-report
13	SimdUTF	oss-fuzz/projects/simdutf	36.50% coverage-report
14	LibUV	None	x
15	uvwasi	None	x
16	v8	yes, as part of Chromium fuzzing	x
17	zlib	oss-fuzz/projects/zlib	69.73% coverage-report

The state of Node.js's core memory-unsafe dependencies' fuzzing efforts is highly positive. Most dependencies are integrated into OSS-Fuzz or ClusterFuzzLite and have a high level of fuzz coverage. Throughout this engagement, we integrated continuous fuzzing into the three remaining dependencies: LibUV, uvwasi and histogram.

Continuous fuzzing of Node.js

Node.js itself was integrated into OSS-Fuzz in May 2020, and this integration was done by David Korczynski, who was also an auditor in this fuzzing engagement. The setup initially included a fuzzer for URL parsing logic and a more general fuzzer that executes fuzzer-seeded data as javascript programs by way of Node.js. The URL fuzzer no longer exists due to changes to URL logic in Node.js (now `ada-url` is used). However, the more general fuzzer has run continuously since then and has found issues in the Simd dependency:

55010	Bug-Security	----	Verified	nodejs	2023-01-11	----	nodejs:fuzz_env: Heap-buffer-overflow in unsigned long simdutf::haswell::convert_masked_utf8_to_utf16< ClusterFuzz Reproducible
55013	Bug	----	Verified	nodejs	2023-01-11	----	nodejs:fuzz_env: Null-dereference WRITE in unsigned long simdutf::haswell::convert_masked_utf8_to_utf16< ClusterFuzz Reproducible

Figure 4: Bugs in the Node.js OSS-Fuzz bug tracker listed here

At the beginning of this engagement, the Node.js OSS-Fuzz build had been broken for a few months and the coverage build had been broken for more than a year, since October 2022, This means that OSS-Fuzz could not build the fuzzers and there were no coverage reports. The first task was, therefore, to repair the OSS-Fuzz, including both the fuzzing build and the code coverage build.

We fixed the fuzzing build by adding a missing initialization step in the fuzzer, and we fixed this in this pr. However, the process of merging took at few weeks so we decided to make OSS-Fuzz use a forked version of Node.js with the fixes (as well as further extensions we discuss later) to enable the fuzzing to run (OSS-Fuzz PR).

The code coverage build was more tricky to handle as even after fixing the actual build, OSS-Fuzz would run into trouble because it would not be able to link all the executables due to resource limitations. The problem is that code coverage builds require a lot of memory available and in combination with Node.js having a significant amount of code (more than a million lines of C/C++ cod) within the linked executables OSS-Fuzz would stop the build process due to memory constraints.

To overcome the issue with limited resources, we took two steps: (1) limit the number of CPUs during link time to a maximum of 1 and (2) limit the amount of code being instrumented with code coverage visualization logic. In particular, (2) was more tricky because we have to adjust the build process of Node.js with respect to the compilation flags provided by OSS-Fuzz, and the caveat is we will not have code coverage visualization of the dependencies within the Node.js code coverage reports. However, the alternative is no code coverage reports, so we considered the best option to ensure we have code coverage visualization of the code in the `src` folder. The change can be seen in this pr for limiting instrumentation and this pr for limiting CPUs.

High-level options for targeting Node

The main goal of our fuzzing efforts was twofold: (1) increase the amount of native code being fuzzed within the Node.js ecosystem and (2) do it in a manner that uses continuous fuzzing by way of OSS-Fuzz or ClusterFuzzLite. To this end, we considered three high-level approaches for targeting the native code, and we will iterate through each of them in the following sections.

Fuzzing the full end-to-end-workflow

The first option is to imitate the entire workflow of how Node.js accepts untrusted input to an application, how the untrusted input is passed onto the internal javascript modules, how the input then travels to the Node.js C++ bindings onto the C/C++ dependencies and finally, potentially to operating system calls. The pros of following this approach are that the fuzzer imitates real or near-real behaviour, meaning that if the fuzzer finds an issue, it is likely that the issue is a true positive. The con of this approach is that it is expensive at runtime and potentially for the developer to include a full environment in a single test.

In the case of Node, this workflow entails setting up all the layers in the Node.js architecture: The source code layer, the internal JS modules, the C++ bindings and the dependencies. The input from the fuzzers would enter the system via the source code layer, ie. the fuzzer would set up an application, and the test case from the fuzzer would be passed to the application.

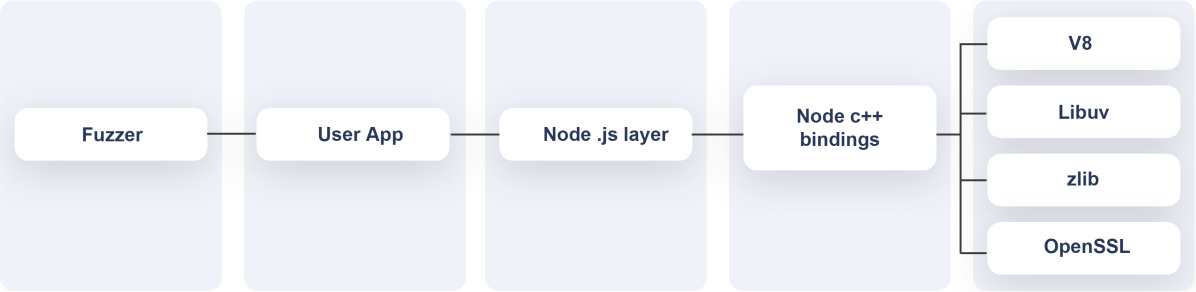

Figure 5: Node.js fuzzing option one, end-to-end fuzzing strategy.

Fuzzing internal APIs directly

With this approach, we can set up fuzzing that calls Nodes internal APIs directly. In a real-world scenario, these internal APIs can be the third or fourth link in the execution chain of Node, and instead of testing the first two links, we can call the internal APIs directly. The pro of this is that some internal APIs are excellent for fuzzing directly. This can be a result of the project having modularized APIs that do heavy

processing over input. By fuzzing internal APIs, we can focus the fuzzer on complex code rather than spending runtime cycles in irrelevant parts of the code. The con is that such fuzzer might not include all sanitization rules or checks that the code base invokes prior to the process reaching the internal APIs, and as a result, the fuzzer may find and report false positives.

In Node, with this approach, we can fuzz any layer in Nodes architecture that processes untrusted data. By doing so, we can test the reliability and security of specific parts of Node.js in a focused manner.

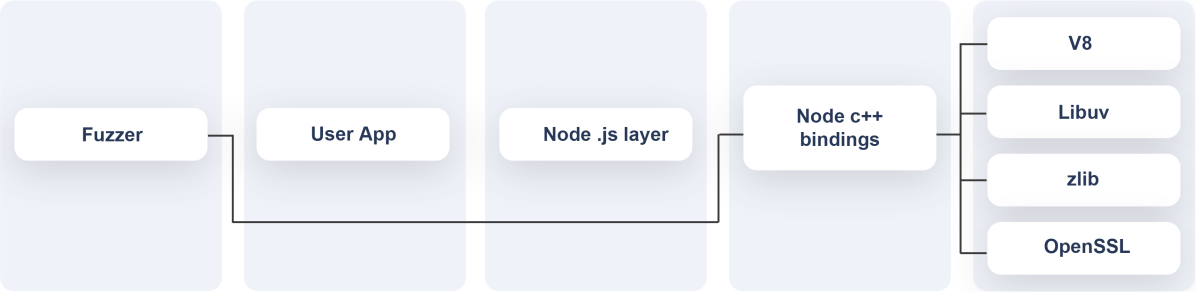

Figure 6: Node.js fuzzing option two, fuzzing native API directly.

Multiple calls

We have the option to fuzz a single Node.js API as well as fuzzing multiple different APIs in the same fuzz harness. In some cases, it might not be the API that we pass random data to that has a bug, but to find the bug in API b, we need first to call API a. For example, the Node.js Buffer module has APIs to create a buffer (Buffer.New()), but it also has APIs to process over buffers. Consider this program:

```
1 var buf = Buffer.from('Creating new buffer');
2
3 console.log(buf.indexOf('new'));
```

If we are to fuzz test `buf.indexOf()`, we first need a buffer to search against. To test this behaviour by way of fuzzing, we also need two different inputs, one when creating a new buffer and one when calling `buf.indexOf()`.

These three approaches do not need to be mutually exclusive, in fact, it is favorable to employ all three in a fuzzing setup and we did this as described in the next section.

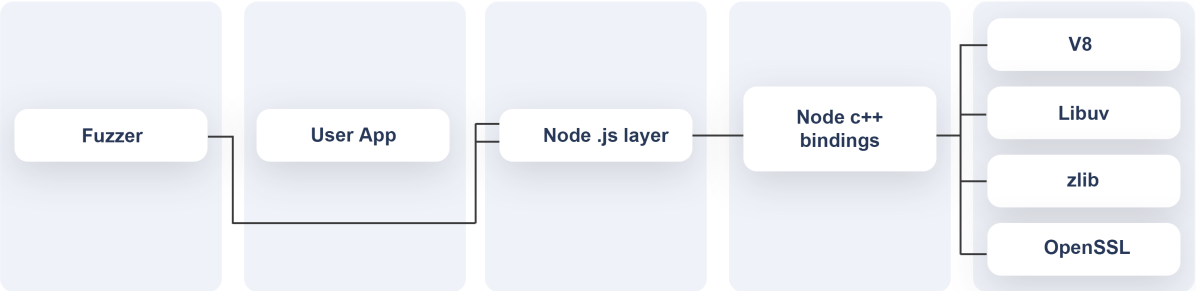

Figure 7: Node.js fuzzing option 3, using more complex .js files seeded with fuzz data.

Expanding the Node.js fuzzing suite

In this section, we outline how we expanded the fuzzing suite, including how we expanded the dependencies being fuzzed and fuzzing of the main Node.js source code.

Fuzzing remaining dependencies

Significantly, we have introduced fuzzing to the three dependencies not in OSS-Fuzz, namely:

HdrHistogram_c: This dependency is maintained in the node/deps/histogram folder and is a mirror of HdrHistogram_c. Currently, the latest version of HdrHistogram_c is 0.11.8, and Node.js is using 0.11.7, although there seems to be no semantic difference between 0.11.7 and 0.11.8.

The code of `HdrHistogram_c` is around 3000 lines of code (counted with `cloc`), so it is likely too small for continuous integration into OSS-Fuzz. To this end, we integrated the project into ClusterFuzzLite.

We added a fuzzer in Add fuzzing by way of ClusterFuzzLite that creates a histogram seeded with fuzz data and then calls various functions used by Node.js on the histogram. The fuzzer found memory leaks in three places of similar nature and also found a division by zero issues (ADA-2023-NODE-3 and ADA-2023-NODE-4). We added fixes for these here and here, respectively. The core of the fuzzer is as follows, where `filename` points to a file with data generated by the fuzzer:

```
1   // open FP to the log file
2   fp = fopen(filename, "r");
3   if (hdr_log_reader_init(&reader)) {
4     return 0;
5   }
6
7   rc = hdr_log_read_header(&reader, fp);
8   if (rc) {
9     fclose(fp);
```

```
10      unlink(filename);
11      return 0;
12    }
13
14    // Output to /dev/null
15    FILE *fp_dev_null = fopen("/dev/null", "w");
16
17    rc = hdr_log_read(&reader, fp, &h, &timestamp, &interval);
18
19    if (0 == rc) {
20      // Call functions used by Node.js
21      hdr_min(h);
22      hdr_max(h);
23      hdr_mean(h);
24      hdr_stddev(h);
25      hdr_value_at_percentile(h, 50.0);
26      hdr_get_memory_size(h);
27
28      hdr_percentiles_print(h, fp_dev_null, 5, 1.0, CLASSIC);
29      hdr_close(h);
30    }
```

Base64: the base64 dependency is pulled from github.com/aklomp/base64. The central feature used is that this library utilizes SIMD operations where possible. This library has an existing pull request focused on integrating the project into OSS-Fuzz, which also contains two fuzzing harnesses here. The fuzzers in this PR look good and cover the important entrypoints. As there is already a PR in place for integrating fuzzing we decided not to add another one but rather let the maintainers of `base64` act based on the existing PR.

LibUV: the libuv dependency is pulled from github.com/libuv/libuv. This library provides various asynchronous I/O related features, e.g. file system operations, and was a library primarily used for Node.js.

LibUV is a central component in the Node.js source. However, it is not a library that is generally a good fuzzing target since there is limited data processing in the library. However, libuv has previously had a vulnerability that affected Node.js in the form of CVE-2020-8252. As such, we decided to implement a ClusterFuzzLite integration into LibUV and the PR for this is available here. The fuzzer targets various functions with some data handling routines, e.g. `uv_ip4_addr`, `uv_ip6_addr`, the vulnerable function from CVE-2020-8252 (`uv_fs_realpath`) and some initial file system operations fuzzing. There are more options available, such as creating more sophisticated state-machines that can be used to fuzz arbitrary sequences of operations provided by LibUV.

uvwasi: the uvwasi dependency is pulled from github.com/nodejs/uvwasi. This library is used for implementing the Web Assembly System Interface (WASI) and uses LibUV throughout to ensure optimal portability. Similar to LibUV, this library does not have a lot of data handling or parsing, although there

are a few places where paths are analyzed (e.g., normalized) and contain data handling logic. To this end, we opted for a ClusterFuzzLite integration as opposed to a full OSS-Fuzz integration.

We added a fuzzer targeting the path resolution logic in `uvwasi/src/path_resolver.c` and specifically `uvwasi__normalize_path`. This fuzzer ran quickly into a write-based buffer overflow due to an off-by-one issue in the parsing logic, namely `ADA-2023-NODE-1` and `ADA-2023-NODE-2`.

Adding new fuzzers targeting Node.js APIs

We expanded the existing fuzzing suite of Node.js, which had only a single fuzzer, to a total of 49 fuzzers. All of the fuzzers are present in the test/fuzzers folder of the forked repository of node, under the `all-new-fuzzers` branch.

The following table gives an overview of the fuzzers and their respective Node.js API targets.

#	Fuzzer name	Primary Node.js API target
1	fuzz_ClientHelloParser	`node::crypto::ClientHelloParser::Parse`
2	fuzz_LoadBIO	`node::crypto::LoadBIO`
3	fuzz_ParseCaaReply	`node::cares_wrap::ParseTxtReply`
4	fuzz_ParseCaaReply	`node::cares_wrap::ParseGeneralReply`
5	fuzz_ParseMxReply	`node::cares_wrap::ParseMXReply`
6	fuzz_ParseNaptrReply	`node::cares_wrap::ParseNaptrReply`
7	fuzz_ParsePublicKey	`node::crypto::ManagedEVPPKey::ParsePublicKeyPEM`
8	fuzz_ParseSoaReply	`node::cares_wrap::ParseSoaReply`
9	fuzz_ParseSrvReply	`node::cares_wrap::ParseSrvReply`
10	fuzz_ParseTxtReply	`node::cares_wrap::ParseTxtReply`
11	fuzz_blob	`Blob.text`
12	fuzz_buffer_compare	`Buffer.compare`
13	fuzz_buffer_equals	`Buffer.equals`
14	fuzz_buffer_includes	`Buffer.includes`

#	Fuzzer name	Primary Node.js API target
15	fuzz_cipheriv	`crypto.createCipheriv`, `cipher.update`, `cipher.final`, `crypto.createDecipheriv`, `decipher.update`, `decipher.final`
16	fuzz_creativePriveKeyDER	`crypto.createPrivateKey (DER format)`
17	fuzz_createPrivateKeyJWK	`crypto.createPrivateKey (JWK format)`
18	fuzz_createPrivateKeyPEM	`crypto.createPrivateKey (PEM format)`
19	fuzz_diffieHellmanDER	`crypto.diffieHellman (DER format)`
20	fuzz_diffieHellmannJWK	`crypto.diffieHellman (JWK format)`
21	fuzz_diffieHellmanPEM	`crypto.diffieHellman (PEM format)`
22	fuzz_fs_write_open_read	`fs.writeFileSync`, `fs.open`, `fs.read`
23	fuzz_fs_write_read_append	`fs.writeFileSync`, `fs.readFile`, `fs.readFileSync`, `fs.appendFile`
24	fuzz_httpparser1	`_http_common.HTTPParser.execute`
25	fuzz_path_basename	`path.basename`
26	fuzz_path_dirname	`path.dirname`
27	fuzz_path_extname	`path.extname`
28	fuzz_path_format	`path.format`
29	fuzz_path_isAbsolute	`path.isAbsolute`
30	fuzz_path_join	`path.join`
31	fuzz_path_normalize	`path.normalize`
32	fuzz_path_parse	`path.parse`
33	fuzz_path_relative	`path.relative`
34	fuzz_path_resolve	`path.resolve`
35	fuzz_path_toNamespacedPath	`path.toNamespacedPath`
36	fuzz_querystring_parse	`path.querystring.parse`
37	fuzz_quic_token	`node::quic::TokenSecret::Validate`, `node::quic::RegularToken::Validate`
38	fuzz_sign_verify	`sign.sign`, `verify.update`, `verify.verify`

#	Fuzzer name	Primary Node.js API target
39	fuzz_stream1	`readable.push`, `chunk.toString`
40	fuzz_string_decoder	`StringDecoder.write`, `StringDecoder.end`
41	fuzz_strings	`node_api_symbol_for`, `napi_set_named_property`, `napi_get_named_property`, `napi_has_named_property`, `napi_get_property_names`, `napi_has_property`, `napi_get_property`, `napi_delete_property`, `napi_has_own_property`, `napi_create_type_error`, `napi_create_range_error`, `node_api_create_syntax_error`, `napi_run_script`
42	fuzz_tls_socket_request	TLS socket requests
43	fuzz_v8_brotli_deserialize	`v8.deserialize`
44	fuzz_x509	`node:crypto:X509Certificate`, `x509.subject`, `x509.checkEmail`, `x509.checkHost`, `x509.checkIP`, `x509.fingerprint`, `x509.fingerprint512`, `x509.infoAccess`, `x509.issuer`, `x509.issuerCertificate`, `x509.extKeyUsage`, `x509.raw`, `x509.serialNumber`, `x509.subject`, `x509.subjectAltName`, `x509.toJSON()`, `x509.toLegacyObject()`, `x509.toString()`, `x509.validFrom`, `x509.validTo`, `x509.verify(x509.publicKey);`
45	fuzz_zlib_brotliCompress	`zlib.brotliCompress`
46	fuzz_zlib_brotliDecompress	`zlib.brotliDecompress`
47	fuzz_zlib_createBrotliDecompress	`zlib.createBrotliDecompress`
48	fuzz_zlib_gzip_createUnzip	`zlib.createUnzip`

Coverage analysis of new fuzzers

The most recent coverage report prior to this engagement was from 26th October 2022, as the coverage build had been broken since then. Figure 8 shows the state of coverage overall before this engagement and Figure 9 shows the files in `src` with code coverage at the same point in time.

Coverage Report

View results by: Directories | Files

PATH	LINE COVERAGE	FUNCTION COVERAGE	REGION COVERAGE
deps/	0.75% (6251/838447)	1.22% (839/68710)	1.13% (8395/744697)
out/	1.35% (316/23337)	0.05% (2/4319)	0.02% (2/11494)
src/	3.62% (2156/59518)	2.73% (146/5354)	1.77% (1365/76978)
test/	30.21% (29/96)	25.00% (3/12)	18.18% (8/44)
TOTALS	0.95% (8752/921398)	1.26% (990/78395)	1.17% (9770/833213)

Figure 8: Code coverage of Node.js on OSS-Fuzz before this engagement report here

crypto/	0.00% (0/11226)	0.00% (0/803)	0.00% (0/12570)
node_builtins.cc	1.01% (5/494)	3.12% (1/32)	0.49% (1/203)
node_http2.cc	1.38% (31/2250)	0.62% (1/160)	0.26% (8/3127)
node_watchdog.cc	2.33% (6/258)	3.57% (1/28)	2.36% (6/254)
node_binding.cc	3.42% (12/351)	9.52% (2/21)	18.01% (65/361)
tracing/	3.91% (37/947)	5.62% (9/160)	4.19% (26/621)
util.cc	3.92% (12/306)	3.85% (1/26)	7.63% (9/118)
node_url.h	5.00% (3/60)	5.56% (1/18)	5.56% (1/18)
node_union_bytes.h	6.45% (2/31)	15.38% (2/13)	5.00% (2/40)
util-inl.h	7.47% (23/308)	7.14% (4/56)	16.00% (52/325)
node_options.h	7.50% (3/40)	13.04% (3/23)	7.32% (3/41)
node_i18n.cc	8.76% (48/548)	6.25% (2/32)	5.81% (35/602)
node.cc	11.29% (79/700)	10.00% (3/30)	7.52% (50/665)
node_credentials.cc	11.96% (36/301)	13.64% (3/22)	4.83% (20/414)
node_platform.cc	20.47% (96/469)	16.44% (12/73)	21.11% (84/398)
util.h	25.58% (44/172)	20.69% (12/58)	16.93% (32/189)
node_options-inl.h	43.92% (130/296)	71.88% (23/32)	24.06% (51/212)
node_mutex.h	51.61% (48/93)	48.48% (16/33)	45.61% (26/57)
node_url.cc	51.89% (795/1532)	57.41% (31/54)	50.54% (790/1563)
node_options.cc	64.97% (690/1062)	52.00% (13/25)	19.88% (64/322)
node_metadata.cc	94.83% (55/58)	83.33% (5/6)	97.50% (39/40)
node_binding.h	100.00% (1/1)	100.00% (1/1)	100.00% (1/1)
TOTALS	3.62% (2156/59518)	2.73% (146/5354)	1.77% (1365/76978)

Figure 9: Code coverage of `node/src` folder before this engagement report here

At the time of writing, not all new fuzzers have yet been included in the OSS-Fuzz code coverage report. Approximately 25% of them are missing. However, to give some insights of the progress we can rely on the coverage report as it is the 14th January 2024.

The code coverage in the same locations following addition of new fuzzers is shown in Figure 10 and Figure 11.

In total, we can see progress such as:

1. 1400 more functions are analyzed within the `src` folder.
2. Code coverage went from 3.6% to 21.7% within the `src` folder.
3. 130 files in the `src` folder now have fuzzing coverage, whereas previously, it was 21.

It is noteworthy that the added coverage is spread well out over many files as shown by Figure 11 and also the amount of new functions analyzed. As such, a large breadth of new code was analyzed with the additions of the new fuzzers.

We anticipate once data from the remaining fuzzers are included the coverage will increase to around 35% in the `src` folder and steadily increase over time.

Coverage Report

View results by: Directories | Files

PATH	LINE COVERAGE	FUNCTION COVERAGE	REGION COVERAGE
deps/	6.82% (1673/24518)	10.99% (304/2767)	6.82% (802/11751)
out/	0.15% (6/4059)	0.37% (2/546)	0.09% (2/2201)
src/	21.74% (15199/69918)	23.68% (1542/6513)	14.36% (13032/90741)
test/	94.41% (2685/2844)	82.33% (205/249)	51.23% (811/1583)
TOTALS	19.30% (19563/101339)	20.38% (2053/10075)	13.78% (14647/106276)

Figure 10: Code coverage of Node.js on OSS-Fuzz at end of this engagement report here

node_crypto.cc	50.00% (7/14)	50.00% (1/2)	52.63% (30/57)
node_dotenv.h	50.00% (1/2)	50.00% (1/2)	50.00% (1/2)
node_util.cc	51.42% (127/247)	33.33% (5/15)	31.61% (61/193)
node_blob.cc	53.56% (233/435)	45.95% (17/37)	24.43% (97/397)
node_zlib.cc	54.12% (479/885)	53.33% (40/75)	50.04% (701/1401)
aliased_buffer-inl.h	54.21% (58/107)	61.11% (11/18)	50.00% (33/66)
util-inl.h	55.22% (185/335)	63.79% (37/58)	36.61% (123/336)
timers.cc	57.26% (67/117)	50.00% (10/20)	35.48% (22/62)
node_messaging.h	60.00% (6/10)	66.67% (4/6)	50.00% (4/8)
env-inl.h	60.03% (371/618)	61.63% (106/172)	41.14% (137/333)
base64-inl.h	62.81% (76/121)	100.00% (7/7)	52.67% (69/131)
string_decoder.cc	64.55% (142/220)	57.14% (4/7)	43.03% (105/244)
node_options.h	64.86% (24/37)	69.57% (16/23)	66.67% (18/27)
handle_wrap.cc	65.69% (67/102)	57.14% (8/14)	49.52% (52/105)
node_platform.cc	66.81% (318/476)	60.81% (45/74)	43.98% (190/432)
base_object.cc	69.60% (87/125)	50.00% (10/20)	48.48% (80/165)
node_context_data.h	70.00% (14/20)	100.00% (2/2)	72.73% (8/11)
string_decoder-inl.h	70.59% (12/17)	80.00% (4/5)	80.00% (4/5)
node_process_object.cc	70.78% (109/154)	33.33% (3/9)	66.36% (142/214)
base_object.h	75.00% (3/4)	50.00% (1/2)	50.00% (1/2)
node_api_internals.h	75.00% (3/4)	50.00% (1/2)	50.00% (1/2)
handle_wrap.h	76.92% (10/13)	80.00% (4/5)	76.92% (10/13)
base64.h	77.78% (7/9)	100.00% (2/2)	75.00% (6/8)
node_realm-inl.h	78.21% (61/78)	75.00% (15/20)	61.54% (24/39)
util.h	78.24% (133/170)	74.60% (47/63)	47.89% (91/190)
node_options.cc	80.68% (944/1170)	68.00% (17/25)	48.66% (163/335)
base_object-inl.h	84.18% (149/177)	84.62% (33/39)	59.70% (80/134)
node_mutex.h	84.95% (79/93)	81.82% (27/33)	62.71% (37/59)
async_wrap-inl.h	87.50% (28/32)	100.00% (8/8)	85.71% (12/14)
node_threadsafe_cow.h	88.89% (8/9)	88.89% (8/9)	88.89% (8/9)
callback_queue-inl.h	89.80% (44/49)	90.91% (10/11)	85.71% (18/21)
req_wrap-inl.h	90.48% (57/63)	85.71% (12/14)	60.00% (27/45)
string_bytes.h	90.91% (10/11)	100.00% (2/2)	80.00% (4/5)
cleanup_queue.cc	93.10% (27/29)	100.00% (5/5)	90.91% (10/11)
threadpoolwork-inl.h	94.00% (47/50)	75.00% (3/4)	74.77% (80/107)
node_metadata.cc	95.65% (66/69)	83.33% (5/6)	98.08% (51/52)
aliased_buffer.h	100.00% (21/21)	100.00% (6/6)	100.00% (6/6)
callback_queue.h	100.00% (1/1)	100.00% (1/1)	100.00% (1/1)
cleanup_queue.h	100.00% (2/2)	100.00% (2/2)	100.00% (2/2)
node_binding.h	100.00% (4/4)	100.00% (2/2)	100.00% (2/2)
node_bob.h	100.00% (1/1)	100.00% (1/1)	100.00% (1/1)
node_config.cc	100.00% (23/23)	100.00% (1/1)	100.00% (36/36)
node_constants.cc	100.00% (792/792)	100.00% (9/9)	91.41% (851/931)
node_symbols.cc	100.00% (11/11)	100.00% (1/1)	100.00% (17/17)
node_union_bytes.h	100.00% (10/10)	100.00% (7/7)	100.00% (7/7)
req_wrap.h	100.00% (2/2)	100.00% (2/2)	100.00% (2/2)
TOTALS	21.74% (15199/69918)	23.68% (1542/6513)	14.36% (13032/90741)

Figure 11: Code coverage of `node/src` folder at end of this engagement report here

Issues found

In this section we iterate through the issues found throughout the audit.

#	ID	Title	Severity	Fixed
1	ADA-2023-NODE-1	Read-based buffer overflows in WASI::	Informational	Yes
2	ADA-2023-NODE-2	Write-based buffer overflow in uvwasi__normalize_path	Informational	Yes
3	ADA-2023-NODE-3	Division by zero in hdrhistogram_c decoder	Informational	Yes
4	ADA-2023-NODE-4	Memory leaks in histogram_c decoders	Informational	Yes

Read-based buffer overflows in WASI::

Severity	Informational
id	ADA-2023-NODE-1
component	uvwasi

The modules in `WASI::` calls into various `uvwasi_path*` functions. Many of these functions will further call into `uvwasi__resolve_path` where `path` is a buffer which holds a size of `path_len` such as in this call. The problem in this case is that `uvwasi_resolve_path` does not enforce limitations in terms of reading only within the buffer, but will also read beyond the buffer in case the buffer is not NULL-terminated.

To show this we set up a fuzzer for `uvwasi__resolve_path` which found the following issue:

```
1     #0 0x4d0a1c in printf_common(void*, char const*, __va_list_tag*) /
         src/llvm-project/compiler-rt/lib/asan/../sanitizer_common/
         sanitizer_common_interceptors_format.inc:553:9
2     #1 0x4d2270 in __interceptor_snprintf /src/llvm-project/compiler-rt
         /lib/asan/../sanitizer_common/sanitizer_common_interceptors.inc
         :1736:1
3     #2 0x56cf56 in uvwasi__combine_paths /src/uvwasi/src/path_resolver.
         c:56:7
4     #3 0x56c3c4 in uvwasi__normalize_relative_path /src/uvwasi/src/
         path_resolver.c:284:9
5     #4 0x56c3c4 in uvwasi__resolve_path /src/uvwasi/src/path_resolver.c
         :442:11
6     #5 0x56b54d in LLVMFuzzerTestOneInput /src/uvwasi/.clusterfuzzlite/
         path_resolve_fuzzer.c:51:5
7     #6 0x43f323 in fuzzer::Fuzzer::ExecuteCallback(unsigned char const
         *, unsigned long) /src/llvm-project/compiler-rt/lib/fuzzer/
         FuzzerLoop.cpp:611:15
8     #7 0x4406d4 in fuzzer::Fuzzer::ReadAndExecuteSeedCorpora(std::
         __Fuzzer::vector<fuzzer::SizedFile, std::__Fuzzer::allocator<
         fuzzer::SizedFile> >&) /src/llvm-project/compiler-rt/lib/fuzzer/
         FuzzerLoop.cpp:804:3
9     #8 0x440ba9 in fuzzer::Fuzzer::Loop(std::__Fuzzer::vector<fuzzer::
         SizedFile, std::__Fuzzer::allocator<fuzzer::SizedFile> >&) /src/
         llvm-project/compiler-rt/lib/fuzzer/FuzzerLoop.cpp:857:3
10    #9 0x43020f in fuzzer::FuzzerDriver(int*, char***, int (*)(unsigned
          char const*, unsigned long)) /src/llvm-project/compiler-rt/lib/
         fuzzer/FuzzerDriver.cpp:912:6
11    #10 0x459862 in main /src/llvm-project/compiler-rt/lib/fuzzer/
         FuzzerMain.cpp:20:10
```

```
12      #11 0x7f9edf598082 in __libc_start_main (/lib/x86_64-linux-gnu/libc
            .so.6+0x24082) (BuildId:
            eebe5d5f4b608b8a53ec446b63981bba373ca0ca)
13      #12 0x420c4d in _start (/out/path_resolve_fuzzer+0x420c4d)
```

with a minor change in the fuzzer to also make the call `uvwasi__normalize_path(data, size, fd.normalized_path, BUFFER_SIZE)`; the following issue occurs

```
1  =================================================================
2  ==97003==ERROR: AddressSanitizer: heap-buffer-overflow on address 0
       x602000005dfa at pc 0x00000056c02f bp 0x7ffd077ee680 sp 0
       x7ffd077ee678
3  READ of size 1 at 0x602000005dfa thread T0
4  SCARINESS: 12 (1-byte-read-heap-buffer-overflow)
5       #0 0x56c02e in uvwasi__strchr_slash /src/uvwasi/src/path_resolver.c
            :27:9
6       #1 0x56c02e in uvwasi__normalize_path /src/uvwasi/src/path_resolver
            .c:89:12
7       #2 0x56b53f in LLVMFuzzerTestOneInput /src/uvwasi/.clusterfuzzlite/
            path_resolve_fuzzer.c:48:9
8       #3 0x43f323 in fuzzer::Fuzzer::ExecuteCallback(unsigned char const
            *, unsigned long) /src/llvm-project/compiler-rt/lib/fuzzer/
            FuzzerLoop.cpp:611:15
9       #4 0x43eb0a in fuzzer::Fuzzer::RunOne(unsigned char const*,
            unsigned long, bool, fuzzer::InputInfo*, bool, bool*) /src/llvm-
            project/compiler-rt/lib/fuzzer/FuzzerLoop.cpp:514:3
10      #5 0x4401d9 in fuzzer::Fuzzer::MutateAndTestOne() /src/llvm-project
            /compiler-rt/lib/fuzzer/FuzzerLoop.cpp:757:19
11      #6 0x440ea5 in fuzzer::Fuzzer::Loop(std::__Fuzzer::vector<fuzzer::
            SizedFile, std::__Fuzzer::allocator<fuzzer::SizedFile> >&) /src/
            llvm-project/compiler-rt/lib/fuzzer/FuzzerLoop.cpp:895:5
12      #7 0x43020f in fuzzer::FuzzerDriver(int*, char***, int (*)(unsigned
             char const*, unsigned long)) /src/llvm-project/compiler-rt/lib/
            fuzzer/FuzzerDriver.cpp:912:6
13      #8 0x459862 in main /src/llvm-project/compiler-rt/lib/fuzzer/
            FuzzerMain.cpp:20:10
14      #9 0x7feb7964f082 in __libc_start_main (/lib/x86_64-linux-gnu/libc.
            so.6+0x24082) (BuildId: eebe5d5f4b608b8a53ec446b63981bba373ca0ca
            )
15      #10 0x420c4d in _start (/out/path_resolve_fuzzer+0x420c4d)
```

The problem is that `uvwasi__resolve_path` and uvwasi__normalize_path assumes that `path` is NULL-terminated, but this is not always guaranteed to be the case. In particular, when a `size` of the buffer is provided as well to the API, it is counter-intuitive that the logic will read beyond the provided size. We recommend ensuring that these APIs do not read beyond the provided buffer length. We indicate this as informational as we did not find this affect NodeJS, but recommend either clearly indicating in the documentation that the string should be NULL-terminated.

The fuzzer used to find this:

```c
#include <stdint.h>
#include <stdlib.h>
#include <string.h>

#include "../src/fd_table.h"
#include "../src/path_resolver.h"
#include "../src/wasi_rights.h"
#include "uvwasi.h"

#define BUFFER_SIZE 128

char normalized_buffer[BUFFER_SIZE];
static uvwasi_t uvwasi;

int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
  uvwasi_errno_t err;
  struct uvwasi_fd_wrap_t fd;

  char *new_str = (char *)malloc(size + 1);
  if (new_str == NULL) {
    return 0;
  }
  memcpy(new_str, data, size);
  new_str[size] = '\0';

  memset(normalized_buffer, 0, BUFFER_SIZE);

  static uvwasi_options_t init_options;
  uvwasi_options_init(&init_options);
  uvwasi_init(&uvwasi, &init_options);

  fd.id = 3;
  fd.fd = 3;
  fd.path = new_str;
  fd.real_path = new_str;
  fd.normalized_path = normalized_buffer;
  fd.type = UVWASI_FILETYPE_DIRECTORY;
  fd.rights_base = UVWASI__RIGHTS_ALL;
  fd.rights_inheriting = UVWASI__RIGHTS_ALL;
  fd.preopen = 0;

  char *resolved = NULL;
  uvwasi__resolve_path(&uvwasi, &fd, data, size, &resolved, 0);
  if (resolved != NULL) {
    free(resolved);
  }

  uvwasi_destroy(&uvwasi);

  free(new_str);
```

```
51    return 0;
52  }
```

Write-based buffer overflow in uvwasi__normalize_path

Severity	Informational
id	ADA-2023-NODE-2
component	uvwasi

A heap write-based buffer overflow was found in `uvwasi/src/path_resolved.c: uvwasi__normalize_path`. The issue was found by the following fuzzer:

```
#include <stdint.h>
#include <stdlib.h>
#include <string.h>

#include "../src/path_resolver.h"

#define BUFFER_SIZE 128

char normalized_buffer[BUFFER_SIZE];

int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
  char *new_str = (char *)malloc(size + 1);
  if (new_str == NULL) {
    return 0;
  }
  memcpy(new_str, data, size);
  new_str[size] = '\0';

  memset(normalized_buffer, 0, BUFFER_SIZE);

  uvwasi__normalize_path(new_str, size, normalized_buffer, BUFFER_SIZE)
      ;

  free(new_str);
  return 0;
}
```

The fuzzer finds an issue with the following ASAN report:

```
=================================================================
==97003==ERROR: AddressSanitizer: global-buffer-overflow on address 0
    x00000101f8a0 at pc 0x00000056c043 bp 0x7ffc71cbc100 sp 0
    x7ffc71cbc0f8
WRITE of size 1 at 0x00000101f8a0 thread T0
    #0 0x56c042 in uvwasi__normalize_path /src/uvwasi/src/path_resolver
        .c
```

```
5       #1 0x56b533 in LLVMFuzzerTestOneInput /src/uvwasi/.clusterfuzzlite/
            path_resolve_fuzzer.c:48:9
6       #2 0x43f323 in fuzzer::Fuzzer::ExecuteCallback(unsigned char const
            *, unsigned long) /src/llvm-project/compiler-rt/lib/fuzzer/
            FuzzerLoop.cpp:611:15
7       #3 0x42aa82 in fuzzer::RunOneTest(fuzzer::Fuzzer*, char const*,
            unsigned long) /src/llvm-project/compiler-rt/lib/fuzzer/
            FuzzerDriver.cpp:324:6
8       #4 0x43032c in fuzzer::FuzzerDriver(int*, char***, int (*)(unsigned
             char const*, unsigned long)) /src/llvm-project/compiler-rt/lib/
            fuzzer/FuzzerDriver.cpp:860:9
9       #5 0x459862 in main /src/llvm-project/compiler-rt/lib/fuzzer/
            FuzzerMain.cpp:20:10
10      #6 0x7f7d5a029d8f in __libc_start_call_main csu/../sysdeps/nptl/
            libc_start_call_main.h:58:16
11      #7 0x7f7d5a029e3f in __libc_start_main csu/../csu/libc-start.c
            :392:3
12      #8 0x420c4d in _start (/tmp/oss-fuzz/build/out/uvwasi/
            path_resolve_fuzzer+0x420c4d)
```

The issue occurs on line 141 in `path_resolver.c`:

```
139       memcpy(ptr, cur, cur_len);
140       ptr += cur_len;
141       *ptr = '\0';
142     }
```

The problem is that `ptr += cur_len` may set `ptr` to point at `normalized_path + normalized_len` which causes an off-by-one issue when the `normalized_len` corresponds to the size of the `normalized_path` buffer.

The proposed fix is to check that `ptr` has not increased beyond the bounds of `normalized_path`:

```
139       memcpy(ptr, cur, cur_len);
140       ptr += cur_len;
141       if (ptr >= (normalized_path + normalized_len))
142         return UVWASI_ENOTCAPABLE;
143       *ptr = '\0';
144     }
```

It's important to highlight in this case that the function used within uvwasi always allocates an extra byte such as here and here. However, we consider it counter-intuitive that `uvwasi__normalize_path` reads beyond the specified length, this is made more counter-intuitive considering the tests of this function provides the size of the buffer and not 1 less than the size of the normalized buffer here.

Considering that the API is used correctly in the places it's called, we consider the best option to either change the tests to indicate the size shuold be +1 and also add documentation for the API highlight the buffer must must be of the size+1 of the size provided.

The fuzzer we used for this is:

```c
#include <stdint.h>
#include <stdlib.h>
#include <string.h>

#include "uvwasi.h"
#include "../src/fd_table.h"
#include "../src/path_resolver.h"
#include "../src/wasi_rights.h"

#define BUFFER_SIZE 128

char normalized_buffer[BUFFER_SIZE];
static uvwasi_t uvwasi;

int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
  uvwasi_errno_t err;
  struct uvwasi_fd_wrap_t fd;

  if (size < 10) {
    return 0;
  }

  char *new_str = (char *)malloc(size + 1);
  if (new_str == NULL) {
    return 0;
  }
  memcpy(new_str, data, size);
  new_str[size] = '\0';

  memset(normalized_buffer, 0, BUFFER_SIZE);

  static uvwasi_options_t init_options;
  uvwasi_options_init(&init_options);
  uvwasi_init(&uvwasi, &init_options);


  fd.id = 3;
  fd.fd = 3;
  fd.path = new_str;
  fd.real_path = new_str;
  fd.normalized_path = normalized_buffer;
  fd.type = UVWASI_FILETYPE_DIRECTORY;
  fd.rights_base = UVWASI__RIGHTS_ALL;
  fd.rights_inheriting = UVWASI__RIGHTS_ALL;
  fd.preopen = 0;


  err = uvwasi__normalize_path(new_str, size, fd.normalized_path,
      BUFFER_SIZE);
```

```
49    if (err == UVWASI_ESUCCESS) {
50      char* resolved = NULL;
51      uvwasi__resolve_path(&uvwasi, &fd, new_str, size, &resolved, 0);
52      if (resolved != NULL) {
53          free(resolved);
54      }
55    }
56
57    uvwasi_destroy(&uvwasi);
58
59    free(new_str);
60    return 0;
61  }
```

Division by zero in hdrhistogram_c decoder

Severity	Informational
id	ADA-2023-NODE-3
component	hdrhistogram_c

The `hdrhistogram_c` dependency was found to have a division by zero issue when decoding histograms. This issue was found by the ClusterFuzzLite integration available here.

The problem is in the following lines where `word_size` is read from the log file, but is not being checked for value before being used in a division operation.

```
501     word_size = word_size_from_cookie(be32toh(encoding_flyweight.cookie
            ));
502     counts_limit = be32toh(encoding_flyweight.payload_len) / word_size;
503     lowest_discernible_value = be64toh(encoding_flyweight.
            lowest_discernible_value);
504     highest_trackable_value = be64toh(encoding_flyweight.
            highest_trackable_value);
505     significant_figures = be32toh(encoding_flyweight.
            significant_figures);
```

In the event `word_size` is `0` a division-by-zero operation will happen on line 502.

The proposed, and accepted, fix is available here and adds a check on the `word_size`:

```
501     word_size = word_size_from_cookie(be32toh(encoding_flyweight.cookie
            ));
502     if (word_size == 0)
503     {
504         FAIL_AND_CLEANUP(cleanup, result, HDR_INVALID_WORD_SIZE);
505     }
506     counts_limit = be32toh(encoding_flyweight.payload_len) / word_size;
507     lowest_discernible_value = be64toh(encoding_flyweight.
            lowest_discernible_value);
508     highest_trackable_value = be64toh(encoding_flyweight.
            highest_trackable_value);
509     significant_figures = be32toh(encoding_flyweight.
            significant_figures);
```

We have set this as informational severity because this specific part of the histogram dependeny is determined not to be reachable of by Node.js.

Memory leaks in histogram_c decoders

Severity	Informational
id	ADA-2023-NODE-4
component	hdrhistogram_c

The `hdrhistogram_c` dependency was found to have three memory leaks when decoding histograms. This issue was found by the ClusterFuzzLite integration available here.

The problem is that `hdr_decode_compressed_v0`, `hdr_decode_compressed_v1` and `hdr_decode_compressed_v2` relies on `hdr_init` to dynamically allocate a `hdr_histogram` struct. However, in the event the decoding fails then this is cleared up using `hdr_free` diretly on the dynamically allocated memory. The problem is that the allocated struct itself contains a pointer to some dynamically allocated memory, which will be left dangling. This is specifically the `counts` array allocated here and assigned to a member of the struct here.

The proposed fix is to use the dedicated `hdr_close` function instead of `hdr_free`:

```
1 void hdr_close(struct hdr_histogram* h)
2 {
3   if (h) {
4     hdr_free(h->counts);
5     hdr_free(h);
6   }
7 }
```

The proposed, and accepted, fix is in this PR.