ADALOGICS

GNU Libmicrohttpd2 Audit

GNU Libmicrohttpd2 Security Audit

David Korczynski, Adam Korczynski, Arthur Chan

15-09-2025

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

About Open Source Technology Improvement Fund

The Open Source Technology Improvement Fund (OSTIF) is dedicated to resourcing and managing security engagements for open source software projects through partnerships with corporate, government, and non-profit donors. We bridge the gap between resources and security outcomes, while supporting and championing the open source community whose efforts underpin our digital landscape.

Over the past ten years, OSTIF has been responsible for the discovery of over 800 vulnerabilities, (121 of those being Critical/High), over 13,000 hours of security work, and millions of dollars raised for open source security. Maximizing output and security outcomes while minimizing labor and cost for projects and funders has resulted in partnerships with multi-billion dollar companies, top open source foundations, government organizations, and respected individuals in the space. Most importantly, we've helped over 120 projects and counting improve their security posture.

Our directive is to support and enrich the open source community through providing public-facing security audits, educational resources, meetups, tooling, and advice. OSTIF's experience positions us to be able to share knowledge of auditing with maintainers, developers, foundations, and the community to further secure our infrastructure in a sustainable manner.

We are a small team working out of Chicago, Illinois. Our website is ostif.org. You can follow us on social media to keep up to date on audits, conferences, meetups, and opportunities with OSTIF, or feel free to reach out directly at contactus@ostif.org or our Github.

Derek Zimmer, Executive Director
Amir Montazery, Managing Director
Helen Woeste, Communications and Community Manager
Tom Welter, Project Manager

Contents

1 Executive Summary

In this report we present the results of a security audit of GNU Libmicrohttpd2. The three main objectives of the audit was to develop a threat model, manually audit the code and establish a continuous fuzzing suite that runs by way of OSS-Fuzz. To this end, this report will summarise our efforts and how we:

1. Laid out a threat model to classify the attack surface, which was used for the remainder of the security audit.
2. Manually reviewed the source code of GNU Libmicrohttpd2 to identify potential logical and security flaws.
3. Developed an extensive set of fuzzing harnesses for GNU Libmicrohttpd2 and created a continuous fuzzing integration with OSS-Fuzz. These fuzzing harnesses are now running on a daily basis continuously.
4. Ran several static analysis tools against the project and went through the reported issues.

The fuzzing harnesses uncovered 2 issues during the engagement and the fuzzer have now run continuously for several weeks by way of OSS-Fuzz. The manual audit uncovered 5 further issues and the static analysis tools reported non-security relevant findings. As such, a total of 7 issues were found and reported, and all 7 issues have been fixed upstream.

In general we consider libmicrohttpd2 to be a well-written software package. However, we suggest is space for improving security practices in areas such as extending fuzzing and using automated static code analysers, and improving documentation/communication on the security context of the project to the users.

Finally, we would like to thank the Sovereign Tech Agency for funding our efforts and we would like to thank OSTIF for facilitating this audit.

2 Threat Model

In this section we will go through the threat model we developed for the security audit. As `libmicrohttpd2` is a library, we focused the threat model on capturing the public APIs of the library, how they may accept untrusted input, as well as how untrusted data may flow into the private (declared with **`static`**) functions. Furthermore, a key purpose of the modelling was to identify the attack surface of `libmicrohttpd2` so that this can be used to guide the remainder of the tasks of the audit. For example, for fuzzing development it is necessary to know the attack surface of the application, to make sure untrusted input is covered by the fuzzers and in order to fuzz and audit deeper parts of the code, it is necessary to have an understanding of how the data at these parts can be controlled by potential attackers so as to avoid over-approximating the security analysis.

2.1 Overview

`libmicrohttpd2` is a lightweight C library that embeds a HTTP or HTTPS daemon into host applications. The core logic resides in `src/mhd2/`, which contains the HTTP daemon, connection state machine, memory pools, parsers and cryptographic helpers.

Applications use public APIs exposed by the library to start a daemon, register callbacks, and read parsed request data by helpers. There are also multiple standalone tools in `src/tools` that provide supporting features independent the live HTTP daemon.

2.2 Grouping the APIs

In order to lay out the attack surface we looked at the APIs exposed by the library and how they could enable execution in the rest of the library. We grouped the functions of `libmicrohttpd2` into three groups that represent how clients would use `libmicrohttpd2`:

Group A, are the functions most exposed to untrusted data

- mhd_recv(…),
- mhd_connection_tls_check(…)
- mhd_stream_parse_request_line(…),
- mhd_str_pct_decode_lenient_n(…),
- mhd_parse_get_args(…)
- mhd_stream_get_request_headers(…),
- parse_content_length(…),
- mhd_stream_process_request_body(…),
- mhd_post_parse(…)

Group B, are helper functions to retrieve internal parsed state and data, often relying on data structures created by group A.

- MHD_request_get_value(…),
- MHD_request_get_values_cb(…)
- MHD_request_get_info_fixed_sz(…),
- MHD_request_get_info_dynamic_sz(…)
- mhd_request_get_auth_header_value(…)

Group C, are helper functions for data processing and response generation

- find_and_parse_auth_basic(…),
- mhd_request_get_auth_basic_creds(…)

- check_argument_match(…),
- mhd_str_pct_decode_lenient_n(…)
- mhd_stream_add_reply_headers(…),
- mhd_stream_build_reply_iov(…),
- mhd_send_data(…)

2.3 Attack surface

All untrusted bytes flow through Group A, become structured state in memory pools, and are then exposed via Group B to application code or processed in Group C for authentication, server-side process and response construction.

- Group A attack surface:
  - Raw bytes handling (`mhd_recv(...)`, `mhd_connection_tls_check(...)`)
  - Request parsing (`mhd_stream_parse_request_line(...)`)
  - Percent decoding and query parsing (`mhd_str_pct_decode_lenient_n(...)`, `mhd_parse_get_args(...)`)
  - Header parsing (`mhd_stream_get_request_headers(...)`, `parse_content_length(...)`)
  - Chunked buffer processing (`mhd_stream_process_request_body(...)`)
  - Multipart form parsing (`mhd_post_parse(...)`)
- Group B attack surface:
  - Application access to parsed attacker data as read-only pointers and getters
  - Lifetime memory pool management (`MHD_request_get_value(...)`, `MHD_request_get_values_cb(...)`, `mhd_request_get_auth_header_value(...)`)
- Group C attack surface:
  - Authentication parsing and validation (`find_and_parse_auth_basic(...)`, `check_argument_match(...)`)
  - Construction of headers and I/O buffers (`mhd_stream_add_reply_headers(...)`, `mhd_stream_build_reply_iov(...)`)
  - Output path integrity and response creation (`mhd_send_data(...)`)

2.4 Boundaries and functions

2.4.1 Group A: Network ingress to Group 1 (internal parsing)

Group A covers the full path from raw network bytes to request state in local thread and memory pool. Everything here is declared MHD_Internal or static and driven by the daemon's internal event loop after an application starts the server via public `MHD_daemon_*` APIs.

Data flow

- `mhd_recv(...)` (MHD_Internal) Reads bytes from TCP/TLS into `c->read_buffer`. Short/partial reads and errors drive state transitions.
- `mhd_conn_tls_check(...)` (MHD_Internal) Advances TLS handshake and selects the correct recv path so plaintext is never misinterpreted.
- `MHD_daemon_add_connection(...)` Creates `MHD_Connection` and initialises the per-request memory pool for all subsequent parsed bytes. Then store the `MHD_Connection` in the `MHD_Daemon`.

- `mhd_pool_create(...)`, `mhd_pool_destroy(...)` (static/MHD_Internal) Managed the memory pool for local storage; correct teardown on every path bounds lifetime and prevents leaks.
- `mhd_stream_get_request_line(...)` (MHD_Internal) Parses method, target, and version from the read buffer; rejects malformed CRLF and spacing.
- `mhd_str_pct_decode_lenient_n(...)` (static/MHD_Internal) In place percent decoding used by the request handlers and helpers.
- `mhd_parse_get_args(...)` (static/MHD_Internal) Splits query arguments at &/=, decodes in place, and stores key/value pairs in pool-backed structures.
- `mhd_stream_get_request_headers(...)` (MHD_Internal) Tokenises headers, parses and inserts header fields into internal lists. The following is some of the supporting functions for these header parsing process.
  - `mhd_str_equal_caseless_n_st(...)` (static/MHD_Internal)
  - `parse_cookie_header(...)` (static/MHD_Internal)
- `mhd_stream_process_request_body(...)` (MHD_Internal) Use streaming approach to parse and process the body of a request object. The following is some of the checking and handling functions for the stream process enforcement.
  - `handle_req_chunk_size_line_no_space(...)` (static)
  - `mhd_stream_check_and_grow_read_buffer_space(...)` (static)
- `mhd_stream_post_parse(...)` (MHD_Internal) Parses URL-encoded and multipart/form-data and stored in the memory pool.
- `mhd_upgrade_try_start_upgrading(...)` (MHD_Internal) Switches the connection to an upgraded tunnel while keeping lifecycle wrapper consistent.

These internal functions are invoked by the daemon's internal event loop (triggered by public `MHD_daemon_*` start APIs). Applications and logic from the developers do not call them directly.

2.4.2 Boundary B: Group 1 to Group 2 (Helpers to retrieve internal parsed state and data)

Group B exposes parsed data and state as read-only pointers into the per-request memory pool. The lifetime of this data is bound by the lifetime of the memory pool which are separated for each request and are destroyed (`mhd_pool_destroy(...)`) after the responses are created and sent.

Data flow

- `MHD_request_get_value(...)` (public) Returns a pointer to a header, argument, or cookie value that is parsed and stored in the local per-request memory pool.
- `MHD_request_get_values_cb(...)` (public) Iterates values of a given kind via user callback; exposes each pool-backed slice.
- `MHD_request_get_info_fixed_sz(...)`, `MHD_request_get_info_dynamic_sz(...)` (public) Provides request/connection metadata (sizes, versions) without re-parsing.
- `mhd_request_get_auth_header_value(...)` (public) Returns a slice of the `Authorization` header for downstream auth helpers.

The application's request callback calls these APIs to read normalised values and metadata and pointers must not be retained after the request finishes.

2.4.3 Boundary C: Group 2 to Group 3 (helpers, authentication, response creation)

Group C provides helpers for the developer callback handling functions to allow performing server logic, including processing of data, authentication or response creation. These functions also support response creation and sending back to the client.

Data flow

- `mhd_request_get_auth_basic_creds(...)` (public) Public entrypoint that decodes basic credentials by the internal helper below.
  - `find_and_parse_auth_basic(...)` (MHD_Internal) Base64 decoding of attacker data into the pool and returns memory pool pointers.
- `check_argument_match(...)` (MHD_Internal) Digest helper that reuses Boundary A parsing (`mhd_parse_get_args(...)`) to ensure URI and query equivalence throughout the request handling process.
- `mhd_stream_build_header_response(...)` (MHD_Internal) Create and prepare standard response header.
- Crypto helpers: `mhd_md5_update(...)`, `mhd_sha256_update(...)`, `mhd_sha512_256_update(...)` (static/MHD_Internal) Helper functions to Process attacker-influenced bytes for authentication and other cryptographic purposes.
- `mhd_send_data(...)` (MHD_Internal) Opposite to `mhd_recv(...)` that write the prepared response to the network and return to the client.

The application's callback reads values (Boundary B), and the daemon's internal pipeline (Boundary C) provides helper functions to perform decoding, authentication, verification, response header and body creation, and the final network write.

2.4.4 Summarising the groups

Boundary A takes raw client bytes and performs the only parsing step, storing everything in a per-request memory pool with a strict lifetime. Boundary B exposes shallow, read-only pointers into that per-request memory pool via public APIs, avoiding re-parsing and copying the raw request and connection data. Boundary C consumes those pointers in helpers to validate authentication, create responses, and send bytes back to the client. The Primary risks remain in raw request parsing, including multiform or chunked request parsing, handling of parsed but not sanitised data in the callback functions and the leaking of information from the response creation.

2.5 Standalone tools

These run outside the daemon and do not expose new network attack surface, but can help generate and validate performance or configurations.

- `src/tools/mhdtl_get_cpu_count.c` Standalone utility with `main(...)` to report CPU count for monitoring of daemon and container resource performance. No network input. No impact on live threat model.
- `src/tools/perf_replies.c` Standalone utility for implementation of HTTP server optimised for fast replies based on MHD2 daemon. More like supporting tools and wrappers to use the MHD2 daemon. No new attack surface introduced.
- `src/include/options-generator.c` Interatice options and configurations generator for the developers on daemon startup. Developer-driven inputs only. No runtime exposure or live threat model impact.

The analysed standalone tools do not alter the live attack surface of the MHD2 daemon and web service.

3 Manual audit

We carried out a manual code review for the three groups of functions from the threat model and the code that they reach. The focus was on the source code within the `src/mhd2` directory, which contains the project's main logic. The audit involved tracing the data flow from public APIs into static functions or those marked as `MHD_Internal`, in order to identify gaps where input validation and sanitisation checks might be insufficient. We also ran four static analysis tools on the target codebase, **CodeQL**, **Infer**, **Cppcheck**, and **Semgrep**. The tools did not report any security relevant issues but had several comments on style. The manual audit complements our fuzzing efforts, discussed in the next section, by concentrating on areas that are difficult for those fuzzing harnesses to cover or where logic flaws are less likely to be triggered by fuzzer-generated input.

In total, five issues were identified during the manual audit:

3.1 Issues found by manual audit

#	ID	Title	Severity	Status	GNUnet MantisBT issue#
1	ADA-GNU-MHD2-1	Buggy conditional check found in `mhd_recv.c`	Low	Fixed	10300
2	ADA-GNU-MHD2-2	Possible Stack Buffer Overflow in libmicrohttpd2 `daemon_add_conn.c`	Low	Fixed	10304
3	ADA-GNU-MHD2-3	A duplicated logic bug discovered in `mhd_str.c`	Informational	Fixed	10326
4	ADA-GNU-MHD2-4	A logic bug in algorithm enum parsing found in `auth_digest.c`	Low	Fixed	10328
5	ADA-GNU-MHD2-5	A conditional check bug in auth scheme parsing found in request_`auth_get.c`	Informational	Fixed	10354

4 Fuzzing development and OSS-Fuzz integration

We developed an extensive fuzzing suite for `libmicrohttpd2` and set up a continuous fuzzing suite by way of OSS-Fuzz. Each fuzzer targets a specific module or set of functions within the project, focusing on exercising logic paths, parsing routines, memory handling, and cryptographic operations. Together, they provide broad coverage of the library's core functionality that is reachable from public functions.

Fuzzers	Description	Source code
fuzz_connection	Targets the connection and stream-processing layer. Random request bodies and states are injected into `MHD_Connection` objects, fuzzing functions such as header parsing, body streaming, buffer management, and post-processing logic.	fuzz_connection.cpp
fuzz_daemon_connection	Focuses on the interaction between daemon and connection objects, including TLS checks, request callbacks, application-level parsing, and socket state toggling (`nodelay`, `cork`).	fuzz_daemon_connection.cpp
fuzz_daemon	Exercises full daemon lifecycle management. It fuzzes daemon creation, option configuration, port binding, entropy injection, and the processing of multiple randomly generated HTTP requests.	fuzz_daemon.cpp
fuzz_mhd2	Provides broad coverage of the high level MHD API. It fuzzes digest authentication calculation, response creation and configuration, daemon lifecycle, and information queries.	fuzz_mhd2.cpp
fuzz_response	Targets response object creation and manipulation. It fuzzes all constructors (`empty`, buffer-based, `iovec`, `fd`, `pipe`), header setting, option toggling, and authentication challenge insertion.	fuzz_response.cpp
fuzz_str	Exercises string utility functions. It fuzzes caseless comparisons, token matching, quoting/unquoting, Base64 and hex conversions, percent-decoding, and integer/string transformations.	fuzz_str.cpp
fuzz_crypto_int	Focuses on internal hashing primitives (`MD5`, `SHA-256`, `SHA-512/256`). It fuzzes multiple update/finish flows, misaligned buffers, re-initialisation, and chained digest feeding.	fuzz_crypto_int.cpp
fuzz_crypto_ext	Similar to `fuzz_crypto_int`, but for external hashing API wrappers. It fuzzes multiple contexts in parallel with varied chunking patterns, re-initialisation, and digest chaining.	fuzz_crypto_ext.cpp
fuzz_libinfo	Targets the library information query APIs (`MHD_lib_get_info_fixed_sz`, `MHD_lib_get_info_dynamic_sz`) with random IDs and raw data buffers.	fuzz_libinfo.cpp

4.1 Fuzzer coverage

These fuzzers collectively provide broad coverage of GNU Libmicrohttpd2. The harnesses target critical internal logic such as connection management, daemon lifecycle, cryptographic primitives, and string utilities, areas where subtle memory handling or logic bugs are more likely to arise.

From the code coverage report the 13th September 2025 accessible here 35.26% of the lines in the `src/mhd2` are covered. The following image shows a subset of the files covered:

mhd_mono_clock.c	40.54% (45/111)	100.00% (2/2)	41.67% (25/60)
daemon_funcs.c	44.90% (66/147)	55.56% (5/9)	38.22% (60/157)
mhd_bithelpers.h	45.16% (14/31)	50.00% (2/4)	35.14% (13/37)
response_destroy.c	65.12% (28/43)	75.00% (3/4)	37.78% (17/45)
response_set_options.c	65.82% (52/79)	100.00% (1/1)	62.30% (38/61)
daemon_create.c	66.67% (18/27)	100.00% (1/1)	68.00% (17/25)
mempool_funcs.c	70.30% (213/303)	80.00% (8/10)	75.09% (208/277)
post_parser_funcs.c	74.53% (1709/2293)	96.43% (27/28)	73.26% (989/1350)
mhd_lib_init.c	77.17% (71/92)	81.82% (9/11)	76.98% (97/126)
respond_with_error.c	77.97% (46/59)	100.00% (1/1)	81.40% (35/43)
mhd_str.c	79.52% (1402/1763)	80.56% (29/36)	63.85% (567/888)
sha256_ext.c	83.33% (20/24)	100.00% (4/4)	81.82% (18/22)
response_add_header.c	83.56% (61/73)	100.00% (5/5)	79.67% (98/123)
md5_ext.c	84.00% (21/25)	100.00% (4/4)	81.82% (18/22)
response_from.c	89.24% (199/223)	90.00% (9/10)	85.57% (166/194)
response_auth_basic.c	92.38% (97/105)	100.00% (2/2)	85.71% (84/98)
response_auth_digest.c	93.25% (304/326)	100.00% (4/4)	90.04% (217/241)
sha256_int.c	95.67% (265/277)	100.00% (4/4)	97.78% (923/944)
sha512_256_int.c	95.81% (297/310)	100.00% (4/4)	98.43% (1126/1144)
md5_int.c	96.58% (226/234)	100.00% (4/4)	98.20% (490/499)
lib_get_info.c	98.30% (231/235)	100.00% (2/2)	94.48% (154/163)
request_funcs.c	100.00% (18/18)	100.00% (2/2)	100.00% (29/29)
TOTALS	35.26% (7274/20630)	43.61% (198/454)	39.16% (6806/17381)

Figure 1: Fuzzers code coverage as reported by OSS-Fuzz.

The coverage report is accesible by way of Fuzz Introspector or by directly adjusting the date in the following link: https://storage.googleapis.com/oss-fuzz-coverage/libmicrohttpd2/reports/20250913/linux/src/mhd2/src/mhd2/report.html.

4.2 OSS-Fuzz integration

The source code of the fuzzers are available in the OSS-Fuzz repository under the libmicrohttpd2 project here. The OSS-Fuzz integration consists of the source code of the fuzzers themselves, as well as two files: `Dockerfile` and a `build.sh`. These two files control the building of the fuzzers within the OSS-Fuzz infrastructure.

The Dockerfile is as follows:

```
1  FROM gcr.io/oss-fuzz-base/base-builder
2  RUN apt-get update && \
3      apt-get install -y autoconf automake libtool pkg-config texinfo \
4          gnutls-dev gnutls-bin
5  RUN git clone https://git.gnunet.org/libmicrohttpd2.git mhd2
```

```
WORKDIR mhd2
COPY build.sh *.cpp *.h $SRC/
COPY *.options *.dict $SRC/
```

The `build.sh` is as follows:

```
BINARY=$SRC/mhd2/src/mhd2/.libs/libmicrohttpd2.a

# Build libmicrohttpd
./autogen.sh
./configure --enable-dauth --enable-md5 --enable-sha256 --enable-sha512-256 \
  --enable-bauth --enable-upgrade --enable-https --enable-messages
ASAN_OPTIONS=detect_leaks=0 make -j$(nproc)
make install

# Compile fuzzer
FUZZERS="fuzz_response fuzz_daemon fuzz_mhd2 fuzz_str fuzz_crypto_int fuzz_libinfo
    fuzz_connection fuzz_daemon_connection"

for fuzzer in $FUZZERS; do
  extra_src=""
  case "$fuzzer" in
    fuzz_response|fuzz_daemon)
      extra_src="$SRC/mhd_helper.cpp"
      ;;
  esac
  case "$fuzzer" in
    fuzz_connection|fuzz_daemon_connection)
      extra_src="$SRC/connection_helper.cpp"
      ;;
  esac

  $CXX $CXXFLAGS -DHAVE_CONFIG_H "$SRC/$fuzzer.cpp" $extra_src \
    -Wno-unused-parameter -Wno-unused-value -pthread \
    -I"$SRC" -I"$SRC/mhd2/src/mhd2" -I"$SRC/mhd2/src/include" \
    -I"$SRC/mhd2/src/incl_priv" -I"$SRC/mhd2/src/incl_priv/config" \
    $LIB_FUZZING_ENGINE "$BINARY" -lgnutls -o "$OUT/$fuzzer"
done

# Rebuild the binary for external crypto
./autogen.sh
./configure --enable-md5=tlslib --enable-sha256=tlslib --enable-sha512-256=builtin
make clean
make -j$(nproc)
make install

$CXX $CXXFLAGS $SRC/fuzz_crypto_ext.cpp -DHAVE_CONFIG_H \
  -Wno-unused-parameter -Wno-unused-value -I$SRC/mhd2/src/mhd2 \
  -I$SRC/mhd2/src/include -I$SRC/mhd2/src/incl_priv \
  -I$SRC/mhd2/src/incl_priv/config $LIB_FUZZING_ENGINE $BINARY \
  -lgnutls -o $OUT/fuzz_crypto_ext

cp $SRC/default.options $OUT/fuzz_daemon.options
cp $SRC/*.dict $OUT/
```

The fuzzers are build within the OSS-Fuzz base-builder image, and rely on the environment variables `CC`, `CXX`, `CFLAGS`, `CXXFLAGS` to control compilation with the respective fuzzing engines of OSS-Fuzz. This is why the `build.sh` script uses these environment variables to compile the code.

Furthermore, the project has a corresponding Yaml file `project.yaml` here which holds meta information about the project. This includes emails that are affiliated with the project, and will recieve notifications on when issues are found. Any maintainers needing to get notifications on security issues should add their emails to this file.

https://oss-fuzz.com provides more information about the OSS-Fuzz status of the project. In order to access this information you need to have your email listed in the `project.yaml` file as well. Furthermore, issues will be shown at this link https://issues.oss-fuzz.com/issues?q=project%3Dlibmicrohttpd2 as well, and issues follow Google's security policy in terms of embargo on details. Anyone with their emails listed in `project.yaml` will be able to access the private issue information before the security disclosure deadline.

4.2.1 Navigating the OSS-Fuzz project

In this section we provide some details on how build and run the fuzzing harnesses of OSS-Fuzz.

4.2.1.1 Build the libmicrohttpd2 fuzzers In order to build the fuzzers that are run by OSS-Fuzz, the below commands can be used:

```
git clone https://github.com/google/oss-fuzz
cd oss-fuzz
python3 infra/helper.py build_fuzzers libmicrohttpd2
...
ls -la build/out/libmicrohttpd2
```

4.2.1.2 Running a fuzzer Once the fuzzers are built following the steps above, the following command can be used to run the `fuzz_str` harness. The command is to be run from the OSS-Fuzz root dir, from where the building also took place.

```
# Run the fuzz_str fuzzer
python3 infra/helper.py run_fuzzer libmicrohttpd2 fuzz_str
```

4.2.1.3 Extracting code coverage of fuzzers A convenient feature of the OSS-Fuzz infrastructure is that it can be used to extract code coverage easily of the fuzzing harnesses. The below commands shot the complete way to do this.

```
# Build the fuzzers
git clone https://github.com/google/oss-fuzz
cd oss-fuzz
python3 infra/helper.py introspectosr --coverage-only --seconds 10 libmicrohttpd2
...
# The fuzzing harnesses will be running for a bit now.
# Show the coverage report
python3 -m http.server 8013 --directory build/out/libmicrohttpd2/report
```

4.3 Issues found by fuzzers

Two issues have been reported by the `libmicrohttpd2` fuzzers so far, and are listed at the table below.

#	ID	Title	Severity	Status	GNU MantisBT issue#
6	ADA-GNU-MHD2-6	Possible Heap Buffer Overflow in libmicrohttpd2 `response_auth_digest.c`	High	Fixed	10253
7	ADA-GNU-MHD2-7	Possible memory-leaking bug discovered in `mempool_funcs.c`	Moderate	Fixed	10296

5 Issues found

Here we present the issues that we identified during both the manual and fuzzing audit. A total of seven issues were found and reported, and all of the issues have been resolved. The below table provides an overview of the issues and following this we go into detail with each of the seven issues.

#	ID	Title	Severity	Status	GNUnet MantisBT issue#
1	ADA-GNU-MHD2-1	Buggy conditional check found in `mhd_recv.c`	Low	Fixed	10300
2	ADA-GNU-MHD2-2	Possible Stack Buffer Overflow in libmicrohttpd2 `daemon_add_conn.c`	Low	Fixed	10304
3	ADA-GNU-MHD2-3	A duplicated logic bug discovered in `mhd_str.c`	Informational	Fixed	10326
4	ADA-GNU-MHD2-4	A logic bug in algorithm enum parsing found in `auth_digest.c`	Low	Fixed	10328
5	ADA-GNU-MHD2-5	A conditional check bug in auth scheme parsing found in request_`auth_get.c`	Informational	Fixed	10354
6	ADA-GNU-MHD2-6	Possible Heap Buffer Overflow in libmicrohttpd2 `response_auth_digest.c`	High	Fixed	10253
7	ADA-GNU-MHD2-7	Possible memory-leaking bug discovered in `mempool_funcs.c`	Moderate	Fixed	10296

5.1 [Libmicrohttpd2] Buggy conditional check in `mhd_recv_tls`

Severity	Low
Status	Fixed
id	ADA-GNU-MHD2-1
Component	mhd_recv.c

5.1.1 Problem

A faulty conditional branch check has been identified in the `mhd_recv_tls(...)` function.

The variable `res` is declared as an `enum mhd_SocketError` and holds the return value of `mhd_tls_conn_recv(...)`. This value is expected to be either `mhd_SOCKET_ERR_NO_ERROR` or a specific socket error code. When `res` equals `mhd_SOCKET_ERR_NO_ERROR`, a subsequent check compares `res` (which will always be `0`) against `buf_size` (which has been asserted to be non-zero).

As a result, the condition `if (res == buf_size)` can never evaluate as true, meaning that the block responsible for setting `c->tls_has_data_in` is never executed.

5.1.2 Consequence

Since `c->tls_has_data_in` is never set, the daemon and connection handler always assume there are no pre-decrypted bytes available in the TLS buffer. This prevents code paths relying on the flag from running, causing buffered data to be ignored and forcing the system to wait unnecessarily for further socket events.

For example, in `conn_data_process.c`, the following condition always evaluates with the `mhd_C_HAS_TLS_DATA_IN` part as false:

```
1  use_recv = (0 != (mhd_SOCKET_NET_STATE_RECV_READY
2                    & (c->sk.ready | mhd_C_HAS_TLS_DATA_IN (c))));
```

This results in unnecessary delays, possible stalls, and reduced throughput.

5.1.3 Mitigation

The comparison should not be made against the error code but against the number of bytes received. The `received` parameter, of type `size_t *`, is specifically intended to capture the number of bytes read.

Suggested fix:

```
1  if (*received == buf_size)
```

This ensures that `c->tls_has_data_in` is correctly set whenever the receive buffer is completely filled and additional TLS data remains available.

5.1.4 Reference

- `mhd_recv.c` lines 95–122
- `mhd_SocketError` enum definition

5.1.5 Reported Issue

- GNUnet MantisBT #10300

5.2 [Libmicrohttpd2] Stack buffer overflow in `MHD_daemon_add_connection`

Severity	Low
Status	Fixed
id	ADA-GNU-MHD2-2
Component	daemon_add_conn.c

5.2.1 Problem

A stack buffer overflow has been identified in the public function `MHD_daemon_add_connection` in `src/mhd2/daemon_add_conn.c`. This API is explicitly documented as callable by application developers for advanced scenarios such as NAT traversal or proxying.

At line 792, the following vulnerable code is executed:

```
1  struct sockaddr_storage addrstorage;
2  /* ... */
3  if (0 < addrlen)
4      memcpy(&addrstorage, addr, addrlen);
```

Although negative or zero values for `addrlen` are filtered out, no check exists to ensure that `addrlen` does not exceed `sizeof(struct sockaddr_storage)`. If `addrlen` is larger, `memcpy` writes beyond the allocated stack buffer, resulting in memory corruption.

This issue is low as if the applications calls this API with a larger value, then it violates OS/Berkeley socket contracts.

5.2.2 Consequence

While typical use through kernel-provided `accept()` calls ensures a valid `addrlen`, the function is a **public API** and may be invoked directly by developers. If the parameter is derived from untrusted input, passing a crafted `addrlen` causes **stack corruption**. This may lead to process crashes or, in the worst case, remote code execution.

The vulnerability is especially concerning because the documentation explicitly describes this API as suitable for direct use by applications. As such, it must fail safely when invalid arguments are provided.

5.2.3 Proof of Concept (`poc.cpp`)

The following PoC deliberately triggers the bug by calling `MHD_daemon_add_connection()` with an `addrlen` set to `sizeof(struct sockaddr_storage)+ 1` with minimal configurations. The source buffer is correctly allocated to avoid read overflows, ensuring that the overflow occurs entirely in the destination stack buffer inside the function. No valid socket setup or HTTP client data is needed to reproduce the bug, this confirms the vulnerability is because of the missing bound check of `addrlen` in the function's implementation.

```
1  #include <microhttpd2.h>
2  #include <sys/socket.h>
3  #include <stdlib.h>
4  #include <string.h>
5  #include <stdint.h>
```

```
6
7  static const struct MHD_Action* dummy_handler(void* cls, struct MHD_Request* req, const
       struct MHD_String* path, enum MHD_HTTP_Method method, uint_fast64_t upload_size) {
8    return MHD_action_abort_request(req);
9  }
10
11 int main(void) {
12   struct MHD_Daemon* daemon = MHD_daemon_create(&dummy_handler, NULL);
13   MHD_daemon_start(daemon);
14
15   int server_socket = socket(AF_INET, SOCK_STREAM, 0);
16   int accepted_socket = accept(server_socket, NULL, NULL);
17
18   size_t addrlen = sizeof(struct sockaddr_storage) + 1;
19   unsigned char* buf = (unsigned char*) malloc(addrlen + 1);
20   memset(buf, 0x41, addrlen + 1);
21
22   MHD_daemon_add_connection(daemon, accepted_socket, addrlen, (const struct sockaddr*)
         buf, NULL);
23
24   return 0;
25 }
```

5.2.4 Build and run the poc

```
1  # Clone project
2  git clone https://git.gnunet.org/libmicrohttpd2.git mhd2
3  cd mhd2
4
5  # Build project
6  ./autogen.sh
7  ./configure --enable-dauth --enable-md5 --enable-sha256 --enable-sha512-256 \
8    --enable-bauth --enable-upgrade --enable-https --enable-messages
9  ASAN_OPTIONS=detect_leaks=0 make -j$(nproc)
10 make install
11 BINARY=./src/mhd2/.libs/libmicrohttpd2.a
12
13 # Build poc
14 clang++ -fsanitize=address -stdlib=libc++ poc.cpp -I/src/mhd2/src/include ./src/mhd2/.
       libs/libmicrohttpd2.a -lgnutls -o poc
15
16 # Run poc
17 ./poc
```

5.2.5 Crash Output

Running the PoC with AddressSanitizer produces:

```
1  =================================================================
2  ==149958==ERROR: AddressSanitizer: stack-buffer-overflow on address 0x7fed29b001a0 at
       pc 0x56211d190594 bp 0x7ffcf6132750 sp 0x7ffcf6131f10
3  WRITE of size 129 at 0x7fed29b001a0 thread T0
4      #0 0x56211d190593 in __asan_memcpy /src/llvm-project/compiler-rt/lib/asan/
           asan_interceptors_memintrinsics.cpp:63:3
5      #1 0x56211d1d988b in MHD_daemon_add_connection /src/mhd2/src/mhd2/daemon_add_conn.c
           :792:5
6      #2 0x56211d1d1d5d in main /src/poc.cpp:23:3
7  Address 0x7fed29b001a0 is located in stack of thread T0 at offset 160 in frame
8      #0 0x56211d1d94ef in MHD_daemon_add_connection /src/mhd2/src/mhd2/daemon_add_conn.c
           :665
```

```
  This frame has 1 object(s):
    [32, 160) 'addrstorage' (line 668) <== Memory access at offset 160 overflows this
        variable
SUMMARY: AddressSanitizer: stack-buffer-overflow /src/mhd2/src/mhd2/daemon_add_conn.c
    :792:5 in MHD_daemon_add_connection
```

This confirms the stack overflow at the `memcpy` site.

5.2.6 Mitigation

The function should validate the `addrlen` argument to prevent writes beyond the bounds of `addrstorage`. A safe fix is:

```
if (addrlen > sizeof(struct sockaddr_storage))
    return MHD_SC_INVALID_SOCKET_ADDRESS;
```

This ensures oversized socket addresses are rejected gracefully.

5.2.7 Reference

- `MHD_daemon_add_connection` source code
- `microhttpd2.h` API documentation

5.2.8 Reported Issue

- GNUnet MantisBT #10304

5.3 [Libmicrohttpd2] Logic bug in `mhd_strx_to_uint32_n`

Severity	Informational
Status	Fixed
id	ADA-GNU-MHD2-3
Component	mhd_str.c

5.3.1 Problem

This issue occurs in a function that is unused by the library at the time of writing, so the issue is informational at this point.

The issue is in the internal function `mhd_strx_to_uint32_n` in `src/mhd2/mhd_str.c`. A potential problem would only occur if future additions call into this code. `mhd_strx_to_uint32_n` is used to parse a hexadecimal string into a 32-bit unsigned integer **with a maximum character bound** (`maxlen`). Its output is expected to **match exactly** that of the standard `mhd_strx_to_uint32` (without the bound) when `maxlen >= strlen(input)`, similar to how `mhd_strx_to_uint64` and `mhd_strx_to_uint64_n` are consistent.

However, `mhd_strx_to_uint32_n` contains an extra multiply/add operation per loop iteration, which **causes** the result to mismatch with `mhd_strx_to_uint32`. The extra `res *= 16;` and `res += (unsigned int)digit;` at the end of the while loop leads to each digit being processed **twise** in every iteration, making the result far larger than expected.

```
1  while (i < maxlen && (digit = xdigittovalue (str[i])) >= 0)
2  {
3    uint_fast32_t prev_res = res;
4
5    res *= 16;
6    if (res / 16 != prev_res)
7      return 0;
8    res += (unsigned int) digit;
9    if (res < (unsigned int) digit)
10     return 0;
11
12   res *= 16;
13   res += (unsigned int) digit;
14   i++;
15 }
```

The corresponding 64-bit function `mhd_strx_to_uint64_n` performs **one multiply/add per digit**, matching the result with `mhd_strx_to_uint64` and remaining consistent. But `mhd_strx_to_uint32_n` and `mhd_strx_to_uint32` clearly do **not** produce consistent output, which may confuze maintainers or lead to incorrect behaviour in dependant code.

5.3.2 Consequence

As the buggy function is marked as `MHD_Internal`, it is not meant to be invoked directly by public API users. However, incorrect parsing results could propagate into request or response processing that **relies** on this helper. Because integer overflow handling is still guarded by `maxlen` checks and memory verification, it does **not** pose an immediate memory-safety risk. The main risk here is **functional incorrectness**, which can break assumptions or trigger subtle bugs.

5.3.3 Proof of Concept (poc.cpp)

The PoC compares the outputs of `mhd_strx_to_uint32` vs `mhd_strx_to_uint32_n` and `mhd_strx_to_uint64` vs `mhd_strx_to_uint64_n` with several input strings. It demonstrates that the 64-bit versions are consistent, while the 32-bit versions diverge:

```cpp
#include <cstdio>
#include <cstdint>
#include <cstring>

extern "C" {
  #include "mhd_str.h"
}

int main() {
  const char* tests[] = {
    "0", "A", "FF", "FFFFFFFF"
  };

  for (int i = 0; i < 4; i++) {
    const char* hex = tests[i];
    uint_fast32_t v32 = 0, v32n = 0;
    uint_fast64_t v64 = 0, v64n = 0;

    mhd_strx_to_uint32   (hex, &v32);
    mhd_strx_to_uint32_n (hex, strlen(hex), &v32n);
    mhd_strx_to_uint64   (hex, &v64);
    mhd_strx_to_uint64_n (hex, strlen(hex), &v64n);

    std::printf("Hex: %-10s | 32: %-10lu | 32_n: %-20lu | 64: %-12llu | 64_n: %-12llu\n
        ",
        hex, (unsigned long) v32, (unsigned long) v32n,
        (unsigned long long) v64, (unsigned long long) v64n);
  }
  return 0;
}
```

5.3.4 Build and run the poc

```
# Clone project
git clone https://git.gnunet.org/libmicrohttpd2.git mhd2
cd mhd2

# Build project
./autogen.sh
./configure --enable-dauth --enable-md5 --enable-sha256 --enable-sha512-256 \
  --enable-bauth --enable-upgrade --enable-https --enable-messages
ASAN_OPTIONS=detect_leaks=0 make -j$(nproc)
make install
BINARY=./src/mhd2/.libs/libmicrohttpd2.a

# Build poc
clang++ -fsanitize=address -stdlib=libc++ -std=c++17 -DHAVE_CONFIG_H ../test.cpp -I"
    $SRC/mhd2/src/mhd2" -I"$SRC/mhd2/src/include" -I"$SRC/mhd2/src/incl_priv" -I"$SRC/
    mhd2/src/incl_priv/config" ./src/mhd2/.libs/libmicrohttpd2.a -o poc

# Run poc
./poc
```

5.3.5 Output

```
1 Hex: 0          | 32: 0          | 32_n: 0                    | 64: 0            | 64_n
      : 0
2 Hex: A          | 32: 10         | 32_n: 170                  | 64: 10           | 64_n
      : 10
3 Hex: FF         | 32: 255        | 32_n: 65535                | 64: 255          | 64_n
      : 255
4 Hex: FFFFFFFF   | 32: 4294967295 | 32_n: 18446744073709551615 | 64: 4294967295   | 64_n
      : 4294967295
```

This confirms that:

- `mhd_strx_to_uint32_n` is **not consistent** with `mhd_strx_to_uint32`.
- `mhd_strx_to_uint64_n` is consistent with `mhd_strx_to_uint64` as expected.

5.3.6 Suggested Fix

The fix is to mirror `_uint64_n` and remove the duplicated multiply/add logic, which appears to have been added **accidantally**.

5.3.7 Reported Issue

- GNUnet MantisBT #10326

5.4 [Libmicrohttpd2] Logic bug in algorithm enum parsing in `get_rq_dauth_algo`

Severity	Low
Status	Fixed
id	ADA-GNU-MHD2-4
Component	auth_digest.c

5.4.1 Problem

The issue was discovered in the function `get_rq_dauth_algo(...)` within `src/mhd2/auth_digest.c`. When the client's `algorithm` parameter is **quoted**, several mappings return the **wrong** enum, diverging from the behaviour of the unquoted branch. Specifically:

- `"MD5-sess"` is mapped to **`MHD_DIGEST_AUTH_ALGO_SHA512_256`** (should be **`MHD_DIGEST_AUTH_ALGO_MD5_SESSION`**).
- `"SHA-512-256-sess"` is mapped to **`MHD_DIGEST_AUTH_ALGO_MD5_SESSION`** (should be **`MHD_DIGEST_AUTH_ALGO_SHA512`**).
- `"SHA-512-256"` (non-session) is mapped to **`MHD_DIGEST_AUTH_ALGO_SHA512_256_SESSION`** (should be **`MHD_DIGEST_AUTH_ALGO_SHA512_256`**).

Other mappings are correct and consistent with the **unquoted** branch.

```
1  if (algo_param->quoted)
2  {
3    ...
4    if (mhd_str_equal_caseless_quoted_s_bin_n(..., mhd_MD5_TOKEN mhd_SESS_TOKEN))
5      return MHD_DIGEST_AUTH_ALGO_SHA512_256;
6
7    if (mhd_str_equal_caseless_quoted_s_bin_n(..., mhd_SHA512_256_TOKEN mhd_SESS_TOKEN))
8      return MHD_DIGEST_AUTH_ALGO_MD5_SESSION;
9
10   if (mhd_str_equal_caseless_quoted_s_bin_n(..., mhd_SHA512_256_TOKEN))
11     return MHD_DIGEST_AUTH_ALGO_SHA512_256_SESSION;
12 }
```

The enum `MHD_DigestAuthAlgo` is defined in `microhttpd2.h`, which confirms the correct enum values expected here:

```
1  enum MHD_FIXED_ENUM_MHD_APP_SET_ MHD_DigestAuthAlgo
2  {
3    MHD_DIGEST_AUTH_ALGO_INVALID = 0,
4    MHD_DIGEST_AUTH_ALGO_MD5 = ...,
5    MHD_DIGEST_AUTH_ALGO_MD5_SESSION = ...,
6    MHD_DIGEST_AUTH_ALGO_SHA256 = ...,
7    MHD_DIGEST_AUTH_ALGO_SHA256_SESSION = ...,
8    MHD_DIGEST_AUTH_ALGO_SHA512_256 = ...,
9    MHD_DIGEST_AUTH_ALGO_SHA512_256_SESSION = ...,
10 };
```

Each enum encodes both the **base algorithm** (MD5, SHA-256, SHA-512/256) and whether it is a **session variant** (`-sess`). The buggy code erroneously maps quoted values to different enums, breaking the intended logic of algorithm parsing.

5.4.2 Consequence

With quoted tokens (e.g., `algorithm="MD5-sess"`), the server misinterprets the algorithm and computing with the wrong digest, or rejecting an otherwise valid response as unsupported. This can cause interoperability problems in real world Digest authentication clients, making compliant clients appear broken.

5.4.3 Suggested fix

Simply replace the incorrect returned enum values with their correct counterparts. This minimal change restores parity between quoted and unquoted algorithm parsing logic.

5.4.4 Reported Issue

- GNUnet MantisBT #10328

5.5 [Libmicrohttpd2] Conditional check bug in auth scheme parsing

Severity	Informational
Status	Fixed
id	ADA-GNU-MHD2-5
Component	request_auth_get.c

5.5.1 Problem

The function `mhd_request_get_auth_header_value(...)` compares an auth **prefix** (e.g., `"Basic"`, `"Digest"`) against the **header value**, there exist some guards intended to ensure enough bytes are available, but one of them is checking a **wrong variable length**:

```
1      if (hdr_name.len != f->field.nv.name.len)
2        continue;
3      if (MHD_VK_HEADER != f->field.kind)
4        continue;
5      if (prefix_len > f->field.nv.name.len)
6        continue;
7      if (! mhd_str_equal_caseless_bin_n (hdr_name.cstr,
8                                          f->field.nv.name.cstr,
9                                          hdr_name.len))
10       continue;
11     if (! mhd_str_equal_caseless_bin_n (prefix_str,
12                                         f->field.nv.value.cstr,
13                                         prefix_len))
14       continue;
```

There are two calls to the function `mhd_str_equal_caseless_bin_n(...)`, which requires three parameters. The function loops through the 1st and 2nd parameters character-by-character **N** times, where **N** is determined by the 3rd parameter.

```
1  MHD_INTERNAL bool
2  mhd_str_equal_caseless_bin_n (const char *str1, const char *str2, size_t len)
3  {
4    for (size_t i = 0; i < len; ++i)
5    {
6      const char c1 = str1[i];
7      const char c2 = str2[i];
8      if (charsequalcaseless (c1, c2))
9        continue;
10     else
11       return 0;
12   }
13   return !0;
14 }
```

The first call to `mhd_str_equal_caseless_bin_n(...)` is OK because `hdr_name.len` must equal `f->field.nv.name.len`, and thus it **guarantees** the loop won't go past the range of either `hdr_name.cstr` or `f->field.nv.name.cstr`, stopping just before the NUL terminator. This is guarded by the preceding check, which skips the call and moves to the next iteration when the lengths are not equal.

However, the guard for the second `mhd_str_equal_caseless_bin_n(...)` call checks the **wrong** variable. It compares `prefix_len` against `f->field.nv.name.len`, then passes `f->field.nv.value.cstr` to the function. This

incorrect check means that if `f->field.nv.value.len` is less than `prefix_len`, the guard still allows the call, leading to an unnecessary invocation of `mhd_str_equal_caseless_bin_n(...)` that should be skipped. It does not go past either string or cause a buffer problem, because the comparison stops when the NUL terminator of the shorter string is reached. It is only an avoidable extra call.

Here is an example: assume `prefix_str` is `"ABC"`, `f->field.nv.name` is `"ABCDEF"` and `f->field.nv.value` is `"A"`. Then `prefix_len` will be 3 and `f->field.nv.name.len` will be 6, so the check passes. The three parameters passed to `mhd_str_equal_caseless_bin_n(...)` will be `"ABC"`, `"A"` and 3. This results in two iterations, and the function returns 0 when it compares the NUL terminator with the second character of the prefix.

5.5.2 Consequence

This mismatch between the guard and the subsequent call to `mhd_str_equal_caseless_bin_n(...)`—using different string lengths—can cause a performance issue by executing `mhd_str_equal_caseless_bin_n(...)` when it is not needed.

5.5.3 Suggested fix

The minimal fix is to change the conditional guard to use the correct parameter so that it matches the subsequent `mhd_str_equal_caseless_bin_n(...)` call (i.e. compare `prefix_len` with the value length, not the name length).

5.5.4 Reported Issue

- GNUnet MantisBT #10354

5.6 [Libmicrohttpd2] Heap buffer overflow in `response_add_auth_digest_challenge_alg`

Severity	High
Status	Fixed
id	ADA-GNU-MHD2-6
Component	response_auth_digest.c

5.6.1 Problem

A heap-buffer-overflow is found when adding a Digest auth challenge header. The root cause is an incorrect allocation size in `response_add_auth_digest_challenge_alg()` function that uses the size of a pointer to the header struct rather than the size of the struct itself. Because the code then does pointer arithmetic `new_hdr + 1` (which advances by the size of the struct), the calculated string buffer (`hdr_str`) starts past the end of the allocated block, and later `memcpy()` writes overflow the heap.

Faulty allocation (https://git.gnunet.org/libmicrohttpd2.git/tree/src/mhd2/response_auth_digest.c#n213)

```
/* ** Allocate ** */
new_hdr = (struct mhd_RespAuthDigestHeader *)
          malloc (sizeof(struct mhd_RespAuthDigestHeader *)
                  + hdr_maxlen + 1);
if (NULL == new_hdr)
  return MHD_SC_RESPONSE_HEADER_MEM_ALLOC_FAILED;
hdr_str = (char *) (new_hdr + 1);
```

The `sizeof(...)` uses `sizeof(struct mhd_RespAuthDigestHeader *)` instead of `sizeof(struct mhd_RespAuthDigestHeader)` which returns the size of pointer(always 8 bytes in x64) instead of the size of the struct. Immediately after, `hdr_str` is computed as `(char*)(new_hdr + 1)`, and **`new_hdr + 1` advances by the size of the struct**, not by the size of a pointer, placing `hdr_str` outside the allocated region.

After the faulty allocation, the function builds the header by repeatedly copying pieces into `hdr_str` (https://git.gnunet.org/libmicrohttpd2.git/tr

```
memcpy(hdr_str + pos, hdr_pref_realm_pref.cstr, hdr_pref_realm_pref.len);
```

These writes land beyond the end of the originally allocated block and cause overflow.

5.6.2 Consequence

Because the allocation is too small, subsequent string copies write past the end of the allocated buffer. This leads to heap memory corruption, which may cause crashes, unpredictable behaviour, or potentially allow remote code execution when Digest authentication headers are generated.

5.6.3 Reported Issue

- GNUnet MantisBT #10253

5.7 [Libmicrohttpd2] Memory leak in `mhd_pool_create` / `mhd_pool_destroy`

Severity	Moderate
Status	Fixed
id	ADA-GNU-MHD2-7
Component	mempool_funcs.c

5.7.1 Problem

The problematic code is in the `mhd_pool_create` function found in `src/mhd2/mempool_funcs.c` (link). The snippet below uses several macro-definition checks, so the compiler may produce different variants depending on which macros are defined.

```
#ifdef mhd_USE_LARGE_ALLOCS
  pool->is_large_alloc = false;
  if ( (max <= 32 * 1024) ||
       (max < MHD_sys_page_size_ * 4 / 3) )
  {
    pool->memory = (uint8_t *) mhd_MAP_FAILED;
  }
  else
  {
    /* Round up allocation to page granularity. */
    alloc_size = max + MHD_sys_page_size_ - 1;
    alloc_size -= alloc_size % MHD_sys_page_size_;
    pool->is_large_alloc = true;
#  if defined(mhd_MAP_ANONYMOUS)
    pool->memory = (uint8_t *) mmap (NULL,
                                     alloc_size,
                                     PROT_READ | PROT_WRITE,
                                     MAP_PRIVATE | MAP_ANONYMOUS,
                                     -1,
                                     0);
#  else  /* ! mhd_MAP_ANONYMOUS */
    pool->memory = (uint8_t *) VirtualAlloc (NULL,
                                             alloc_size,
                                             MEM_COMMIT | MEM_RESERVE,
                                             PAGE_READWRITE);
#  endif /* ! mhd_MAP_ANONYMOUS */
  }
#else  /* !mhd_USE_LARGE_ALLOCS */
  if (mhd_MAP_FAILED != pool->memory)
    pool->is_large_alloc = true;
  else
#endif /* !mhd_USE_LARGE_ALLOCS*/
  if (! 0)
  {
    alloc_size = mhd_ROUND_TO_ALIGN (max);
    if (MHD_MEMPOOL_ZEROING_NEVER == zeroing)
      pool->memory = (uint8_t *) malloc (alloc_size);
    else
      pool->memory = (uint8_t *) mhd_calloc (1, alloc_size);
    if (((uint8_t *) NULL) == pool->memory)
    {
      free (pool);
```

```
43          return NULL;
44        }
45    }
```

When **both** `mhd_USE_LARGE_ALLOCS` and `mhd_MAP_ANONYMOUS` are defined (as in the OSS-Fuzz Linux environment), the effective logic reduces to:

If `max` is `<= 32 * 1024` **or** `< (MHD_sys_page_size_ * 4 / 3)`, the code follows the small-allocation path and later allocates via `malloc`/`mhd_calloc`, which is fine.

However, when `max` exceeds `32 * 1024` (and meets the page-sized threshold), a bug appears: `pool->is_large_alloc` is set to **`true`** and `pool->memory` is allocated with `mmap`. Immediately afterwards, the unconditional **`if`** `(!0)` block runs, and `pool->memory` is reassigned to a new `malloc`/`mhd_calloc` allocation **without** freeing the original `mmap` region. This creates the **first memory leak**. Although the `mmap` region will be automatically `munmap` when the process terminates, that region remains unusable until process exit, creating a runtime memory leakage problem.

More seriously, when the pool is later destroyed by `mhd_pool_destroy` (link), a **second** issue is triggered:

```
1  MHD_INTERNAL void
2  mhd_pool_destroy (struct mhd_MemoryPool *restrict pool)
3  {
4    if (NULL == pool)
5      return;
6
7    mhd_assert (pool->end >= pool->pos);
8    mhd_assert (pool->size >= pool->end - pool->pos);
9    mhd_assert (pool->pos == mhd_ROUND_TO_ALIGN (pool->pos));
10   mhd_UNPOISON_MEMORY (pool->memory, pool->size);
11 #ifdef mhd_USE_LARGE_ALLOCS
12   if (pool->is_large_alloc)
13   {
14 #  if defined(mhd_MAP_ANONYMOUS)
15     munmap (pool->memory,
16             pool->size);
17 #  else
18     VirtualFree (pool->memory,
19                  0,
20                  MEM_RELEASE);
21 #  endif
22   }
23   else
24 #endif /* mhd_USE_LARGE_ALLOCS*/
25   if (! 0)
26     free (pool->memory);
27
28   free (pool);
29 }
```

With `mhd_USE_LARGE_ALLOCS` and `mhd_MAP_ANONYMOUS` defined, the effective logic becomes:

```
1  MHD_INTERNAL void
2  mhd_pool_destroy (struct mhd_MemoryPool *restrict pool)
3  {
4    if (NULL == pool)
5      return;
6
7    mhd_assert (pool->end >= pool->pos);
8    mhd_assert (pool->size >= pool->end - pool->pos);
9    mhd_assert (pool->pos == mhd_ROUND_TO_ALIGN (pool->pos));
10   mhd_UNPOISON_MEMORY (pool->memory, pool->size);
11   if (pool->is_large_alloc)
12   {
```

```
13      munmap (pool->memory,
14              pool->size);
15    }
16    else
17    if (! 0)
18      free (pool->memory);
19
20    free (pool);
21  }
```

Because `mhd_pool_create` set `pool->is_large_alloc = true` but then **overwrote** `pool->memory` with a heap pointer, `mhd_pool_destroy` ends up calling `munmap` on memory that was allocated by `malloc` or `mhd_calloc`. This is undefined behaviour: it may crash, silently no-op, or corrupt memory. In the latter two cases, the heap allocation is never freed, which is how the **second leak** occurs.

In short, a large-allocation path leaves an `mmap` region leaked (first leak), and the destroy path may call `munmap` on a heap pointer allocated by `malloc` or `mhd_calloc` while failing to `free` it, causing a secondary leak. This likely occurs only when the two macros above are enabled simultaneously.

5.7.2 Suggested fixes

It is assumed `pool->memory` should use only `mmap` or the `malloc`/`mhd_calloc` to allocate. So a suggested fix is to change the condition in the `mhd_pool_creation` from `if (!0)` to `if (!pool->is_large_alloc)` to ensure the `pool->memory` should only be allocated with one approach.

5.7.3 Reported Issue

- GNUnet MantisBT #10296

6 Future work

In general we consider libmicrohttpd2 to be a well-written piece of software, but suggest various improvements that could help the security posture of the project.

6.1 Extend the fuzzing suite to improve code coverage.

The OSS-Fuzz integration we created has a total of 9 fuzzers and achieves a code coverage of around 35%. We suggest the maintainers extend on this OSS-Fuzz integration with a focus on achieving higher code coverage. Data on the fuzzing project can be found here. Which holds links to the coverage report, such as 21st September 2025.

Example files where coverage can be increased and that includes parsing logic include stream_process_request.c and post_parser_funcs.. These would be good targets to approach as a first.

In general, we suggest targeting code coverage of 75% and consider that this would represent a mature and strong fuzzing set up.

6.2 Add security focused documentation

A step towards improving general security posture is to add a dedicated security documentation for libmicrohttpd2. This documentation could include:

1) Documentation on processes used by the project to maintain security (e.g. referencing fuzzing, how it's used etc.)
2) Documentation on how to report security vulnerabilities if found.
3) Documentation on general assumptions made by the code and potential attack surface. This could include specific guides for users on what input/output needs to be sanitized when using libmicrohttpd2.
4) Suggestions, if possible, on how to contribute to libmicrohttpd2's security posture, e.g. extending fuzzers, fixing warnings from SAST tools or alike.

7 Conclusions

In this report we summarised the audit of GNU Libmicrohttpd2 performed using static analyse, manual review, and fuzzing. The main goal was to evaluate the project's security and robustness, and this was achieved through a combined effort that lead to the discovery of seven issues in total. These include two memory-safety vulnerabilities (stack and heap buffer overflows), two conditional check flaws, two logic erros, and one memory leak. The manual audit uncovered five subtle logic and validation problems, and the fuzzing uncovered two issues.

We have created an extension fuzzing integration of `libmicrohttpd2` and enabled conitnuous fuzzing by way of OSS-Fuzz. As such, the fuzzing harness are now run daily in Google's cloud based on the most recent source in `libmicrohttpd2`, providing continuous testing of the project's codebase. This ensure that regressions and new vulnerabilites can be detected quickly, while also contributing to ongoing coverage growth.

In general we consider libmicrohttpd2 to be a well-written software package, but we also include several recommendations for steps that can be taken to further improve the security posture of `libmicrohttpd2`.

Finally, we would like to thank the Sovereign Tech Agency for funding our efforts and OSTIF for facilitating this audit.