Security Audit Report for the Mozilla Secure Open Source Fund

GNU libmicrohttpd (MHD)

Overview

The Least Authority security consultancy performed a security audit of the GNU libmicrohttpd library, on behalf of Mozilla's Secure Open Source Fund. GNU libmicrohttpd (https://www.gnu.org/software/libmicrohttpd/) is a C library with about 35,000 lines of code to allow developers to embed HTTP server functionality into their applications, of which about 19,000 lines of code are for the directory containing the library C header files and about 16,000 lines of test code.

Background

Recently, the GNU libmicrohttpd team of Christian Grothoff and Evgeny Grin (co-maintainers) have completed the features and consider the project ready for version 1.0. In accordance with their plans for this significant release, they would like to have the code audited to ensure it is more secure. Previously, the code was audited twice by RedHat and Ubuntu. Although only minor issues were discovered in both of these audits, the code base has grown significantly since the last review. The primary concern for the audit is that the code is of the highest quality as possible, not the speed of the release, especially due to the number of other projects that are dependent on the GNU libmicrohttpd library.

Coverage

Target Code and Revision

All file references are based on the version 0.9.52 and git tag of the GNU libmicrohttpd codebase, which has revision id:

```
938b9b8dae70739c6e629bf144b57b5d6212e6b1
```

All file references use Unix-style paths relative to the working directory root.

Dependencies

Although our primary focus was on the application code, we examined dependency code and behavior where relevant to a particular line of investigation.

The dependencies are GNU TLS library for https support and curl for testing, these were not reviewed.

Strategy

We determined that the following activities would be the primary focus of our audit work:

- Manual code review with some automated tests and analysis.
- Analyze the protocol parser.
- Look for vulnerabilities with multi-threading (MHD has 4 threading modes)
- Review the code changes since last audits or the otherwise less reviewed code.
- Look at the implementation and use of the standard libraries.
- Look for opportunities for Denial of Service attacks in scope.

Manual Code Review

In manually reviewing the code, we looked for any potential issues with code logic or formatting, error handling, protocol and header parsing, memory/array out of bounds read/writes and other related issues. This included looking for misleading indentation errors that could produce some unintended consequences, usages of `sprintf()` that could potentially lead to format string attacks, usages of unbounded string functions, `malloc()`'s without `free()`'s, double `free()` (i.e. `free()` called on an already freed pointer, reads from the network etc.). We looked carefully at all the network facing code and see if any buffers leak any secrets. We looked at the parsers with an eye on the strict compliance with the protocols it is implementing and report the deviations, if any.

The files we manually reviewed included:

- base64.c
- basicauth.c
- connection.c
- daemon.c
- digestauth.c
- internal.c
- memorypool.c
- md5.c

- postprocessor.c
- response.c
- mhd_str.c
- mhd_threads.c
- mhd_sockets.c

Code Analysis

In addition to manually reviewing the code, we used a few other tools to analyze potential problems. Some tools that we attempted to use are:

- gcc's address sanitizer (asan) to find double frees, buffer overflow and other memory bugs.
- gcc's undefined behaviour sanitizer (ubsan) to detect C undefined behaviour.
- gcc's thread sanitizer (tsan) to detect race conditions.
- Clang's static analyzer scanbuild.
- cppcheck, another static analyser.
- David Wheeler's flawfinder.
- American Fuzzy Lop (afl) to look for parsing bugs.

We also compiled the source code with `-Wall -pedantic` and look at all the warnings (if any) and formatted the code using *astyle* to reveal any formatting bugs in C code. After the Apple's "goto fail" and "gnutls" vulnerabilities, gcc has added a flag `-Wmisleading-indentation` to detect and warn for bugs caused by indentation.

We wrote test cases to evaluate the features of the library, writing and extending test programs (as preparation for fuzzing). We reviewed tests in the library and add any that will highlight any bugs. (Note: These have not been submitted to the upstream maintainers yet.) We reviewed the documentation.

Fuzzing

We performed fuzzing with American fuzzy lop (AFL) to look for parsing related bugs, such as unintended behaviors and potential errors, by generating valid input cases for the program. We were looking for opportunities for an attacker to crash or use semi-valid input to perform specific behaviors within MHD. AFL could potentially report bugs in the parsers where for certain inputs, the parser goes into an undefined state and then after, rejects valid inputs, effectively doing Denial of Service. It could also expose bugs where certain types of inputs could cause memory safety related bugs (like the Heartbleed attack). In order to do the fuzzing, we created and extended test programs that could be added to the MHD's build system.
We chose AFL to look for parsing related bugs because of its popularity and high level of code coverage. Since AFL works only on stdin/stdout/stderr, we used a certain modified version that

works for network clients and servers. We did not run AFL tests on all possible modes in MHD as AFL takes a long time to perform the fuzzing. We ran it for certain inputs with some test programs. In all cases, AFL ran (sometimes for several days) without any hangs or crashes and we did not find any problems.

Not in Scope

We did not do a performance analysis of GNU libmicrohttpd during this audit. The dependencies (GNU TLS library for https support and curl for testing) were not reviewed.

Findings

Code Selection

GNU microhttpd is targeted to run on wide ranging devices from low power router hardware to large machines, driving the choice to write it in C. Often, C is also chosen for compatibility with certain resource-constrained environments. However, C standards, for various historical reasons, have made choices that makes writing secure programs hard, but not impossible. Because of this, we want to note that C is not necessarily the safest choice for writing such a library. Nonetheless, we understand and respect the selection of the authors and proceeded with our review of the code with the goal of making it more secure despite this.

Code Quality

Overall, we would like to note that the quality of the code reviewed was impressive, even considering the many lines of code included. It is clear that the project authors and maintainers have dedicated themselves to a high standard of development, both with writing and reviewing code. The two previous audits of this project surely contributed to the high standard for security, too. With this in mind, we found ourselves further challenged to look for vulnerabilities and opportunities for improvements. If all projects were written to this standard, there would be far fewer attacks on the open source projects that contribute to the current state of technology.

Issues

We list the issues we found in the library in the order we found and reported them.

Issue A: Use of a file descriptor before it is initialized, when compiled with all warnings on (-Wall)

Severity: Low

Synopsis: We found an uninitialized file descriptor being used when the library is compiled with `-Wall`, which has not been `open`'ed in **daemon.c**. Compilation with these options revealed a file descriptor used before initialization warning in daemon.c:1870 (`sendfile(...)`) function, which appears genuine.

Technical Details: In **daemon.c**, file descriptor variable, `fd` has not been initialized and is being passed in the code below. In Linux kernel, `sendfile(2)` system call allows copying from one file descriptor to another without the data getting into the userspace (as one would do with a `read(2)` and `write(2)` system calls), thereby making it very efficient.

```
offset = (off64_t) offsetu64;
if ( (offsetu64 <= (uint64_t) OFF64_T_MAX) &&
           (0 < (ret = sendfile64 (connection->socket_fd,
                                   fd,
                                   &offset,
                                   left))) )
```

Mitigation: We recommend to initialize the file descriptor. This has already been fixed in the git 'master' branch. There is a proliferation of `#ifdef` macros for portability. The number of code paths that is possible with `N` such macros is in the order of 2^N. It would be nice on the code reader if the macros are reduced to the absolute minimum required.

Remediation: We recommend compiling with warnings on(i.e. `-Wall`), all the time or may be with `-Werror` that will turn warnings into error.

Status: This was reported to the MHD team and we believe it has already been fixed.

Issue B: Use of unbounded string functions

Severity: Medium
Synopsis: We found multiple uses of unbound versions of string functions (string formatting, string copy, etc.) during our manual code review.

Technical Details: String formatting functions write out strings in a given buffer by representing the given input data in the specified format. E.g.:

```
char buf[10];
char *str = "hello world";
sprintf(buf, "%s", str);
```

Other functions like `strcpy(3)` copy the given source string to a destination string. These functions assume a properly NULL terminated string as a source string and do not check

whether the destination string is long enough to hold the source string. The use of such functions are a common source of buffer overflow bugs. The use of bounded version of such functions (like `strncpy(3)` or `strlcpy(3)`) is safer.

Impact: If the input in the string is too long it will overflow and write arbitrary data into adjacent memory. Buffer overflows may be unknowingly caused by modifications during refactoring if the string to be written into is defined elsewhere as a macro.

Preconditions: For this vulnerability to be exploitable, the input string should be controllable by the attacker via input, so that it could overwrite adjacent buffers and thereby control their values as well.

Technical Details: in **connection.c** we found the following occurrences of unbounded string functions:

```
  sprintf (date,
           "Date: %3s, %02u %3s %04u %02u:%02u:%02u GMT\r\n",
           days[now.tm_wday % 7],
           (unsigned int) now.tm_mday,
           mons[now.tm_mon % 12],
           (unsigned int) (1900 + now.tm_year),
           (unsigned int) now.tm_hour,
           (unsigned int) now.tm_min,
           (unsigned int) now.tm_sec);
```

...

```
     sprintf (code,
              "%s %u %s\r\n",
           (0 != (connection->responseCode & MHD_ICY_FLAG))
           ? "ICY"
           : ( (MHD_str_equal_caseless_ (MHD_HTTP_VERSION_1_0,
                    connection->version))
           ? MHD_HTTP_VERSION_1_0
           : MHD_HTTP_VERSION_1_1),
           rc,
           reason_phrase);
```

...

```
content_length_len
     = sprintf (content_length_buf,
                    MHD_HTTP_HEADER_CONTENT_LENGTH ": "
MHD_UNSIGNED_LONG_LONG_PRINTF "\r\n",
                    (MHD_UNSIGNED_LONG_LONG)
connection->response->total_size);

…

off += sprintf (&data[off],
                "%s: %s\r\n",
                pos->header,
                pos->value);

…

  if (MHD_CONNECTION_FOOTERS_RECEIVED == connection->state)
     {
     strcpy (&data[off],
               date);
     off += strlen (date);
     }

…

  if (0 == nc)
     {
     /* Fresh nonce, reinitialize array */
     strcpy (nn->nonce,
               nonce);
     …
     …
     return MHD_YES;
     }

…

     daemon->custom_error_log = (MHD_LogCallback) &vfprintf;
```

…

Mitigation: A safer way for string construction and copy function would be to use the bounded versions that also take a maximum length as input. The `sprintf` function has a bounded version, `snprintf`. The `vfprintf()` function in glibc has had a number of vulnerabilities in the past. We think it would be best to avoid the use of it if possible. An alternative would be to use a combination of `vsnprintf()` to write into a string and then use `fprintf` to write the resulting string into the given file descriptor. Since it is being used for logging in this case, just rewriting the code to use the non-variable-length argument functions might be sufficient.

Remediation: It may be a good idea to avoid the unsafe versions of C string manipulation and formatting functions and instead use the safe, bounded variants of them.

Issue C: Whitespace RFC 7230 header rules noncompliance

Severity: Low
Synopsis: HTTP 1.1 is defined by RFC 7230 which supercedes RFC 2616. Appendix A.2 lists the changes in RFC 7230 with respect to RFC 2616. We found that there is a compliance issue with the header rules in **connection.c**. Appendix A.2 in rfc7230 mandates that headers (which are of the form "field: value") should not have any trailing whitespaces in the "field name" part after the name and before the ':'. Invalid whitespaces around field-names are now required to be rejected - according to the RFC 7230 Section (3.2.4)(https://tools.ietf.org/html/rfc7230#section-3.2.4). Here is the relevant text from Appendix A.2 in RFC 7230:

> "*Invalid whitespace around field-names is now required to be rejected, because accepting it represents a security vulnerability. The ABNF productions defining header fields now only list the field value.*"

Impact: Invalid whitespaces around field-names is now required to be rejected with an *HTTP 400 Bad Request* response. But MHD does not handle it for the version we audited. We tested for headers spanning multiple lines and there was a correct *HTTP 400 Bad Request* response, so we think the impact of this issue is minor.

Preconditions: For this vulnerability to be exploitable, the HTTP client needs to send a request with a header field name with spaces of this form.

```
curl hostname:port -H 'header with spaces: value'
```

Technical Details: From reading the code in **connection.c**, it looks like this is not handled by MHD: The following are the relevant functions.

```
process_broken_line(),
process_header_line(),
parse_connection_headers()
```

We tested with a simple example and it does not return a HTTP 400 (Bad request) as mandated by the standard.

Also this session with the src/microhttpd/examples/minimal_example server shows that the library does not handle it properly:

1. On one terminal window, start minimal_server:
   ```
   $ ./minimal_server 8080
   ```

2. On another terminal window:

Case 1: Valid header. Correct response from the server.

```
$ curl -vv localhost:8080 -H 'Host: 127.0.0.1'
* Rebuilt URL to: localhost:8080/
*   Trying ::1...
* TCP_NODELAY set
* connect to ::1 port 8080 failed: Connection refused
*   Trying 127.0.0.1...
* TCP_NODELAY set
* Connected to localhost (127.0.0.1) port 8080 (#0)
> GET / HTTP/1.1
> Host: 127.0.0.1
> User-Agent: curl/7.52.1
> Accept: */*
>
< HTTP/1.1 200 OK
< Connection: Keep-Alive
< Content-Length: 90
< Date: Fri, 07 Apr 2017 09:13:55 GMT
<
* Curl_http_done: called premature == 0
* Connection #0 to host localhost left intact
<html><head><title>libmicrohttpd
demo</title></head><body>libmicrohttpd demo</body></html>
```

Case 2: Invalid header. Incorrect response from the server.

```
$ curl -vv localhost:8080 -H 'Host : 127.0.0.1'
```

```
* Rebuilt URL to: localhost:8080/
*   Trying ::1...
* TCP_NODELAY set
* connect to ::1 port 8080 failed: Connection refused
*   Trying 127.0.0.1...
* TCP_NODELAY set
* Connected to localhost (127.0.0.1) port 8080 (#0)
> GET / HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/7.52.1
> Accept: */*
> Host : 127.0.0.1
>
< HTTP/1.1 200 OK
< Connection: Keep-Alive
< Content-Length: 90
< Date: Fri, 07 Apr 2017 09:13:49 GMT
<
* Curl_http_done: called premature == 0
* Connection #0 to host localhost left intact
<html><head><title>libmicrohttpd
demo</title></head><body>libmicrohttpd demo</body></html>
```

Case 3: Invalid header. Incorrect response from the server.

```
$ curl -vv localhost:8080 -H 'Host name: 127.0.0.1'
* Rebuilt URL to: localhost:8080/
*   Trying ::1...
* TCP_NODELAY set
* connect to ::1 port 8080 failed: Connection refused
*   Trying 127.0.0.1...
* TCP_NODELAY set
* Connected to localhost (127.0.0.1) port 8080 (#0)
> GET / HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/7.52.1
> Accept: */*
> Host name: 127.0.0.1
>
< HTTP/1.1 200 OK
< Connection: Keep-Alive
< Content-Length: 90
< Date: Fri, 07 Apr 2017 09:13:42 GMT
```

```
<
* Curl_http_done: called premature == 0
* Connection #0 to host localhost left intact
<html><head><title>libmicrohttpd
demo</title></head><body>libmicrohttpd demo</body></html>
```

Remediation: Although of minor impact, it would still be better if the MHD complies strictly with the RFC 7230 standards.

Status: The issue exists in the version we audited.

Suggestions

Suggestion 1: Improvements to daemon.c

Severity: Informational
Synopsis:

1. *daemon.c:5375* - The body of this `if` statement is always executed. We could as well omit the `if` and just put the body in there.

```
      /* Always use individual control ITCs */
      if (1)
          {
             if (! MHD_itc_init_ (d->itc))
                 {
#ifdef HAVE_MESSAGES
                   MHD_DLOG (daemon,
                             _("Failed to create worker
inter-thread communication channel: %s\n"),
                             MHD_itc_last_strerror_() );
#endif
                   goto thread_failed;
                 }
          }
```

2. *daemon.c:internal_add_connection()* function - variable '*eno*' can potentially get used before initialization.

Project Team

Ramakrishnan Muthukrishnan: Code Reviewer
Ramakrishnan (Ramki) is a Debian developer and lives in Bangalore, India. He has contributed to a bunch of Free software projects like GNU Emacs, Linux kernel and the GNU Radio. He likes to tinker with low-level system software and also enjoys learning and playing with Functional Programming.

Liz Steininger: Project Manager
Liz is a supporter of open source software that encourages transparency and access to information, along with software that enables individuals to freely express themselves and retain the ability to control their own information. She has over 15 years of experience as a Program and Project Manager, Strategist and Analyst working towards these goals.

Zooko Wilcox: Advisory Security Analyst
Zooko has more than 20 years of experience in open, decentralized systems, cryptography and information security, and startups. He is recognized for his work on DigiCash, Mojo Nation, ZRTP, “Zooko's Triangle”, Tahoe-LAFS, BLAKE2, and SPHINCS. He is also the Founder of Least Authority. He sometimes blogs about health science. He tweets a lot.

Daira Hopwood: Advisory Security Analyst
Daira Hopwood participated in the standardization of the TLS protocol and Internationalized Domain Names, found security bugs and design flaws in Java Virtual Machines, wrote code for the Cryptix cryptography library, and did security auditing for the Caja Secure JavaScript project. Daira is a major contributor to the LAFS project and to development of the Zcash cryptocurrency. In hir spare time, ze is designing a capability-secure programming language called "Noether".