FALL
2020

PRESENTS

Fuzzing integration, vulnerability analysis and bug-fixing in Fluent Bit log-processor

in collaboration with the Fluent Bit maintainers and sponsored by The Linux Foundation

Authors

David Korczynski <david@adalogics.com>
Adam Korczynski <adam@adalogics.com>
Date: 10th December, 2020

Executive summary

Goal of engagement

The overall goal of the engagement described in this report was to integrate vulnerability analysis by way of fuzzing into the Fluent Bit project. This was done in a manner such that vulnerability analysis will happen continuously (even after the engagement) and effort was also spent in fixing many of the vulnerabilities found. The source code of Fluent Bit is written in C and this makes Fluent Bit susceptible to memory corruption vulnerabilities and bugs, and these were the concern of this engagement.

Scope of engagement

The focus of the engagement was fiuzzing the core Fluent Bit engine, which corresponds to the C files inside the **src/** directory of the Fluent Bit source code. In this context, the Fluent Bit source code corresponds to the code in the Github repository https://github.com/fluent/Fluent Bit

Methodology

Ada Logic’s security researchers performed an initial analysis of the Fluent Bit library to understand where the Fluent Bit library exposes APIs that may be used by plugins or external developers. These APIs correspond to the entry points of the library and make up the threat model of this engagement. Following this, Ada Logics entered a phase of developing fuzzers that attack these APIs in a myriad of ways, and fixing the security bugs found by these fuzzers. This was done repeatedly for several iterations. The fuzzers and the fixes are all integrated into the Fluent Bit source code, and an infrastructure was set up such that the fuzzers are run by Google’s OSS-Fuzz service. The effect of this is that the fuzzers run continuously both during and after this engagement.

Results summarised

16 fuzzers developed for Fluent Bit

More than 30 Bugs found in the code, with 20 that have security relevance

16 vulnerabilities and bugs fixed

Code and fixes committed upstream, this includes more than 40 commits.

Integration of continuous fuzzing by way of OSS-Fuzz

Engagement process and methodology

In this section we go through the overall approach and process of the project. flb_parser.c

Initial assessment of library

The first step was to perform an initial assessment of the code in the **src/** directory of the Fluent Bit source code. The code in this directory is used explicitly by plugins and the Fluent Bit main application, and these plugins are exposed to potential adversarial inputs. The primary concern of this initial assessment of the library was, therefore, to determine the parts of the library that are relevant for fuzzing and the parts that are related to each other. In this section we outline these.

Parsing routines

An obvious source of complexity are the parsers that exist in the Fluent Bit code, for both json, ltsvc, logfmt and regex parsing, which are primarily placed in the files:

- flb_parser.c
- flb_parser_decoder.c
- flb_parser_json.c
- flb_parser_logfmt.c
- flb_parser_ltsv.c
- flb_parser_regex.c
- flb_config.c
- flb_config_map.c

All of these files have been fuzzed extensively and several bugs were found (see result section below for more details on bugs).

Data packing:

Routines related to data packing and unpacking are equally relevant and similarly contain a high degree of complexity. The two main files related to data packing are:

- flb_pack.c
- flb_pack_gelf.c

The code in both of these files have been fuzzed extensively and bugs have been found in the packing routines as well.

Utility code and data structures

Another important category of code that was fuzzed are the routines related to utility logic of Fluent Bit. This includes routines for converting strings representing time formats, URI decoding and more. This category of code include:

- flb_unescape.c
- flb_uri.c
- flb_strptime.c

- flb_hash.c
- flb_sds.c
- flb_sha512.c
- flb_slist.c
- flb_gzip.c
- flb_utils.c
- flb_ra_key.c
- flb_record_accessor.c
- flb_kv.c
- flb_mp.c

Http client and aws logic

The http-logic as well as logic related to AWS Signature Version 4 also contain various parser routines and are highly suitable for fuzzing. The files in this category include:

- flb_http_client.c
- flb_signv4.c

General Engine

A final category of code includes the logic related to running a Fluent Bit engine. Although this is more difficult to fuzz since it has artifacts in the code that is more geared towards an application rather than a library it requires a bit more careful consideration. Files in this list that contain logic related to engine execution include:

- flb_engine.c
- flb_router.c
- flb_input.c
- flb_output.c

We have deployed fuzzers for all of these different categories of logic in the Fluent Bit base.

Fuzzer writing

All fuzzers are placed in the directory Fluent Bit/tests/internal/fuzzers/ in the Fluent Bit source code. The following table gives an overview of the fuzzers and which parts of the code they target:

Fuzzer name	Target code
config_fuzzer.c	Parsing routines
config_map_fuzzer.c	Parsing routines
flb_json_fuzzer.c	Parsing routines
http_fuzzer.c	Http client and aws logic

msgpack_parser_fuzzer.c	Parsing routines
parse_ltsv_fuzzer.c	Parsing routines
parser_fuzzer.c	Parsing routines
parser_json_fuzzer.c	Parsing routines
msgpack_to_gelf_fuzzer.c	Data packing:
pack_json_state_fuzzer.c	Data packing:
signv4_fuzzer.c	Http client and aws logic
http_fuzzer.c	Http client and aws logic
record_ac_fuzzer.c	Utility code and data structures
strp_fuzzer.c	Utility code and data structures
utils_fuzzer.c	Utility code and data structures
engine_fuzzer.c	General Engine

Pull-requests with fuzzer code

The following pull requests (sorted by date of PR, earliest first) contain all of the various fuzzer integrations:

1. https://github.com/fluent/fluent-bit/pull/2090
2. https://github.com/fluent/fluent-bit/pull/2114
3. https://github.com/fluent/fluent-bit/pull/2502
4. https://github.com/fluent/fluent-bit/pull/2541
5. https://github.com/fluent/fluent-bit/pull/2665
6. https://github.com/fluent/fluent-bit/pull/2725
7. https://github.com/fluent/fluent-bit/pull/2778
8. https://github.com/fluent/fluent-bit/pull/2816

In general, the following URL will enable you to see all commits related to fuzzing as well as fixes: https://github.com/fluent/fluent-bit/commits?author=DavidKorczynski

Example fuzzer

In this section we describe one of the fuzzers integrated into the project.

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

```
#include <string.h>
#include <stdlib.h>
#include <Fluent Bit/flb_time.h>
#include <Fluent Bit/flb_parser.h>

int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size){
    void *out_buf = NULL;
    size_t out_size = 0;
    struct flb_time out_time;
    struct flb_config *fuzz_config;
    struct flb_parser *fuzz_parser;

    /* json parser */
    fuzz_config = flb_config_init();
    fuzz_parser = flb_parser_create("fuzzer", "json", NULL, NULL,
                                    NULL, NULL, MK_FALSE, NULL,
                                    0, NULL, fuzz_config);
    flb_parser_do(fuzz_parser, (char*)data, size,
                  &out_buf, &out_size, &out_time);

    if (out_buf != NULL) {
        free(out_buf);
    }

    flb_parser_destroy(fuzz_parser);
    flb_config_exit(fuzz_config);

    return 0;
}
```

Bug fixing

After having created the fuzzers and have them run, the next step of the engagement is to fix bugs. The bugs that were found have all been shared with the maintainers of Fluent Bit. As such, we will not go in detail with each of these in this report, but rather go through a single bug to illustrate the process as well as an example of a bug in Fluent Bit.

We describe the bug with ID 5162073690210304 (forward referencing to table with bugs fixed), which is a stack-based buffer overflow. The report from Address Sanitizer is as follows:

```
=================================================================
==1==ERROR: AddressSanitizer: stack-buffer-overflow on address 0x7ffd68dad750 at pc 0x000000521f97
bp 0x7ffd68dad650 sp 0x7ffd68dace18
READ of size 128 at 0x7ffd68dad750 thread T0
    #0 0x521f96 in __asan_memcpy
/src/llvm-project/compiler-rt/lib/asan/asan_interceptors_memintrinsics.cpp:22:3
    #1 0x556b67 in flb_sds_cat Fluent Bit/src/flb_sds.c:132:5
    #2 0x556098 in flb_msgpack_gelf_value Fluent Bit/src/flb_pack_gelf.c:140:15
    #3 0x553c67 in flb_msgpack_to_gelf Fluent Bit/src/flb_pack_gelf.c:720:27
    #4 0x556367 in flb_msgpack_raw_to_gelf Fluent Bit/src/flb_pack_gelf.c:787:11
    #5 0x552947 in LLVMFuzzerTestOneInput Fluent
Bit/tests/internal/fuzzers/msgpack_to_gelf_fuzzer.c:15:14
    #6 0x459e51 in fuzzer::Fuzzer::ExecuteCallback(unsigned char const*, unsigned long)
/src/llvm-project/compiler-rt/lib/fuzzer/FuzzerLoop.cpp:595:15
    #7 0x444f22 in fuzzer::RunOneTest(fuzzer::Fuzzer*, char const*, unsigned long)
/src/llvm-project/compiler-rt/lib/fuzzer/FuzzerDriver.cpp:323:6
    #8 0x44afde in fuzzer::FuzzerDriver(int*, char***, int (*)(unsigned char const*, unsigned
long)) /src/llvm-project/compiler-rt/lib/fuzzer/FuzzerDriver.cpp:852:9
    #9 0x472fc2 in main /src/llvm-project/compiler-rt/lib/fuzzer/FuzzerMain.cpp:20:10
    #10 0x7fcd781d682f in __libc_start_main /build/glibc-LK5gWL/glibc-2.23/csu/libc-start.c:291
    #11 0x4201f8 in _start
Address 0x7ffd68dad750 is located in stack of thread T0 at offset 144 in frame
    #0 0x552a8f in flb_msgpack_to_gelf Fluent Bit/src/flb_pack_gelf.c:407
```

Root-cause analysis of this bug revealed that following code inside of **flb_pack_gelf.c** led to the error:

```
char temp[48] = {0};
...

else if (v->type == MSGPACK_OBJECT_POSITIVE_INTEGER) {
    val = temp;
    val_len = snprintf(temp, sizeof(temp) - 1,
                       "%" PRIu64, v->via.u64);
}
...

if (v->type == MSGPACK_OBJECT_EXT) {
    tmp = flb_msgpack_gelf_value_ext(s, quote, val, val_len);
}
```

The behaviour of the above code is to set **val** to be **temp** and set the content of **val (temp)** by way of **sprintf** and the expected behaviour is that **sprintf** returns the number of characters printed to the target buffer. Then, **val** and **temp** are passed to **flb_msgpack_gelf_value_ext** where the logic inside of this function expects to be able to read **val_len** bytes of the **val** buffer. However, an overflow occurs within the **flb_msgpack_gelf_value_ext** function as shown by the stacl trace, and the practical reason for this is that the **val_len** variable can contain a value larger than 48 which corresponds to sizeof(temp) (the number of bytes in temp). The very reason for this is that **sprintf** can in certain circumstances return a value larger than **sizeof(temp)** as shown in the documentation:

```
RETURN VALUE
       Upon successful return, these functions return the number of characters
       printed (excluding the null byte used to end output to strings).

       The  functions  snprintf() and vsnprintf() do not write more than
       size bytes (including the terminating null byte ('\0')).  If the output
       was truncated due to this limit, then the re-
       turn value is the number of characters (excluding the terminating
       null byte) which would have been written to the final string if enough
       space had  been  available.
```

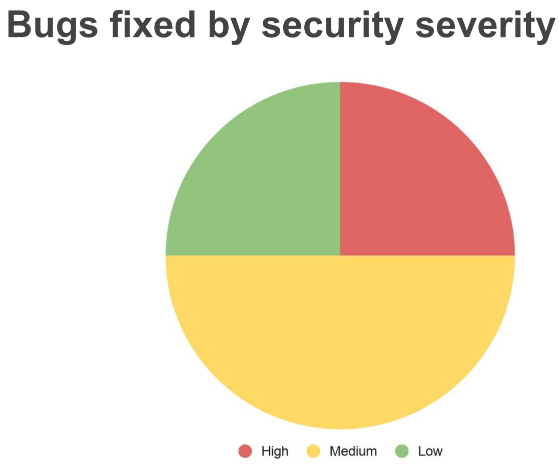
The fix of this is to ensure the value returned by **snprintf** does not extend beyond the size of **temp**:

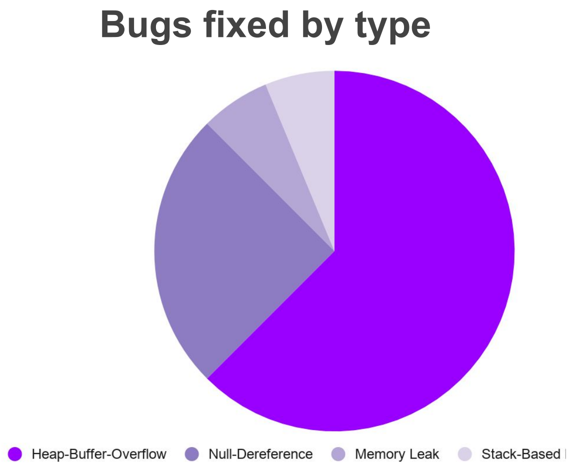
```
char temp[48] = {0};
...

else if (v->type == MSGPACK_OBJECT_POSITIVE_INTEGER) {
    val = temp;
    val_len = snprintf(temp, sizeof(temp) - 1,
                       "%" PRIu64, v->via.u64);
    /*
     * Check if the value length is larger than our string.
     * this is needed to avoid stack-based overflows.
     */
    if (val_len > sizeof(temp)) {
        return NULL;
    }

}
...

if (v->type == MSGPACK_OBJECT_EXT) {
    tmp = flb_msgpack_gelf_value_ext(s, quote, val, val_len);
}
```

We will not go into more detail with the other bugs found since this is too verbose. However, in the next section on results a link is provided to commits for each of the bugs fixed during the engagement.

Results

In this section we present the results of the engagement focusing on the bugs found as well as the bugs fixed. The following two diagrams give an overview of the bugs fixed:





Overview of bugs found and fixed

Bug id	Bug type	Commit with Fix	Security severity
5714223612821504	Heap-buffer-overflow (write)	Commit link	High
5087208312406016	Heap-buffer-overflow (read)	Commit link	Medium
5634053153488896	Null-dereference	Commit link	Low
5758082711552000	Heap-buffer-overflow (read)	Commit link	Medium
5976803149348864	Heap-buffer-overflow (read)	Commit link	Medium

4662513180082176	Heap-buffer-overflow (write)	Commit link	High
5663333098979328	Heap-buffer-overflow (read)	Commit link	Medium
5200866812100608	Heap-buffer-overflow (read)	Commit link	Medium
5177402080362496	Heap-buffer-overflow (read)	Commit link	Medium
	Memory leak	Commit link	Low
5125726487183360	Heap-buffer-overflow (write)	Commit link	High
	Heap-buffer-overflow (write)	Commit link	High
5731415213473792	Null-dereference	Commit link	Low
4907517888692224	Null-dereference	Commit link	Low
5162073690210304	Stack-based buffer overflow (read)	Commit link	Medium
5645355185864704	Null-dereference	Commit link	Medium

During the engagement we also found that two libraries embedded into Fluent Bit were outdated, in particular **msgpack** and **miniz**, both of which have been integrated into OSS-Fuzz since they were last embedded into Fluent Bit. Per advice from Ada Logics these libraries have now been updated in the Fluent bit repository.

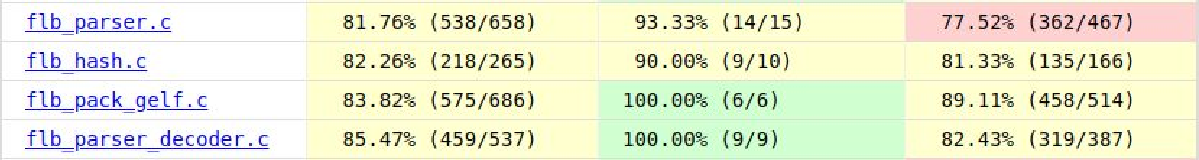
Remaining bugs to fix

There are several remaining bugs found by the fuzzers, and it is likely that more bugs will be found as the fuzzers get more time to explore the code. These have all been communicated to the Fluent Bit authors on email and through the OSS-Fuzz interface. The primary advice from the engagement is that Fluent Bit maintainers fix these bugs and continue to maintain the fuzzers of the project.

The parts of Fluent Bit that remains to be analysed

An important element of auditing is to get a good idea of how much of the target is effectively covered. This is both important from a perspective of assessing the security posture of the target as well as understanding which areas need further assurance down the line. In this section we give an estimate of this.

The figure on page 11 shows the coverage of the fuzzers achieved during a run of the fuzzers locally. A total of 47% code coverage is achieved and this includes the code in the src/aws directory which we did not consider in the engagement. Excluding the lines of the aws directory we get 54% coverage. This coverage is, however, less than what is achieved by OSS-Fuzz, where, the coverage is higher for several of the files, for example **flb_parser.c** and **flb_parser_decoder.c** files:



flb_parser.c	81.76% (538/658)	93.33% (14/15)	77.52% (362/467)
flb_hash.c	82.26% (218/265)	90.00% (9/10)	81.33% (135/166)
flb_pack_gelf.c	83.82% (575/686)	100.00% (6/6)	89.11% (458/514)
flb_parser_decoder.c	85.47% (459/537)	100.00% (9/9)	82.43% (319/387)

However, since at the time of writing a remaining PR needs to be merged into Fluent Bit we performed the local run to include all updates.

The coverage is an interim status and will increase based on

1. Fixing the remaining bugs as these bugs are hindering the fuzzers in exploring more code. This is the most important part.
2. Letting the fuzzers run for more time as the fuzzers have not yet reached their potential.

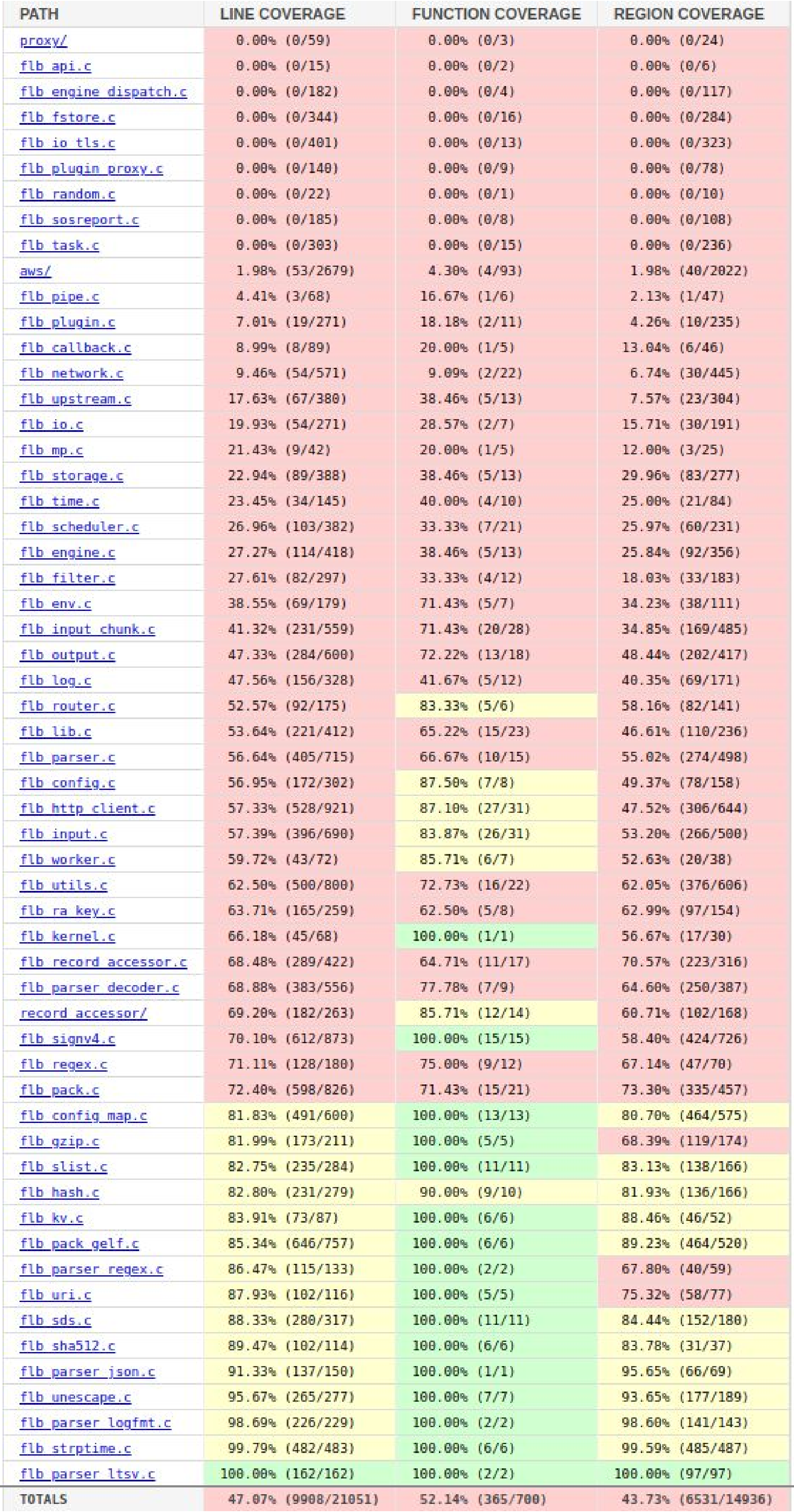
Once these have been achieved we estimate the fuzzers will increase a fair amount of coverage as most of the fuzzers still have significantly more reach than is exhibited with the current corpus.

In general, the coverage extends well through the project, the complex functions (e.g. parsing routines in particular) are thoroughly analysed and most files are covered by the fuzzers. Certain files were left out in exchange for spending more effort on more critical code, for example code such as **flb_sosreport.c** and **flb_random.c** which are used for outputting reports and a wrapper for /dev/urandom, respectively.

In general, a significant part of the code that is not covered by fuzzers can be categorised as follows:

- Code blocked by bugs that the fuzzers hit.
- Code related to handling cases where common APIs fail (such as malloc). There is a significant amount of this in the library.
- Various output-focused code (e.g. **flb_sosreport.c**)
- Error reporting code, e.g. **flb_utils_error**()

- Network-related code (we do hit http parsing routines)
- Utility code that is not part of larger code-flows and other minor functions that are used by few plugins.



PATH	LINE COVERAGE	FUNCTION COVERAGE	REGION COVERAGE
proxy/	0.00% (0/59)	0.00% (0/3)	0.00% (0/24)
flb_api.c	0.00% (0/15)	0.00% (0/2)	0.00% (0/6)
flb_engine_dispatch.c	0.00% (0/182)	0.00% (0/4)	0.00% (0/117)
flb_fstore.c	0.00% (0/344)	0.00% (0/16)	0.00% (0/284)
flb_io_tls.c	0.00% (0/401)	0.00% (0/13)	0.00% (0/323)
flb_plugin_proxy.c	0.00% (0/140)	0.00% (0/9)	0.00% (0/78)
flb_random.c	0.00% (0/22)	0.00% (0/1)	0.00% (0/10)
flb_sosreport.c	0.00% (0/185)	0.00% (0/8)	0.00% (0/108)
flb_task.c	0.00% (0/303)	0.00% (0/15)	0.00% (0/236)
aws/	1.98% (53/2679)	4.30% (4/93)	1.98% (40/2022)
flb_pipe.c	4.41% (3/68)	16.67% (1/6)	2.13% (1/47)
flb_plugin.c	7.01% (19/271)	18.18% (2/11)	4.26% (10/235)
flb_callback.c	8.99% (8/89)	20.00% (1/5)	13.04% (6/46)
flb_network.c	9.46% (54/571)	9.09% (2/22)	6.74% (30/445)
flb_upstream.c	17.63% (67/380)	38.46% (5/13)	7.57% (23/304)
flb_io.c	19.93% (54/271)	28.57% (2/7)	15.71% (30/191)
flb_mp.c	21.43% (9/42)	20.00% (1/5)	12.00% (3/25)
flb_storage.c	22.94% (89/388)	38.46% (5/13)	29.96% (83/277)
flb_time.c	23.45% (34/145)	40.00% (4/10)	25.00% (21/84)
flb_scheduler.c	26.96% (103/382)	33.33% (7/21)	25.97% (60/231)
flb_engine.c	27.27% (114/418)	38.46% (5/13)	25.84% (92/356)
flb_filter.c	27.61% (82/297)	33.33% (4/12)	18.03% (33/183)
flb_env.c	38.55% (69/179)	71.43% (5/7)	34.23% (38/111)
flb_input_chunk.c	41.32% (231/559)	71.43% (20/28)	34.85% (169/485)
flb_output.c	47.33% (284/600)	72.22% (13/18)	48.44% (202/417)
flb_log.c	47.56% (156/328)	41.67% (5/12)	40.35% (69/171)
flb_router.c	52.57% (92/175)	83.33% (5/6)	58.16% (82/141)
flb_lib.c	53.64% (221/412)	65.22% (15/23)	46.61% (110/236)
flb_parser.c	56.64% (405/715)	66.67% (10/15)	55.02% (274/498)
flb_config.c	56.95% (172/302)	87.50% (7/8)	49.37% (78/158)
flb_http_client.c	57.33% (528/921)	87.10% (27/31)	47.52% (306/644)
flb_input.c	57.39% (396/690)	83.87% (26/31)	53.20% (266/500)
flb_worker.c	59.72% (43/72)	85.71% (6/7)	52.63% (20/38)
flb_utils.c	62.50% (500/800)	72.73% (16/22)	62.05% (376/606)
flb_ra_key.c	63.71% (165/259)	62.50% (5/8)	62.99% (97/154)
flb_kernel.c	66.18% (45/68)	100.00% (1/1)	56.67% (17/30)
flb_record_accessor.c	68.48% (289/422)	64.71% (11/17)	70.57% (223/316)
flb_parser_decoder.c	68.88% (383/556)	77.78% (7/9)	64.60% (250/387)
record_accessor/	69.20% (182/263)	85.71% (12/14)	60.71% (102/168)
flb_signv4.c	70.10% (612/873)	100.00% (15/15)	58.40% (424/726)
flb_regex.c	71.11% (128/180)	75.00% (9/12)	67.14% (47/70)
flb_pack.c	72.40% (598/826)	71.43% (15/21)	73.30% (335/457)
flb_config_map.c	81.83% (491/600)	100.00% (13/13)	80.70% (464/575)
flb_gzip.c	81.99% (173/211)	100.00% (5/5)	68.39% (119/174)
flb_slist.c	82.75% (235/284)	100.00% (11/11)	83.13% (138/166)
flb_hash.c	82.80% (231/279)	90.00% (9/10)	81.93% (136/166)
flb_kv.c	83.91% (73/87)	100.00% (6/6)	88.46% (46/52)
flb_pack_gelf.c	85.34% (646/757)	100.00% (6/6)	89.23% (464/520)
flb_parser_regex.c	86.47% (115/133)	100.00% (2/2)	67.80% (40/59)
flb_uri.c	87.93% (102/116)	100.00% (5/5)	75.32% (58/77)
flb_sds.c	88.33% (280/317)	100.00% (11/11)	84.44% (152/180)
flb_sha512.c	89.47% (102/114)	100.00% (6/6)	83.78% (31/37)
flb_parser_json.c	91.33% (137/150)	100.00% (1/1)	95.65% (66/69)
flb_unescape.c	95.67% (265/277)	100.00% (7/7)	93.65% (177/189)
flb_parser_logfmt.c	98.69% (226/229)	100.00% (2/2)	98.60% (141/143)
flb_strptime.c	99.79% (482/483)	100.00% (6/6)	99.59% (485/487)
flb_parser_ltsv.c	100.00% (162/162)	100.00% (2/2)	100.00% (97/97)
TOTALS	47.07% (9908/21051)	52.14% (365/700)	43.73% (6531/14936)

Advice following engagement

In this section we outline the advice following the engagement. This is meant as guidance for how the Fluent Bit maintainers can increase security posture in the near and longer-term future.

Advice for the short term

1. Fix the remaining set of bugs captured by the fuzzers

Advice for the long term

1. Continuing to maintain the fuzzers. In particular, we do not recommend breaking them such that they no longer run in OSS-Fuzz.
2. Monitor the increase in coverage following bug fixing and identify the parts of the code that miss analysis. In case these parts are security-relevant, fuzzing support should be extended to this code.
3. Integrate an infrastructure for fuzzing the plugins of Fluent Bit. We believe the smartest approach to this is coming up with a set of

Conclusions

In this engagement we have developed sixteen fuzzers for the Fluent Bit log processor, fixed more than fifteen bugs found by these fuzzers and also integrated continuous fuzzing of Fluent Bit by way of OSS-Fuzz.

The overall perspective of the engagement is that it was successful in illustrating a complete perspective into the vulnerabilities and bugs of the Fluent Bit library. The fuzzers we developed found more than thirty bugs in the library showing that the integration of fuzzing had a significant impact on the security of Fluent Bit. However, not all of these are security relevant and many of the bugs were fixed by Ada Logics during the engagement.

There is still work to do for the Fluent Bit maintainers to ensure a high level of security in the library. It is imperative to fix the remaining set of bugs and once these are fixed the Fluent Bit library will be well-covered by fuzzer analysis. Once this has been achieved, the security posture of Fluent Bit will have made significant improvements and it is only then the security against memory corruption vulnerabilities in Fluent Bit will be high.

The main step for Fluent Bit in the short term is to fix the remaining set of bugs found by the fuzzers. Following this, the Fluent Bit maintainers need to monitor the state of the fuzzers and after letting the fuzzers run for approximately a month provide the logic necessary to cover the missing coverage in the library.