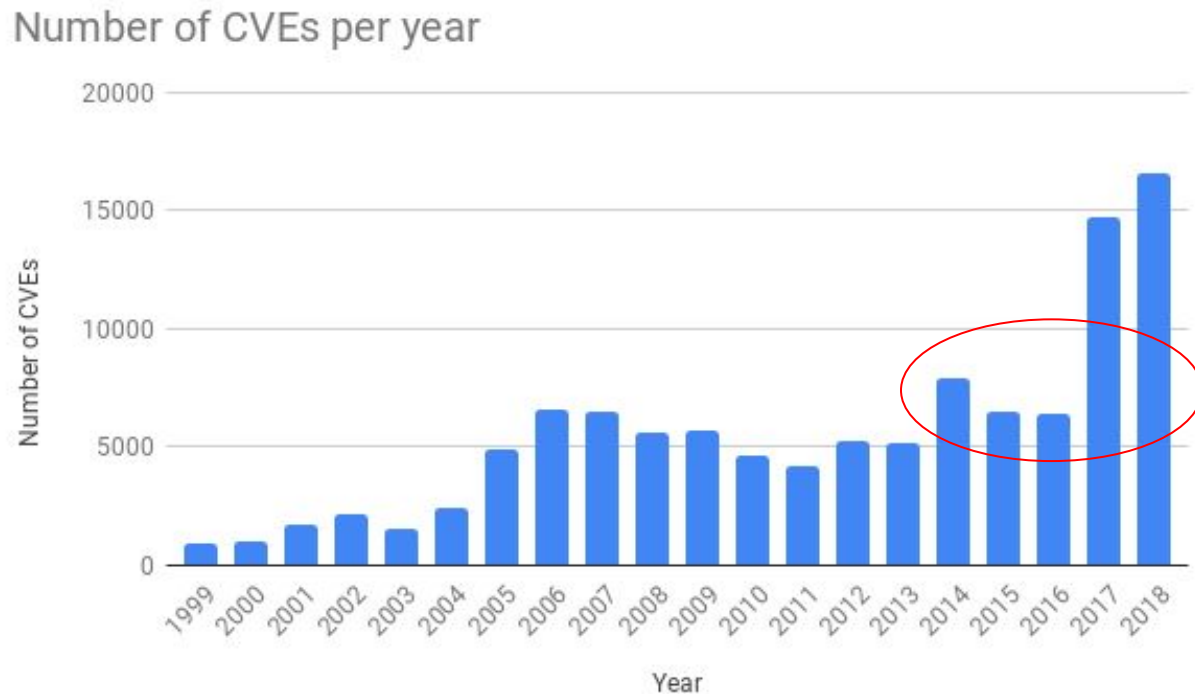


Что такое фаззинг?



Важность фаззинга



The number of vulnerabilities per year

Важность фаззинга

Trophies

As of August 2023, OSS-Fuzz has helped identify and fix over [10,000](#) vulnerabilities and [36,000](#) bugs across [1,000](#) projects.

BDU:2024-04430: Уязвимость функции treatLengthEndState() модуля asn1/ber/src/main/java/org/apache/directory/api/asn1/ber/Asn1Decoder.java Apache Directory LDAP API, позволяющая нарушителю вызвать отказ в обслуживании		Ви
Описание уязвимости	Уязвимость функции treatLengthEndState() в модуле asn1/ber/src/main/java/org/apache/directory/api/asn1/ber/Asn1Decoder.java Apache Directory LDAP API связана с отсутствием контроля вводимых пользователем данных. Эксплуатация уязвимости может позволить нарушителю, действующему удалённо, вызвать отказ в обслуживании	
Вендор	Apache Software Foundation	
Наименование ПО	Apache Directory LDAP API	
BDU:2024-04627: Уязвимость компонента парсера LDAP URL программного обеспечения Apache Directory LDAP API, позволяющая злоумышленнику вызвать отказ в обслуживании		Вид ▾
Описание уязвимости	Уязвимость компонента парсера LDAP URL программного обеспечения Apache Directory LDAP API связана с отсутствием контроля вводимых пользователем данных. Эксплуатация уязвимости может позволить нарушителю, действующему удалённо, вызвать отказ в обслуживании	
Вендор	Apache Software Foundation	
Наименование ПО	Apache Directory LDAP API	

Теперь мы готовы к
тому, чтобы написать
свой фаззер

Подготовка окружения

 Dockerfile

```
1  # base image
2  FROM registry.altlinux.org/alt/alt:p10
3
4  # set timezone
5  ENV TZ=Europe/Moscow
6  RUN ln -snf /usr/share/zoneinfo/$TZ /etc/localtime && echo $TZ > /etc/timezone
7
8  # install common system packages
9  RUN apt-get update && apt-get -y install wget git make build-essential
10
11 WORKDIR /home/fuzzer/
12 # get the source code of json-parser v1.1.0
13 RUN wget https://github.com/json-parser/json-parser/archive/refs/tags/v1.1.0.tar.gz
14
15 # install AFL++ fuzzer
16 RUN apt-get install -y AFLplusplus llvm15.0
17 ENV AFL_SKIP_CPUFREQ=1
18 ENV AFL_TRY_AFFINITY=1
19 ENV AFL_I_DONT_CARE_ABOUT_MISSING_CRASHES=1
20
21 # use llvm-15 by default
22 ENV ALTPRINT_LLVM_VERSION=15.0
23
24 ENTRYPOINT ["/bin/bash"]
25
```


Собираем образ и запускаем контейнер:

```
#bash
$ docker build -t json_fuzz_img .
$ docker run -it --name="json_fuzz" -v \
  "$(pwd)/artifacts:/home/fuzzer/artifacts" json_fuzz_img
```

Dockerfile с окружением

Собираем целевое приложение

<https://github.com/json-parser/json-parser>

 **json-parser** Public

Watch 56 Fork 289 Star 1.3k

master 1 Branch 2 Tags

Q fuzz

Add file

Code

About

Very low footprint DOM-style JSON parser written in portable ANSI C

No matches found

8 months ago 219 Commits

Скомпилируем последний релиз

```
#bash
$ cd /home/fuzzer/json-parser-1.1.0
$ ./configure
$ make
```

Написание теста

```
5 int main(int argc, char** argv) {
6     size_t len;
7     char buf[1024];
8     json_char* json;
9     json_value* value;
10
11     len = read(STDIN_FILENO, buf, 1023);
12     buf[len] = '\0';
13     json = (json_char*)buf;
14
15     value = json_parse(json, len);
16     if (value == NULL) {
17         printf("FAILED to parse json\n");
18         return 1;
19     }
20
21     printf("SUCCESS to parse json\n");
22     json_value_free(value);
23     return 0;
24 }
```

json_parse.c test

Скомпилируем наш тест:

```
#bash
$ cd /home/fuzzer/json-parser-1.1.0 && mkdir fuzz && cd fuzz
$ cp /home/fuzzer/artifacts/json_parse .
$ clang json_parse.c -I.. -L.. -ljsonparser -lm -o json_parse
```

Подготавливаем корпус

JSON-PARSER-1.1.0 [CONTAINERS]

tests

- ext-invalid-0001.json
- ext-invalid-0002.json
- ext-valid-0001.json
- ext-valid-0002.json
- ext-valid-0003.json
- invalid-0000.json
- invalid-0001.json
- invalid-0002.json
- invalid-0003.json
- invalid-0004.json
- invalid-0005.json
- invalid-0006.json
- invalid-0007.json
- invalid-0008.json
- invalid-0009.json

тесты в json-parser

```
# cd /home/fuzzer/json-parser-1.1.0/fuzz
# mkdir corpus
# cp ../tests/*.json corpus/
# ls -l corpus/

51 Dec 12 20:32 ext-invalid-0001.json
47 Dec 12 20:32 ext-invalid-0002.json
114 Dec 12 20:32 ext-valid-0001.json
```

копируем тесты в наш корпус

```
#bash
for filename in corpus/*; do
  ./json_parse < $filename;
done
```

```
FAILED to parse json
FAILED to parse json
FAILED to parse json
FAILED to parse json
FAILED to parse json
SUCCESS to parse json
SUCCESS to parse json
SUCCESS to parse json
SUCCESS to parse json
SUCCESS to parse json
SUCCESS to parse json
```

запускаем все тесты с json_parse

Добавляем мутатор

```
int main() {
    size_t len;
    char buffer[1024];

    len = read(STDIN_FILENO, buffer, 1023);
    buffer[len] = '\0';

    if (len == 0) {
        printf("Пустая строка, нечего мутировать.\n");
        return 0;
    }

    // Инициализируем генератор случайных чисел
    srand((unsigned int)time(NULL));

    // Выбираем случайный индекс для мутации
    size_t rand_index = rand() % len;

    // Генерируем случайное значение для замены
    buffer[rand_index] = (char)(rand() % 256);

    // Выводим результат
    printf("%s\n", buffer);
    return 0;
}
```

mutate.c мутатор

1. Компилируем мутатор:

```
#bash
$ clang mutate.c -o mutate
```

2. Мутируем корпус и подаём на вход *json_parse*:

```
#bash
while true; do
    for filename in corpus/*; do
        cat $filename | ./mutate | ./json_parse
    done
done
```

Улучшаем мутатор

1. Устанавливаем radamsa:

```
#bash
$ cd /home/fuzzer
$ git clone https://gitlab.com/akihe/radamsa.git
$ cd radamsa
$ make
$ make install
```

2. Тестируем с radamsa

```
#bash
while true; do
  for filename in corpus/*; do
    cat $filename | radamsa | ./json_parse
  done
done
```

```
[root@e8c4155841fd fuzz]# echo test | radamsa
0000t000sfse0t
[root@e8c4155841fd fuzz]# echo test | radamsa
00(0 e0s t
[root@e8c4155841fd fuzz]# cat corpus/valid-0002.json
[ true, false, "\u20AC\u20AD" ]

[root@e8c4155841fd fuzz]# cat corpus/valid-0002.json | radamsa
[ true, false, "\u18446744073709551617AC\u00AD" ]

[root@e8c4155841fd fuzz]# cat corpus/valid-0002.json | radamsa
[ true, false, "\u3467109550420703543AC\u20AD" ]
```

Работа radamsa мутатора

Добавляем обработку падений

Добавляем проверку на то, что возвращаемое значение больше 127 или равно 124

```
3 while true; do
4     for filename in corpus/*; do
5         # save mutated testcase to "tmp" file
6         radamsa $filename > tmp
7         # set timeout for 5s
8         cat tmp | timeout 5 ./json_parse
9         # check exit code to detect a crash
10        EXIT_CODE=$?
11        if [[ $EXIT_CODE -gt 127 || $EXIT_CODE -eq 124 ]]; then
12            # rename testcase to "crash" and exit
13            mv tmp crash
14            exit 1
15        fi
16    done
17 done
18
```

radamsa_detect.sh

exit code	Description
0	Successful exit without errors
...	...
124	in `timeout`: If the command times out, and --preserve-status is not set, then exit with status 124.
128+N	Command encountered fatal error. The N tells us which signal was received.
132	SIGILL
134	SIGABRT
139	SIGSEGV

таблица кодов возврата

Black box fuzzer



Мы реализовали black box фаззер!
Но можем ли мы пойти дальше?

Идея учёта покрытия

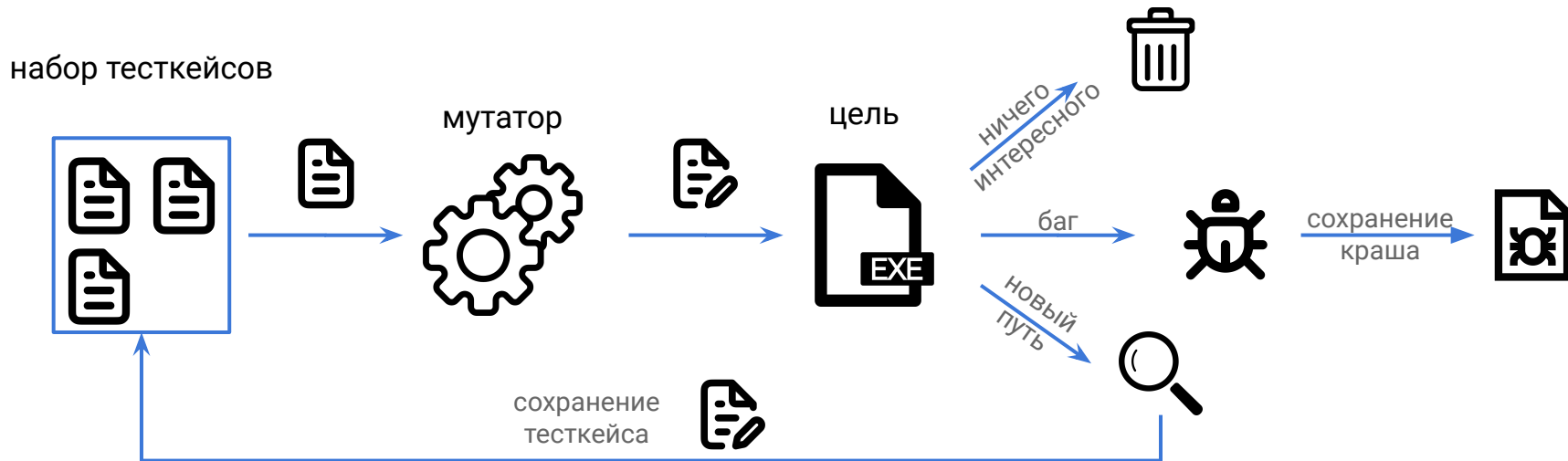
Проблемы:

- сейчас наш фаззер мутирует на основе **статического** набора входных тесткейсов.
- “шаг” наших мутаций слишком маленький, чтобы смутировать достаточно сложную json строку.
- тест должен стремиться покрыть как можно больше участков кода целевой программы, но наш тест не обращает внимание на то, открылось ли новое состояние в целевой программе.

Решение:

- Если какие-то входные данные привели к новому состоянию в программе, то логично добавить их в corpus и далее мутировать на их основе.
- основываясь на интересных входных данных, мы можем постепенно исследовать формат целевой программы.

Структурная схема серого фаззера



Учёт пройденного пути в фаззере (AFL++)

https://aflplusplus/docs/technical_details/

1. Вставляем датчики для учёта покрытия:

```
cur_location = <COMPILE_TIME_RANDOM>;  
shared_mem[cur_location ^ prev_location]++;  
prev_location = cur_location >> 1;
```

2. Поддерживаем глобальный bitmap со всеми найденными состояниями:

A -> B -> C -> D -> E (tuples: AB, BC, CD, DE)
A -> B -> D -> C -> E (tuples: AB, BD, DC, CE)] → (global map: AB, BC, CD, DE, BD, DC, CE)

3. Сравниваем трассу выполнения с глобальным bitmap:

#1: A -> B -> C -> D -> E (save testcase)

#2: A -> B -> C -> A -> E (save testcase)

#3: A -> B -> C -> A -> B -> C -> A -> B -> C -> D -> E (nevermind)

4. Учитываем количество “попаданий” в каждый отдельный edge:

edge AB: 1, 2, 3, 4-7, 8-15, 16-31, 32-127, 128+

Собираем цель с AFL++ инструментарием

Скомпилируем целевую программу с инструментарием AFL++:

```
#bash
$ cd /home/fuzzer/json-parser-1.1.0
$ CC=afl-clang-lto ./configure
$ make
$ rm -f libjsonparser.so #use static linking
```

Скомпилируем нашу обёртку с помощью AFL++ компилятора:

```
#bash
$ cd fuzz
$ afl-clang-lto json_parse.c -l.. -L.. -ljsonparser -lm -o json_parse
```


Запускаем фаззинг:

```
$ afl-fuzz -i corpus/ -o out -- ./json_parse
```

american fuzzy lop ++4.20c {default} (./json_parse) [explore]						
process timing						overall results
run time : 0 days, 0 hrs, 0 min, 24 sec						cycles done : 0
last new find : 0 days, 0 hrs, 0 min, 1 sec						corpus count : 338
last saved crash : none seen yet						saved crashes : 0
last saved hang : none seen yet						saved hangs : 0
cycle progress			map coverage			
now processing : 208.1 (61.5%)			map density : 8.19% / 67.97%			
runs timed out : 0 (0.00%)			count coverage : 4.68 bits/tuple			
stage progress			findings in depth			
now trying : havoc			favored items : 55 (16.27%)			
stage execs : 253/600 (42.17%)			new edges on : 99 (29.29%)			
total execs : 306k			total crashes : 0 (0 saved)			
exec speed : 12.7k/sec			total tmouts : 0 (0 saved)			
fuzzing strategy yields			item geometry			
bit flips : 0/248, 0/247, 0/245			levels : 14			
byte flips : 0/31, 0/30, 0/28			pending : 240			
arithmetics : 0/2156, 0/4060, 0/3780			pend fav : 5			
known ints : 0/276, 0/1130, 0/1558			own finds : 337			
dictionary : 0/0, 0/0, 0/0, 0/0			imported : 0			
havoc/splice : 224/211k, 107/87.5k			stability : 100.00%			
py/custom/rq : unused, unused, unused, unused						
trim/eff : 22.79%/836, 96.77%			[cpu000: 6%]			
strategy: explore			state: started :-)			

Собираем покрытие

После итерации фаззинга мы хотим визуализировать наработанное покрытие.

1. Собираем целевое приложение с инструментацией:

```
#bash
$ CC=clang CFLAGS="-fprofile-instr-generate -fcoverage-mapping" ./configure
```

2. Собираем обёртку с инструментацией:

```
#bash
$ clang -fprofile-instr-generate -fcoverage-mapping json_parse.c -l.. -L.. -ljsonparser -lm -o
cov_json_parse
```

3. Запускаем обёртку со всеми файлами в корпусе:

```
#bash
$ for filename in out/default/queue/*; do cat $filename | ./cov_json_parse; done
```

Создаём отчёт о покрытии

1. Индексируем сгенерированный profraw:

```
#bash
$ llvm-profdata merge default.profraw -o default.profdata
```

2. Генерируем html отчёт о покрытии:

```
#bash
$ llvm-cov show cov_json_parse --instr-profile=default.profdata \
-format=html -output-dir=report
```

3. Копируем папку с отчётом на хост и открываем отчёт в браузере

```
#bash на хосте
$ docker cp json_fuzz:/home/fuzzer/json-parser-1.1.0/fuzz/report .
$ open report/index.html
```

Coverage Report

Created: 2024-12-18 11:37

Click [here](#) for information about interpreting this report.

Filename	Function Coverage	Line Coverage	Region Coverage	Branch Coverage
fuzz/json_parse.c	100.00% (1/1)	82.35% (14/17)	80.00% (4/5)	50.00% (1/2)
json.c	66.67% (6/9)	30.09% (207/688)	28.31% (152/537)	21.13% (90/426)
Totals	70.00% (7/10)	31.35% (221/705)	28.78% (156/542)	21.26% (91/428)

Увеличиваем вероятность обнаружения бага

Сейчас обработка крешей в AFL++ почти ничем не отличается от нашей обработки с помощью кодов возврата ([слайд 10](#)). Но в нашей программе могут быть баги, которые не обязательно приводят к падению программы (например integer overflow или stack buffer overflow на один байтик). Для обнаружения подобных проблем мы можем использовать [санитайзеры](#).

Самые популярные санитайзеры:

- [AddressSanitizer](#) - memory error detector for C/C++
- [UndefinedBehaviorSanitizer](#) - undefined behavior detector
- [MemorySanitizer](#) - detector of uninitialized memory reads in C/C++

```
=====
==500665==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x6040000000b8 at pc 0x55b7777cba00 bp 0x7ffd04468310 sp 0x7ffd04468308
READ of size 4 at 0x6040000000b8 thread T0
#0 0x55b7777cb9ff in rpmReadSignature /home/fuzz/rpn/rpn/lib/signature.c:243:27
#1 0x55b777bcd2445 in main /home/fuzz/rpn/rpn/fuzz/read_signature/read_signature.c:39:24
#2 0x7fd3e2e4dd8f (/lib/x86_64-linux-gnu/libc.so.6+0x29d8f) (BuildId: c289da5071a3399de893d2af81d6a30c62646e1e)
#3 0x7fd3e2e4de3f in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x29e3f) (BuildId: c289da5071a3399de893d2af81d6a30c62646e1e)
#4 0x55b776a5e9e4 in _start (/home/fuzz/rpn/rpn/fuzz/read_signature/test_read_signature+0x1a49e4) (BuildId: a5dd97497d1b964b)

0x6040000000b8 is located 8 bytes after 42-byte region [0x604000000090,0x6040000000ba)
allocated by thread T0 here:
#0 0x55b776af880e in malloc (/home/fuzz/rpn/rpn/fuzz/read_signature/test_read_signature+0x23e80e) (BuildId: a5dd97497d1b964b)
#1 0x55b776ba5536 in rmalloc /home/fuzz/rpn/rpn/rpnio/rpmalloc.c:44:13
#2 0x55b77777c4703 in rpmReadSignature /home/fuzz/rpn/rpn/lib/signature.c:185:10
#3 0x55b777bcd2445 in main /home/fuzz/rpn/rpn/fuzz/read_signature/read_signature.c:39:24
#4 0x7fd3e2e4dd8f (/lib/x86_64-linux-gnu/libc.so.6+0x29d8f) (BuildId: c289da5071a3399de893d2af81d6a30c62646e1e)

SUMMARY: AddressSanitizer: heap-buffer-overflow /home/fuzz/rpn/rpn/lib/signature.c:243:27 in rpmReadSignature
Shadow bytes around the buggy address:
 0x03fffffffe00: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 0x03fffffffe80: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 0x03fffffffe00: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 0x03fffffffe80: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 0x040000000000: fa fa 00 00 00 00 05 fa fa 00 00 00 00 fa
=>0x040000000080: fa fa 00 00 00 00 00[02]fa fa 00 00 00 00 fa
0x040000000100: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
0x040000000180: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
0x040000000200: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
0x040000000280: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
0x040000000300: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
Shadow byte legend (one shadow byte represents 8 application bytes):
Addressable: 00
Partially addressable: 01 02 03 04 05 06 07
Heap left redzone: fa
Freed heap region: fd
Stack left redzone: f1
Stack mid redzone: f2
Stack right redzone: f3
Stack after return: f5
Stack use after scope: f8
Global redzone: f9
Global init order: fe
```

Делаем разные сборки

Запускаем скрипт по сборке целевого приложения с разными санитайзерами. А также сборка нашей обёртки:

```
#bash
$ cd /home/fuzzer/json-parser-1.1.0/fuzz
$ ./build.sh
```

```
1  #! /bin/bash
2  cd /home/fuzzer/json-parser-1.1.0
3  # afl++ build
4  CC=afl-clang-lto ./configure && make
5  mkdir afl_build && cp libjsonparser.a afl_build
6  make clean
7  # asan build
8  CC=afl-clang-lto CFLAGS="-fsanitize=address" ./configure && make
9  mkdir asan_build && cp libjsonparser.a asan_build
10 make clean
11 # ubsan build
12 CC=afl-clang-lto CFLAGS="-fsanitize=undefined" ./configure && make
13 mkdir ubsan_build && cp libjsonparser.a ubsan_build
14 make clean
15 # cov build
16 CC=clang CFLAGS="-fprofile-instr-generate -fcoverage-mapping" ./configure && make
17 mkdir cov_build && cp libjsonparser.a cov_build
18 make clean
19
20 cd /home/fuzzer/json-parser-1.1.0/fuzz
21 # afl++ build
22 afl-clang-lto json_fuzz.c -I.. -L../afl_build/ -ljsonparser -lm -o afl_fuzz
23
24 # asan build
25 afl-clang-lto json_fuzz.c -fsanitize=address -I.. -L../asan_build/ \
26 -ljsonparser -lm -o asan_fuzz
27
28 # ubsan build
29 afl-clang-lto json_fuzz.c -fsanitize=undefined -I.. -L../ubsan_build/ \
30 -ljsonparser -lm -o ubsan_fuzz
31
32 # ubsan build
33 clang json_fuzz.c -I.. -L../cov_build/ -ljsonparser -lm -o cov_fuzz
```

Ещё раз запускаем фаззинг

1. Запускаем главный инстанс:

```
#bash
$ cd /home/fuzzer/json-parser-1.1.0/fuzz
$ afl-fuzz -M main -i corpus -o out -- ./afl_fuzz
```

2. Запускаем второстепенный инстанс в новом терминале:

```
#bash
$ afl-fuzz -S asan -i corpus -o out -- ./asan_fuzz
```

3. Запускаем ещё один второстепенный инстанс в новом терминале:

```
#bash
$ afl-fuzz -S ubsan -i corpus -o out -- ./ubsan_fuzz
```

лучше использовать [tmux](#)

Узнать статус фаззинга:

```
#bash
$ afl-whatsup out
```

```
Summary stats
=====

Fuzzers alive : 3
Dead or remote : 1 (excluded from stats)
Total run time : 4 hours, 12 minutes
Total execs : 711 millions
Cumulative speed : 139416 execs/sec
Total average speed : 46472 execs/sec
Current average speed : 134607 execs/sec
Pending items : 0 faves, 140 total
Pending per fuzzer : 0 faves, 46 total (on average)
Coverage reached : 38.56%
Crashes saved : 0
Hangs saved : 0
Cycles without finds : 277/0/2149/0
Time without finds : 20075 days, 11 hours
```

В следующих сериях...

- Улучшаем скорость
- Улучшаем эффективность мутаций
- Обработка найденных падений (triage)
- Автоматизация фаззинг процесс
- Поймём, почему распались The Beatles