Анализ бинарного файла, скомпилированного обычным компилятором - Lab6

Matthew Rusakov

May 2025

Предисловие

Я собрал бинарник с помощью команды

```
gcc main.c json_fuzz.c -o json_parser -lm
```

1 Введение

В ходе анализа были использованы следующие средства:

- Valgrind — для выявления ошибок работы с памятью и утечек;
- strace — для отслеживания системных вызовов и анализа поведения процесса;
- AddressSanitizer (ASan) — для обнаружения переполнений буфера и некорректного доступа;
- Undefined Behavior Sanitizer (UBSan) — для диагностики неопределённого поведения программы.

Отчёт организован по инструментам, с отдельным описанием каждого, результатами тестов, предполагаемыми причинами ошибок и командами для воспроизведения. Отдельное внимание уделено поддержке Control-Flow Integrity (CFI), а также сравнению обычной и afl-инструментированной сборки.

2 Valgrind

Команды запуска Для анализа с помощью Valgrind использовалась команда:

```
valgrind --leak-check=full --track-origins=yes --log-file=valgrind.log ./json_parser test.json
```

Это позволило получить подробный отчёт о проблемах с памятью, включая утечки и некорректные обращения.

Найдена проблема Valgrind выявил **ошибку: Invalid read of size 8**. Это означает, что программа пыталась прочитать данные за пределами выделенного блока памяти, что может приводить к повреждению данных или сбоям.

Местоположение Проблема возникла в функции `json_value_free_ex` по адресу `0x4b6d848`, примерно на 11 байт за границей 29-байтного блока. Это точно указывает на ошибку в управлении памятью.

Анализ и причина Анализ показал, что ошибка связана с некорректной обработкой структуры при освобождении памяти. Вероятно, ошибка возникает из-за неверного расчёта размеров или неправильной итерации по массиву объектов, что приводит к выходу за границы.

3 Strace

Команды запуска Для анализа системных вызовов использовалась команда:

```
strace -o trace.log ./json_parser test.json
```

Это позволило отследить последовательность вызовов и pinpoint момент сбоя.

Найдена проблема: было зафиксировано аварийное завершение процесса с **сегментационной ошибкой (SIGSEGV)**. Это наиболее частый индикатор некорректного доступа к памяти.

Местоположение Сбой произошёл сразу после вызова `write(1, "string: testjson\n 19)`. Это указывает на то, что после печати строки программа попыталась использовать уже освобождённые или повреждённые данные.

Анализ и причина Вероятная причина заключается в логической ошибке: либо двойное освобождение памяти, либо выход за границы массива, либо повреждение кучи. Такие ошибки крайне опасны и могут быть использованы для эксплуатации программы.

4 AddressSanitizer (ASan)

Важный момент проанализовать бинарник с ASan я не смог просто так, потребовалось собрать бинарник заново, с использованием:

```
gcc -fsanitize=address -g main.c json_fuzz.c -o json_parser_asan -lm
```

Команды запуска Теперь можно проанализировать с ASan

```
./json_parser_asan test.json
```

Найдена проблема : было обнаружено **Heap-buffer-overflow**, то есть выход за пределы динамического буфера в куче.

Местоположение Ошибка была зафиксирована в файле `json_fuzz.c` на строке 200, по адресу `0x503000000098`, который на 11 байт превышал размер блока.

Анализ и причина Ошибка возникает, когда программа не проверяет границы массива, особенно при рекурсивной обработке вложенных JSON-структур. Необходима доработка логики обхода и добавление проверок размеров.

5 Undefined Behavior Sanitizer (UBSan)

Важный момент : как и для ASan, мне потребовалось заново скомпилировать файлы

```
gcc -fsanitize=undefined -g main.c json_fuzz.c -o json_parser_ubsan -lm
```

Команды запуска Теперь можно проанализировать с UBSan:

```
./json_parser_ubsan test.json
```

Найдена проблема : было выявлено **Member access within misaligned address**, то есть доступ к члену структуры по адресу, который не выровнен по требуемой границе.

Местоположение Ошибка возникла в `json_fuzz.c`, строка 179, при доступе к полю `json_value`.

Анализ и причина Скорее всего, проблема вызвана некорректным приведением типов или выделением памяти без учёта требований выравнивания. Такие ошибки могут приводить к падениям на некоторых архитектурах и ухудшению производительности.

6 Анализ поддержки Control-Flow Integrity (CFI)

Команды запуска Для проверки поддержки CFI использовались команды:

```
readelf -S json_parser | grep -i cfi
objdump -h json_parser | grep -i cfi
checksec --file=json_parser
```

Эти инструменты позволяют анализировать бинарный файл на наличие защитных механизмов.

Проблема и анализ Проверка показала, что бинарный файл собран с основными мерами защиты (RELRO, Stack Canary, NX, PIE), но **не использует CFI**. Это делает возможными некоторые типы атак, такие как ROP (Return-Oriented Programming).

Причина Поддержка CFI требует специальной сборки с компилятором Clang с флагами `-fsanitize=cfi -flto -fvisibility=hidden`, а также включения Link-Time Optimization (LTO).

7 Fuzzing

Фаззинг с бинарником, собранным обычным компилятором сильно слабее, чем с AFL, потому что он работает в режиме black-box, то есть без инструментированных хуков, с помощью которых можно найти больше проблем. Для него я использовал Radamsa и shell скрипт для запуска фаззинга:

```
#!/bin/bash

ITERATIONS=50

echo "Запускаем fuzzing $ITERATIONS итераций..."

for i in $(seq 1 $ITERATIONS); do
    radamsa test.json > fuzz_input.json

    ./json_parser fuzz_input.json

    EXIT_CODE=$?
    if [ $EXIT_CODE -ne 0 ]; then
        echo " Итерация $i: бинарник вернул код $EXIT_CODE"
        echo "Сохраняем провалившийся ввод в crash_$i.json"
        cp fuzz_input.json crash_$i.json
    fi
done
```

```
echo "Fuzzing завершен"
```

Более того, было сложно проводить фаззинг, потому что бинарник даже на валидном жсоне выдавал Segmentation fault:

```
m@hp:~/CLionProjects/4-5-6-7$ cat test.json
{"test": "testjson"}
m@hp:~/CLionProjects/4-5-6-7$ ./json_parser test.json
--------------------------------

 object[0].name = test
  string: testjson
Segmentation fault (core dumped)
m@hp:~/CLionProjects/4-5-6-7$
```

Поэтому итоги фаззинга:

- На все валидные json бинарник возвращал код 139 (Segmentation fault), хотя по логам - обрабатывал json
- На все невалидные - просто возвращал код 1 и сообщение "Unable to parse data"

8 Выводы

Проведённый анализ показал наличие ряда серьёзных проблем:

- Чтение и запись за пределами выделенной памяти;
- Утечки памяти;
- Переполнение буфера;
- Невыровненный доступ к структурам;
- Отсутствие механизмов защиты, таких как CFI;
- Segmentation fault даже после полной обработки jsona

9 Сравнение уязвимостей в обычной и afl-инструментированной сборке

Команды запуска Для анализа с использованием afl++ применялась команда:

```
afl-fuzz -i inputs -o outputs -- ./json_parser_afl @@
```

Это позволило запустить фаззинг с инструментированным бинарником.

Анализ В обеих версиях программы удалось воспроизвести ошибки:

- Heap-Buffer-Overflow в `json_value_free_ex`;
- Ошибки с повреждёнными указателями, например `free(): invalid pointer`.

Однако инструментированная сборка позволила выявить дополнительные баги, например:

- Heap-Buffer-Overflow при обработке escape-последовательностей `\uXXXX`;

- Undefined Behavior с аварийным завершением программы (SIGILL) ещё до начала полноценного фаззинга.

Вывод: Использование afl++ и санитайзеров значительно расширяет охват тестирования, позволяя находить более сложные и редкие ошибки, которые не всегда воспроизводимы в стандартной сборке.

Список литературы

[1] GitHub Link: https://github.com/MattWay224/reverse-engineering-course В этом репозитории можно найти все лабы и информацию про каждое задание в каждой лабе