

Практикум: исследование C++ приложения с помощью libFuzzer

Учебное задание для слушателя курса по безопасной разработке ПО

Цель работы: освоить базовый цикл in-process фаззинга на примере простой C++ функции: подготовить harness для libFuzzer, собрать цель с санитайзерами, запустить фаззинг, воспроизвести найденное падение, понять первопричину и предложить исправление.

Материал предназначен только для учебного стенда и собственного кода. Не запускайте фаззинг на чужом ПО, продуктах заказчика или сетевых сервисах без разрешения владельца и изоляции среды.

1. Что изучается в работе

Тема	Практический результат
libFuzzer	Слушатель запускает фаззер через функцию <code>LLVMFuzzerTestOneInput</code> .
AddressSanitizer и UndefinedBehaviorSanitizer	Слушатель получает диагностический отчет о выходе за пределы стекового массива.
Корпус входных данных	Слушатель создает минимальные seed-файлы и наблюдает рост покрытия.
Root cause analysis	Слушатель связывает crash-файл с конкретной ошибкой в проверке длины.

2. Требования к окружению

Рекомендуемая среда: Ubuntu Linux, WSL2 с Ubuntu или другая Linux-среда с Clang. В проекте уже есть файл `src/main_libfuzzer.cpp`.

```
sudo apt update
sudo apt install -y clang build-essential
```

Проверьте компилятор:

```
clang++ --version
```

Успешная проверка: команда выводит версию Clang. Для libFuzzer отдельный пакет обычно не нужен: runtime подключается через флаг `-fsanitize=fuzzer`.

Исходный файл `src/main_libfuzzer.cpp` поставляется в архиве с примером, поэтому его нужно распаковать: `sudo apt install -y unzip`, затем `unzip my-first-python-project-v02.zip`. На Ubuntu 24.04 LTS штатный Clang — версии 18, этого достаточно.

3. Исходный учебный код

Файл задания: `src/main_libfuzzer.cpp`. Он содержит функцию разбора простого бинарного формата и harness для libFuzzer.

```
#include <stdint>
#include <stddef>
#include <iostream>

static int parse_demo_format(const uint8_t* data, size_t size) {
    if (size < 5) {
        return 0;
    }

    if (data[0] != 'F' || data[1] != 'U' || data[2] != 'Z' || data[3] != '1') {
        return 0;
    }

    uint8_t declared_size = data[4];
    char name[16] = {};

    for (uint8_t i = 0; i < declared_size && 5 + i < size; ++i) {
        name[i] = static_cast<char>(data[5 + i]);
    }

    if (name[0] == 'L' && name[1] == 'I' && name[2] == 'B') {
        std::cout << "interesting libFuzzer input\n";
    }

    return 0;
}

extern "C" int LLVMFuzzerTestOneInput(const uint8_t* data, size_t size) {
    parse_demo_format(data, size);
    return 0;
}
```

3.1. Формат входных данных

Смещение	Поле	Назначение
0..3	FUZ1	Magic-значение, по которому программа понимает, что файл относится к ожидаемому формату.
4	declared_size	Заявленная длина имени. Значение полностью контролируется входным файлом.
5..N	name	Данные, которые программа копирует в локальный буфер <code>char name[16]</code> .

В коде намеренно оставлена ошибка: значение `declared_size` ограничивает цикл копирования, но не ограничивается размером массива `name`.

4. Подготовка корпуса

Создайте директории для сборки, корпуса и результатов:

```
mkdir -p build corpus findings
```

Создайте два начальных seed-файла. Первый проходит проверку magic, второй показывает фаззеру альтернативный короткий вход:

```
printf 'FUZ1\x01A' > corpus/seed1
printf 'hello' > corpus/seed2
```

Проверьте содержимое:

```
xxd -g 1 corpus/seed1
xxd -g 1 corpus/seed2
```

Задание: объясните, почему seed с `FUZ1` полезнее случайной строки для быстрого поиска глубокой ветки кода.

5. Сборка цели для libFuzzer

Соберите fuzz target с libFuzzer, AddressSanitizer и UndefinedBehaviorSanitizer:

```
clang++ -std=c++17 -g -O1 \
  -fsanitize=fuzzer,address,undefined \
  src/main_libfuzzer.cpp \
  -o build/main_libfuzzer
```

Пояснение ключевых флагов:

Флаг	Что делает
<code>-fsanitize=fuzzer</code>	Подключает libFuzzer runtime и добавляет собственный <code>main</code> фаззера.
<code>-fsanitize=address</code>	Находит ошибки памяти: выход за границы буфера, use-after-free и похожие дефекты.
<code>-fsanitize=undefined</code>	Находит часть случаев неопределенного поведения C/C++.
<code>-g -O1</code>	Сохраняет отладочную информацию и оставляет оптимизацию, приемлемую для fuzzing-сборки.

В файле для libFuzzer не должно быть обычного `main()`, если используется `-fsanitize=fuzzer`. Точкой входа является `LLVMFuzzerTestOneInput`.

6. Запуск фаззинга

Запустите короткую fuzz-сессию:

```
./build/main_libfuzzer \
  -max_total_time=30 \
  -artifact_prefix=findings/ \
  corpus/
```

Что означают параметры:

Параметр	Назначение
<code>-max_total_time=30</code>	Остановить запуск через 30 секунд. Для занятия удобно ограничивать время.
<code>-artifact_prefix=findings/</code>	Сохранять crash-файлы и другие артефакты в отдельную директорию.
<code>corpus/</code>	Директория с начальными seed-файлами и расширяющимся корпусом.

Директория из `-artifact_prefix` должна существовать до запуска. Если её нет, libFuzzer не начинает работу и печатает `ERROR: The required directory "findings/" does not exist`. Поэтому в разделе 4 создаётся `findings`, а перед повторным прогоном в разделе 11 — `findings_after_fix`.

Типичный фрагмент вывода при обнаружении ошибки:

```
runtime error: index 16 out of bounds for type 'char[16]'
ERROR: AddressSanitizer: stack-buffer-overflow
WRITE of size 1
#0 ... parse_demo_format ... src/main_libfuzzer.cpp
#1 ... LLVMFuzzerTestOneInput ... src/main_libfuzzer.cpp
Test unit written to findings/crash-<hash>
```

Успешный результат этапа: libFuzzer сохранил crash-файл в директории `findings/`, а отчет ASan указывает на строку записи в `name[i]`.

7. Воспроизведение crash

После нахождения ошибки libFuzzer сохраняет конкретный входной файл. Запустите fuzz target на этом файле напрямую:

```
ls -l findings/
./build/main_libfuzzer findings/crash-*
```

Ожидаемый результат: программа снова падает с тем же типом ошибки. Это важный этап: найденный фаззером дефект должен быть воспроизводимым.

Посмотрите байты crash-файла:

```
xxd -g 1 findings/crash-*
```

Задание: найдите в crash-файле первые четыре байта `46 55 5a 31`. Это ASCII-строка `FUZ1`. Затем определите значение пятого байта и объясните, почему оно приводит к переполнению.

8. Минимизация входа

libFuzzer умеет уменьшать crash-вход, сохраняя воспроизведение ошибки. Это упрощает анализ.

```
mkdir -p minimized
./build/main_libfuzzer \
  -minimize_crash=1 \
  -runs=10000 \
  findings/crash-* \
  -artifact_prefix=minimized/
```

Если минимизация завершилась успешно, сравните размер исходного и минимизированного файла:

```
ls -lh findings/ minimized/
xxd -g 1 minimized/crash-*
```

Минимизированный вход не обязан быть человекочитаемым. Его ценность в том, что он короче и все еще воспроизводит тот же дефект.

9. Анализ первопричины

Проблемный участок:

```
uint8_t declared_size = data[4];
char name[16] = {};

for (uint8_t i = 0; i < declared_size && 5 + i < size; ++i) {
    name[i] = static_cast<char>(data[5 + i]);
}
```

Условие `5 + i < size` проверяет только безопасность чтения из входного буфера. Оно отвечает на вопрос: есть ли еще байт во входных данных?

Но в условии нет проверки безопасности записи в `name`. Для массива `char name[16]` допустимы индексы `0..15`. Когда `i == 16`, запись `name[i]` выходит за пределы массива.

Корневая причина: длина из недоверенного входа используется как граница записи в фиксированный стековый буфер без ограничения размером буфера назначения.

10. Исправление

Исправление должно учитывать три ограничения: заявленную длину, доступные байты входа и размер буфера назначения.

```
#include <algorithm>

uint8_t declared_size = data[4];
char name[16] = {};

size_t available_size = size - 5;
size_t copy_size = std::min<size_t>(
    static_cast<size_t>(declared_size),
    std::min(sizeof(name), available_size)
);

for (size_t i = 0; i < copy_size; ++i) {
    name[i] = static_cast<char>(data[5 + i]);
}
```

После исправления пересоберите `target` и проверьте старый `crash`-файл:

```
clang++ -std=c++17 -g -O1 \
  -fsanitize=fuzzer,address,undefined \
  src/main_libfuzzer.cpp \
  -o build/main_libfuzzer

./build/main_libfuzzer findings/crash-*
```

Ожидаемый результат после исправления: старый `crash`-файл больше не вызывает падение. Затем нужно повторно запустить фаззинг, чтобы проверить, не осталось ли похожих дефектов.

11. Повторная проверка после исправления

Создайте директорию для артефактов повторного прогона (`libFuzzer` требует, чтобы она уже существовала) и запустите фаззинг на более длительное время:

```
mkdir -p findings_after_fix
./build/main_libfuzzer \
  -max_total_time=60 \
  -artifact_prefix=findings_after_fix/ \
  corpus/
```

Для учебной работы достаточно зафиксировать результат: падений нет в течение заданного времени или найден новый дефект, который требует отдельного анализа.

12. Что нужно сдать преподавателю

1. Команду сборки fuzz target.
2. Команду запуска libFuzzer.
3. Фрагмент отчета ASan с типом ошибки и строкой исходного кода.
4. Hexdump crash-файла.
5. Краткое объяснение первопричины дефекта.
6. Исправленный фрагмент кода.
7. Результат повторной проверки после исправления.

13. Контрольные вопросы

1. Почему libFuzzer требует функцию `LLVMFuzzerTestOneInput` ?
2. Чем in-process fuzzing отличается от запуска программы через файлы и аргументы командной строки?
3. Почему seed `FUZ1\x01A` помогает быстрее пройти проверку magic?
4. Какую ошибку показывает AddressSanitizer в этом задании?
5. Почему проверка `5 + i < size` не защищает массив `name` ?
6. Какие три ограничения нужно учитывать при безопасном копировании?
7. Зачем воспроизводить crash отдельно от основной fuzz-сессии?
8. Для чего нужна минимизация crash-файла?

14. Краткая памятка команд

```
sudo apt install -y clang build-essential unzip

mkdir -p build corpus findings
printf 'FUZ1\x01A' > corpus/seed1
printf 'hello' > corpus/seed2

clang++ -std=c++17 -g -O1 \
  -fsanitize=fuzzer,address,undefined \
  src/main_libfuzzer.cpp \
  -o build/main_libfuzzer

./build/main_libfuzzer -max_total_time=30 -artifact_prefix=findings/ corpus/
./build/main_libfuzzer findings/crash-*
xxd -g 1 findings/crash-*

mkdir -p minimized
./build/main_libfuzzer -minimize_crash=1 -runs=10000 findings/crash-* -artifact_prefix=minimized/

mkdir -p findings_after_fix
./build/main_libfuzzer -max_total_time=60 -artifact_prefix=findings_after_fix/ corpus/
```

Версия методички: 2026-10-09. Учебный проект: `main_libfuzzer.cpp`. Команды проверены на WSL2 с Ubuntu 24.04.4 LTS, clang 18.1.3: цель собирается, падение находится в пределах 30 секунд, старый crash-файл перестаёт воспроизводиться после исправления.