Лабораторная работа №4
Фаззинг-тестирование

Цель работы

Получить навыки фаззинг-тестирования программ

Особенности использование libFuzzer

Инструмент libFuzzer предназначен для проведения фаззинг-тестирования отдельных функций, но такое применение обычно встречается только в ознакомительных примерах, и функций различных библиотек, как статических, так и динамических. Наиболее часто этот инструмент используется разработчиками программного обеспечения, так как предполагает хорошее знание структуры исходных текстов и структуры данных, принимаемых на вход исследуемыми функциями.

Общий порядок работы с libFuzzer можно описать так:

1) изучение исходных текстов цели (цель - программа, подвергаемая фаззинг-тестированию) и выбор наиболее подходящей для фаззинга функции: в идеальном случае она должна принимать на вход символьный/байтовый набор, либо структуру данных, зависящую от поступающих входных данных от пользователя;
2) написание функции-обертки (она же называется точка входа libFuzzer), которая:
   а) принимает данные непосредственно от libFuzzer (в виде набора байт);
   б) формирует необходимый контекст для вызова целевой функции;
   в) обрабатывает результаты работы функции и классифицирует их на штатные/нештатные;
3) компиляция цели и обертки с инструментацией кода;
4) запуск процесса фаззинг-тестирования;
5) анализ полученных результатов.

Запуск фаззинга на тестовой программе

Напишем функцию, содержащую потенциальную ошибку (файл badFunction.cpp). При выполнении ряда условий для входных данных имитируется ошибка, приводящая к аварийному завершению программы. В нашем случае это будет просто возбуждение исключения. Приведем пример такой функции:

```
void badFunction(uint8_t *data, std::size_t size) {
    if (size>4) {
        if (data[0]=='E') {
            if (data[1]=='r') {
                if (data[2]=='r') {
                    if (data[3]=='o') {
                        if (data[4]=='r') {
                            throw std::exception();
                        }
                    }
                }
            }
        }
    }
}
```

Теперь напишем функцию-обертку (файл wrapper.cpp), которая является входной точкой для фаззинга нашей «плохой» функции:

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

Формат функции является стандартным для libFuzzer:

1) имя функции всегда **LLVMFuzzerTestOneInput**;
2) тип возвращаемого значения всегда **int**;

3) входные параметры всегда одного типа:

а) **const uint8_t *data** – массив байт, поступающий от фаззера. Мутируется при каждом такте работы фаззера;

б) **size_t size** – количество байт в массиве.

При этом, функция вызывается фаззером согласно соглашениям по вызову функций языка Си, поэтому если фаззинг выполняется в программах на языке Си++, перед функцией необходимо указать модификатор extern "C".

В файл wrapper.cpp необходимо добавить прототип нашей «плохой» функции, так как для нее у нас нет заголовочного файла.

При проведении фаззинг-тестирования происходит многократный вызов функции LLVMFuzzerTestOneInput с различным содержимым массива data. Если для тестового сценария (в данном случае это одна порция данных, передаваемой целевой функции) достигается новое покрытие кода (отслеживается за счет инструментации), то он считается интересным и добавляется в корпус (corpus) фаззера.

Стоит отметить, что в данном случае рассматривается достаточно простая цель, принимающая на вход набор байт, поэтому никакой дополнительный контекст не формируется. На практике, как правило, функции бывают сложнее, поэтому перед их вызовом проделываются дополнительные действия.

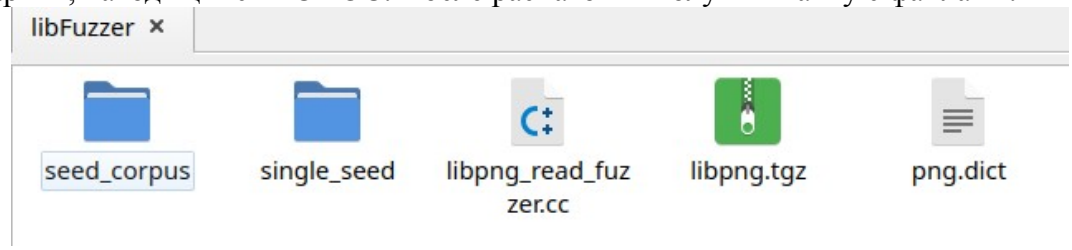
Скомпилируем написанные функцию и обертку компилятором clang. В команду сборки добавляется флаг «-fsanitize=fuzzer», позволяющий проинструментировать цель для получения обратной связи по покрытию. Команда сборки:

```
clang -fsanitize=fuzzer badFunction.cpp wrapper.cpp -o test_fuzzer
```

В каталоге должен появиться исполняемый файл для фаззинг-тестирования «test_fuzzer». После запуска файла фаззер быстро завершает работу. Результаты работы сохраняются в файл с именем «crash-*». В нем находятся данные, которые привели к аварийному завершению программы.

Фаззинг реальной библиотеки

Рассмотрим фаззинг библиотеки libpng, написанной на языке Си. Для этого скачаем и распакуем архив, находящийся в ЭИОС. После распаковки получим папку с файлами:



Файл libpng.tgz содержит архив исследуемой библиотеки, файл libpng_read_fuzzer.cc содержит уже готовую функцию-обертку, файл ping.dict содержит словарь для тестирования файлов png, а в папке seed_corpus лежит набор файлов корпуса фаззинга.

Сперва необходимо распаковать библиотеку. Это делается из консоли командой

```
tar xvf libpng.tgz
```

Библиотека распакуется в папку libpng.

Затем необходимо подготовить библиотеку к сборке для фаззинг тестирования. Для этого надо перейти в папку библиотеки и отключить логирование, так как логирование сильно замедлит фаззинг тестирование. Это делается так:

```
cd libpng
sed -i scripts/pnglibconf.dfa -e "s/option STDIO/option STDIO disabled/"
```

Далее надо выполнить сборку библиотеки для фаззинг-тестирования. Для этого сперва создается файл конфигурации:

```
autoreconf -f -i
```

Затем создается переменная окружения FUZZ_CXXFLAGS, в которой указаны флаги для фаззинга:

```
export FUZZ_CXXFLAGS="-O2 -fno-omit-frame-pointer -g \
-fsanitize=address,fuzzer-no-link"
```

Здесь -O2 задает уровень оптимизации для повышения быстродействия; -fno-omit-frame-pointer указывает, что из процессов оптимизации нужно исключить регистр указателя базы, так как он используется санитайзерами; -g добавляет отладочную информацию; -fsanitize=address,fuzzer-no-link добавляет санитайзер памяти address, а также указывает, что при сборке библиотеки к ней пока еще не надо прилинковывать фаззер.

Далее проект конфигурируется и собарается:

```
./configure CC="clang" CFLAGS="$FUZZ_CXXFLAGS"
make
```

После успешной сборки проекта необходимо выйти в родительскую папку:

```
cd ..
```

Теперь можно приступать к сборке фаззера:

```
clang++ libpng_read_fuzzer.cc -O2 -fno-omit-frame-pointer -g \
-fsanitize=address,fuzzer -I libpng  libpng/.libs/libpng16.a -lz -o libpng_fuzzer
```

После сборки у нас появится исполняемый файл фаззера libpng_fuzzer. При запуске фаззеру можно указать различные флаги, а также несколько папок с копрпусами. Рассмотрим некоторые из флагов:

-max_len=*число* – задает максимальный размер мутируемых данных;

-max_total_time=*число* – время фаззинга в секундах;

-print_final_stats =0|1 – печать финальной статистики фаззинга;

-dict=*файл* – словарь для фаззинга;

-timeout=*число* – время обработки одного набора данных, которое не считается зависанием программы.

Если папок с корпусами не указать, то фаззер начнет работу с пустым корпусом, а создаваемые при работе сиды будут помещаться во временную папку и после окончания работы будут удалены. Если указать несколько папок с корпусами, то новые сиды будут записываться в первую из них. Поэтому иногда при фаззинге первой указывают пустую папку, чтобы фаззер в нее добавлял новые сиды и сформировал корпус. Пример запуска фаззера с флагами и корпусами:

```
./libpng_fuzzer -max_len=2048 -max_total_time=180 -dict=png.dict -timeout=10 \
-print_final_stats=1  empty_corpus single_seed
```

В папке empty_corpus будет создан новый корпус. А корпус в папке single_seed останется без изменений. По завершении работы фаззера (в примере 3 минуты) будет выдана финальная статистика:

```
Done 9866153 runs in 181 second(s)
stat::number_of_executed_units: 9866153
stat::average_exec_per_sec:     54509
stat::new_units_added:          596
stat::slowest_unit_time_sec:    0
stat::peak_rss_mb:              561
```

Кроме статистики интерес представляет последняя строка результатов работы:

```
#9866153        DONE   cov: 235 ft: 446 corp: 106/17Kb lim: 2048 exec/s: 54509 rss: 561Mb
```

В этих данных стоит обратить внимание на следующие показатели:

- «number_of_executed_units» – количество вызовов функции LLVMFuzzerTestOneInput. Чем

это значение больше, тем лучше для фаззинга. Сэмплы в идеальном случае должны быть минимальной длины, тогда будет достигнута максимальная скорость;

- «average_exec_per_second» – среднее количество выполнений в секунду. Вычисляется отношение общего количество запусков к общему времени выполнения;

- «cov: <n>» – достигнутое покрытие в единицах libFuzzer. Основная цель фаззинга – достижение максимального покрытия;

- «corp: <k>/<s>» – количество сэмплов в корпусе фаззера и занимаемый ими размер. Показатель косвенно связан с покрытием: сэмплы, приводящие к приросту покрытия, заносятся в корпус. Соответственно, чем больше будет покрытие, тем больше будет корпус.

Наиболее важные результаты работы фаззера следует занести в таблицу 1 для дальнейшего анализа.

Таблица 1 - Финальная статистика проведения фаззинг-тестирования

Запуск \ Статистика	Количество запусков	Скорость работы	Покрытие	Корпус
Без корпуса и без словаря				
Без корпуса и со словарем				
С корпусом и без словаря				
С корпусом и со словарем				

Кроме статистики фаззер еще выдает рекомендуемый словарь.

Задание к лабораторной работе

1. Выполнить фаззинг-тестирование файла badFunction.cpp. Посмотреть и проанализировать результат.
2. Изменить файл badFunction.cpp таким образом, чтобы последовательность символов, приводящих к аварийному завершению, совпадала с именем студента (в тестовом примере последовательность - «Error»).
3. Выполнить фаззинг-тестирование измененного файла badFunction.cpp. Посмотреть и проанализировать результат.
4. Подготовить библиотеку libpng к фаззинг-тестированию.
5. Выполнить фаззинг-тестирование с пустым корпусом и без словаря. Время ограничить тремя минутами, таймаут десятью секундами, максимальный размер мутируемых данных (сэмплов) 2048 байтами. Результат занести в таблицу 1.
6. Выполнить фаззинг-тестирование с пустым корпусом, но со словарем. Ограничения взять такие же, как в предыдущем пункте. Результат занести в таблицу 1.
7. Выполнить фаззинг-тестирование с корпусом, но без словаря. В качестве корпуса использовать данные папки seed_corpus. Ограничения взять такие же, как в предыдущем пункте. Результат занести в таблицу 1.
8. Выполнить фаззинг-тестирование с корпусом, и со словарем. В качестве корпуса использовать данные папки seed_corpus. Ограничения взять такие же, как в предыдущем пункте. Результат занести в таблицу 1.
9. Проанализировать результаты и сделать выводы.