

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ



**«Московский государственный технический
университет
имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)**

ФАКУЛЬТЕТ «Информатика и системы управления»
КАФЕДРА «Защита информации»

ОТЧЁТ

на тему:

«Развёртка фаззера AFL»

(по курсу «системное программное обеспечение»)

Выполнил:

Студент ИУ10-36

Большедворский Сергей Павлович

Преподаватель:

Лысюк Сергей Владиславович, ИУ10

Москва, 2022

Оглавление

| | |
|---|----|
| 1. Введение..... | 3 |
| 1.1 Способы анализа кода, отличие динамического от статического анализа | 3 |
| 1.2 Фаззеры | 4 |
| 2. American Fuzzy Lop (AFL) | 5 |
| 2.1 Понятие, особенности, преимущества..... | 5 |
| 2.2 Установка стенда | 7 |
| 2.3 Практическое использование фаззера, обнаружение бага CVE-2015-3145..... | 8 |
| 2.4 Описание AFL-fuzzer UI | 14 |
| 2.5 Анализ вывода программы | 22 |
| 3. Вывод | 23 |

1. Введение

1.1 Способы анализа кода, отличие динамического от статического анализа

Для выявления уязвимостей в программе используются инструменты статического и динамического анализа исходного кода. В случае статического анализа поиск возможных ошибок осуществляется без запуска исследуемой программы, например, по исходному коду приложения. Обычно при этом строится абстрактная модель программы, которая, собственно, и является объектом анализа. Следует подчеркнуть следующие характерные особенности статического анализа:

- Возможен отдельный анализ отдельно взятых фрагментов программы (обычно отдельных функций или процедур), что дает достаточно эффективный способ борьбы с нелинейным ростом сложности анализа.
- Возможны ложные срабатывания, обусловленные либо тем, что при построении абстрактной модели некоторые детали игнорируются, либо тем, что анализ модели не является исчерпывающим.
- При обнаружении дефекта возникают, во-первых, проблема проверки истинности обнаруженного дефекта и, во-вторых, проблема воспроизведения найденного дефекта при запуске программы на определенных входных данных.

В отличие от статического анализа, динамический анализ осуществляется во время работы программы. При этом:

- Для запуска программы требуются некоторые входные данные.

- Динамический анализ обнаруживает дефекты только на трассе, определяемой конкретными входными данными; дефекты, находящиеся в других частях программы, не будут обнаружены.
- В большинстве реализаций появление ложных срабатываний исключено, так как обнаружение ошибки происходит в момент ее возникновения в программе; таким образом, обнаруженная ошибка является не предсказанием, сделанным на основе анализа модели программы, а констатацией факта ее возникновения.

Автоматическое тестирование в первую очередь предназначено для программ, для которых работоспособность и безопасность при любых входных данных являются наиважнейшими приоритетами: веб-сервер, клиент/сервер SSH, sandboxing, сетевые протоколы.

1.2 Фаззеры

Фаззинг — техника тестирования программного обеспечения, часто автоматическая или полуавтоматическая, заключающаяся в передаче приложению на вход неправильных, неожиданных или случайных данных. Предметом интереса являются падения и зависания, нарушения внутренней логики и проверок в коде приложения, утечки памяти, вызванные такими данными на входе. Фаззинг является разновидностью выборочного тестирования, часто используемого для проверки проблем безопасности в программном обеспечении и компьютерных системах.

Существуют две основные техники фаззинга — это мутационное и порождающее тестирования. При мутационном тестировании генерация последовательностей происходит на основе заранее определенных данных и шаблонов. Именно они составляют стартовый корпус фаззера. Изменяя

байт за байтом значения на входе и проверяя работу программы, фаззер может делать выводы об успешности тех или иных «мутаций», чтобы в следующем раунде сгенерировать более эффективные последовательности. Как видите, сама концепция достаточно простая. Но за счет того, что количество итераций достигает сотен и тысяч миллионов (время тестирования при этом составляет несколько суток даже на мощных машинах), фаззеры находят в программах самые нетривиальные ошибки. В свою очередь, порождающее тестирование — это более продвинутая техника фаззинга, которая предполагает построение грамматик входных данных, основанное на спецификациях. Это могут быть как файлы различных форматов, так и сетевые пакеты в протоколах обмена. В данном случае наши результаты должны соответствовать заранее определенным правилам. Порождающее тестирование сложнее мутационного в реализации, но и вероятность успеха здесь гораздо выше.

Соответственно фаззер — инструмент для обеспечения выполнения процесса фаззинга. Существует большое количество различных фаззеров, предоставляющих широкий функционал динамического анализа ПО: Radamsa, Honggfuzz, Libfuzzer, OSS-Fuzz, Sulley Fuzzing Framework, boofuzz, BFuzz, PeachTech Peach Fuzzer и другие. В условиях данной работы мы остановимся на одном из наиболее популярных, простых и прогрессивных решений - AFL.

2. American Fuzzy Lop (AFL)

2.1 Понятие, особенности, преимущества

[American Fuzzy Lop\(AFL\)](#) — средство динамического анализа программ (фаззер), основным отличием которого является инструментация

кода на этапе компиляции (про инструментацию будет чуть позже), производительность и ориентированность на практическое применение.

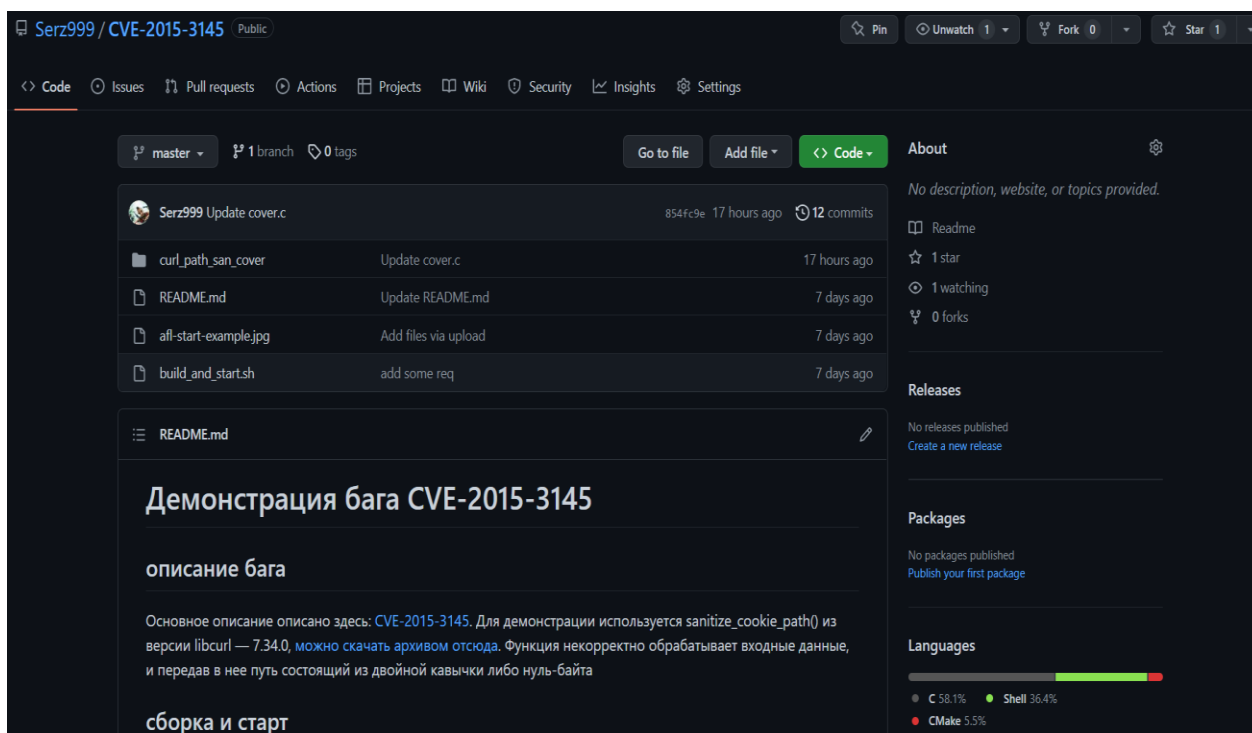
Главное преимущество этой программы заключается в том, что развертывание стенда происходит без особой настройки. Она была создана на основе множества исследований о том, как работают лучшие фаззеры и какие результаты наиболее полезны для тестировщиков. Программа также подойдет для минимизации времени, необходимого для компиляции запроса и получения результатов, при минимальном воздействии на систему, где бы она ни была установлена. На самом деле, разработчик American Fuzzy Lop настолько уверен в способности фаззера работать без вмешательства пользователя, что не добавил почти никаких элементов управления. Пользователи видят приятный интерфейс в стиле ретро, где видно, что делает фаззер и результаты его активности. Несмотря на то, что разработчик уверен в способности American Fuzzy Lop находить любые ошибки в тестируемых программах, инструмент также совместим и с другими фаззерами. Он может генерировать данные тестирования, которые при необходимости будут загружены в другие, более специализированные, инструменты фаззинга. Это потенциально способно повысить эффективность других инструментов, а также сократить время выполнения тестирования.

Обычно фаззер стартует приложение в новом процессе, затем подает ему на вход тестовые данные в `stdin` или используя временный файл, если процесс упадет — AFL это заметит и запишет поданные данные в заданную директорию `crashes`. Для генерации тестов необходим так называемый `corpus`, представляющий собой набор тестовых данных, которые обрабатывает приложение.

2.2 Установка стенда

Ниже будут приведены фрагменты кода из самописного bash-скрипта автоматической сборки, развёртки и запуска проекта (`build_and_start.sh`), а так же фрагменты кода обёртки над функцией из исходного кода [libcurl версии 7.34.0](#), написанный на Си (`cover.c`).

Проект, готовый для развёртки фаззера, для удобства, был расположен в отдельном репозитории на GitHub: [Демонстрация бага CVE-2015-3145](#). Отсюда же можно скачать и установить стэнд, клонировав себе локально репозиторий и запустив скрипт `./build_and_start.sh`. Помимо прочего репозиторий содержит более удобный и сжатый материал по теме практического использования AFL-фаззера для нахождения ошибок исполняемой программы при передаче валидных входных данных, представленный в файле `README.md`



2.3 Практическое использование фаззера, обнаружение бага CVE-2015-3145

Существует два способа фаззить приложение: целиком и по отдельным(юнит) модулям. Для наглядности, в качестве примера, точно заставим упасть заведомо неисправную функцию `curl sanitize_cookie_path()` при передаче на вход заранее подготовленного HTTP ответа веб-сервера. `curl` — это специальная программа или утилита, которая позволяет делать различные сетевые запросы по различным протоколам данных. Смысл у этой программы единственный: сделать какой-то запрос и получить ответ. `curl` это клиент, который выполняет запросы к какому-то серверу. Так выглядит исходник неисправной функции:


```

static char *sanitize_cookie_path(const char *cookie_path)
{
    size_t len;
    char *new_path = strdup(cookie_path);
    if(!new_path)
        return NULL;

    /* some stupid site sends path attribute with '\"'. */
    if(new_path[0] == '\\') {
        memmove((void *)new_path, (const void *)(new_path + 1), strlen(new_path));
    }
    if(new_path[strlen(new_path) - 1] == '\\') {
        new_path[strlen(new_path) - 1] = 0x0;
    }

    /* RFC6265 5.2.4 The Path Attribute */
    if(new_path[0] != '/') {
        /* Let cookie-path be the default-path. */
        free(new_path);
        new_path = strdup("/");
        return new_path;
    }

    /* convert /hoge/ to /hoge */
    len = strlen(new_path);
    if(1 < len && new_path[len - 1] == '/') {
        new_path[len - 1] = 0x0;
    }

    return new_path;
}

```

Данная функция некорректно обрабатывает входные данные, и передав в нее путь, состоящий из двойной кавычки либо нуль-байта (в качестве примера нуль-байта может послужить символ конца классической

Си-строки), libcurl назначит нуль-байт по отрицательному указателю массива char* new_path, и испортит память на куче.

Напишем простенькую обёртку, в которой вызовем и передадим аргументы из тестового файла в функцию sanitize_cookie_path(), которая имеет заранее известную уязвимость, описанную в [CVE-2015-3145](#).

```
int main(int argc, char **argv)
{
    unsigned char buf[2048];
    char *res = NULL;

    assert(argc == 2);

    FILE *f = fopen(argv[1], "rb");
    assert(f);

    size_t len = fread(buf, 1, sizeof(buf), f);
    buf[len] = 0x00;
    if (len == 0 || strlen(buf) == 0) {
        return 0;
    }

    printf("read = %zu\n", len);
    printf("in = %s\n", buf);

    res = sanitize_cookie_path(buf);

    if (res) {
        printf("res = %s\n", res);
        free(res);
    }
    return 0;
}
```

Начнем с установки необходимых компонентов для компиляции кода: gcc и clang

```
#install gcc and clang compailers
sudo apt install gcc
sudo apt install clang
```

Установим, распакуем и соберём AFL-фаззер с помощью make:

```
#install AFL
wget http://lcamtuf.coredump.cx/afl/releases/afl-2.52b.tgz
tar -xzf afl-2.52b.tgz; rm afl-2.52b.tgz

#build the afl-fuzzer
cd afl-2.52b
make; sudo make install
```

Мы могли бы использовать gcc компилятор по умолчанию, который поставляется с AFL, но он гораздо медленнее по сравнению с другими предлагаемыми компиляторами, которые поставляются с ним. AFL использует LLVM возможности для ускорения процесса фаззинга. LLVM (Low Level Virtual Machine) — это универсальная система анализа, трансформации и оптимизации программ или, как её называют разработчики, «compiler infrastucture». В основе LLVM лежит промежуточное представление кода, над которым можно производить трансформации во время компиляции, компоновки (линковка) и выполнения. Из этого представления генерируется оптимизированный машинный код для целого ряда платформ, как статически, так и динамически. Для AFL мы можем включить режим LLVM, собрав модуль с помощью следующих команд:

```
#enable llvm_mode for fast fuzzing
cd llvm_mode
sudo apt-get install llvm-dev llvm
make
cd ..
make; sudo make install
```

Настало время поднять тему статической инструментации исходного кода, то есть инструментации на этапе компиляции программы. В общем случае под инструментацией понимают возможность отслеживания или установления количественных параметров уровня производительности программного продукта, а также возможность диагностировать ошибки и записывать информацию для отслеживания причин их возникновения. Проще говоря это процесс модификации исследуемой программы с целью ее дальнейшего анализа.

Для фаззинга программы, используя AFL, обязательно требуется инструментировать исходный код приложения, что так же прописано в официальной документации фаззера.

На самом деле процесс фаззинга может занимать довольно значительное время: от одного дня до нескольких недель и даже месяцев, и не факт, что получится обнаружить уязвимость во время работы программы. Чтобы значительно повысить шансы на успех необходимо сильно повысить чувствительность фаззинга, используя встроенный функционал [clang ASAN - Address Sanitizer](#). AddressSanitizer — это быстрый детектор ошибок памяти. Он состоит из инструментального модуля компилятора и динамической ран-тайм библиотеки. Address Sanitizer позволит почти гарантированно вызвать краш, связанный с порчей памяти на куче при передаче двойных кавычек в аргумент функции `sanitize_cookie_path()`. Для активации функций Address Sanitizer'a используется флаг `-fsanitize` при инструментации обёртки исходного кода под AFL. Так выглядят команды:

```
#cd to project root folder
cd ..

#create compile cover.c to binary file with afl-clang-fast instrumentation for fuzzing
cd curl_path_san_cover
afl-clang-fast -g -fsanitize=address cover.c -o cover
cd ..
```

Создадим тестовый файл со строкой, содержащий url с двойной кавычкой “/xxx/” и положим в папку inputs на вход для фаззера. Затем обязательно создадим папку out для вывода итогов прогонки тестовых данных, в последствии можно будет зайти и просмотреть содержимое этого каталога:

```
#create the test corpus for fuzzer
mkdir inputs
touch inputs/test_url.txt
echo "\"/xxx/\"" > inputs/test_url.txt

mkdir out
```

Стоит подметить что AFL, — это умный фаззер. Он изменяет входные данные в начале фаззинга для создания новых тестовых случаев, которые, по его мнению, приведут к открытию новых путей. В нашем случае, очевидно, что лучший вариант для обнаружения ошибки — это HTTP-ответ с двойными кавычками. Для демонстрации корректной работы фаззера остаётся только добавить переменную окружения для использования функций clang-ASAN `AFL_USE_ASAN=1`, запустить `afl-fuzz` и передать ему требуемые параметры:

```
#start fuzzer
AFL_USE_ASAN=1 afl-2.52b/afl-fuzz -m none -i inputs -o out ./curl_path_san_cover/cover @@
```

`curl_path_san_cover/cover` — это путь до бинарного файла тестируемой программы, заранее инструментированного под AFL. Параметр `-m none` отключит лимит памяти, после `-i/-o` указывается имя

папки с данными на вход/выход соответственно, а @@ будет заменяться именем временного файла при фаззинге, если не задать этот параметр — тестовые данные будут подаваться в stdin. Почти сразу после запуска AFL обнаружит crash и сгенерирует тестовый вход в директории out/crashes. В итоге откроется окно, на котором можно увидеть информацию о процессе фаззинга:

```

american fuzzy lop 2.52b (cover)

process timing |-----| overall results
  run time : 0 days, 0 hrs, 0 min, 28 sec | cycles done : 7
  last new path : 0 days, 0 hrs, 0 min, 28 sec | total paths : 6
  last uniq crash : 0 days, 0 hrs, 0 min, 28 sec | uniq crashes : 1
  last uniq hang : none seen yet | uniq hangs : 0
-----|-----|
cycle progress |-----| map coverage
  now processing : 1 (16.67%) | map density : 0.02% / 0.03%
  paths timed out : 0 (0.00%) | count coverage : 1.00 bits/tuple
-----|-----|
stage progress |-----| findings in depth
  now trying : havoc | favored paths : 6 (100.00%)
  stage execs : 24/128 (18.75%) | new edges on : 6 (100.00%)
  total execs : 34.9k | total crashes : 503 (1 unique)
  exec speed : 1205/sec | total tmouts : 0 (0 unique)
-----|-----|
fuzzing strategy yields |-----| path geometry
  bit flips : 1/224, 0/218, 1/206 | levels : 2
  byte flips : 0/28, 0/22, 0/10 | pending : 0
  arithmetics : 1/1554, 0/283, 0/71 | pend fav : 0
  known ints : 1/126, 0/533, 0/405 | own finds : 5
  dictionary : 0/0, 0/0, 0/0 | imported : n/a
    havoc : 2/19.7k, 0/11.5k | stability : 100.00%
    trim : 70.73%/9, 0.00%
-----|-----|
                                                    [cpu000:111%]

```

Стратегия фаззинга отдельных функций, обрабатывающих пользовательский ввод в большом проекте, может быть более эффективной чем фаззинг всего приложения, особенно, если для кода уже написаны юнит-тесты.

2.4 Описание AFL-fuzzer UI

Окно после запуска процесса - и есть весь графический интерфейс AFL-фаззера. Здесь отображаются текущие результаты поиска. Если

говорить более сжато, то самый важный параметр это `uniq crashes`, который определяет количество найденных крахов программы. Параметр `cycles done` показывает сколько циклов прогонки тестовых аргументов через программу было выполнено. `Cycle progress` сообщает вам, как далеко продвинулся фаззер с текущим циклом. В `map coverage` приведены некоторые сведения о покрытии в инструментированном двоичный файле программы. `Stage progress` дает нам всесторонний взгляд на то, что на самом деле делает фаззер прямо сейчас.

```
— process timing —
    run time : 0 days, 0 hrs, 0 min, 28 sec
  last new path : 0 days, 0 hrs, 0 min, 28 sec
last uniq crash : 0 days, 0 hrs, 0 min, 28 sec
last uniq hang : none seen yet
```

Теперь, рассмотрим каждый параметр чуть подробнее. Начнём с `process timing` (время процесса). Этот раздел сообщает вам, как долго работает фаззер и сколько времени прошло с момента его последней находки.

Для дальнейшего понимания, стоит пояснить что `path` в контексте фаззера означает сокращение для тестовых случаев, которые запускают новые шаблоны выполнения, сбои и зависания.

На самом деле фаззинг в реальной эксплуатации может исчисляться в неделях или месяцах, поэтому данные о времени в частных случаях могут иметь практический смысл.

```
— overall results —
cycles done : 7
total paths : 6
uniq crashes : 1
uniq hangs : 0
```


Overall results (общие результаты) дают поверхностное представление о процессе в целом. Cycles done – количество циклических пробегов. Total paths – общее количество “путей”. Uniq crashes – количество зарегистрированных “уникальных” случаев экстренного завершения программы, о которых упоминалось ранее (в данном конкретном случае нам удалось обнаружить один, заведомо известный уникальный сбой в работе функции `sanitize_cookie_path()`). Uniq hangs – количество значимых уникальных зависаний (в данном случае зависания не предполагались).

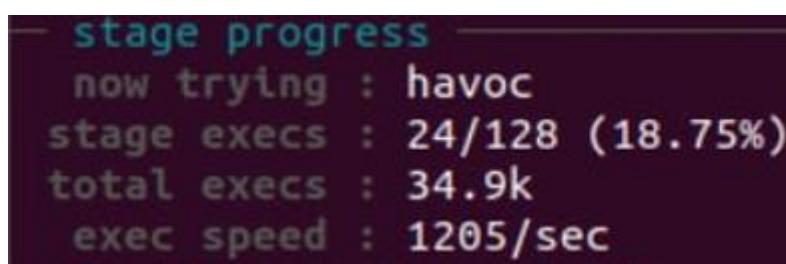
```
— cycle progress —
now processing : 1 (16.67%)
paths timed out : 0 (0.00%)
```

Cycle progress (Статус текущего цикла) сообщает вам, как далеко продвинулся фаззер с текущим циклом очереди. Now processing показывает идентификатор тестового примера, над которым он в настоящее время работает. Paths timed out - количество входных данных, которые он решил отбросить.

```
— map coverage —
map density : 0.02% / 0.03%
count coverage : 1.00 bits/tuple
```

Map coverage (Покрытие карты) - в этом разделе приведены некоторые сведения о покрытии, наблюдаемом инструментами, встроенными в целевой бинарный файл. Map density - отображает сколько ветвей мы уже затронули. Число слева описывает текущий вход, число справа — это значение для всего входного корпуса. Проценты могут достигать разных значений, но значения выше 70% встречаются очень редко в очень сложных программах, в которых интенсивно используется код, сгенерированный шаблонами. Count coverage - имеет дело с изменчивостью количества попаданий кортежа в двоичном файле. По сути,

если каждая выбранная ветвь всегда выполняется фиксированное количество раз для всех входных данных, которые мы пробовали, это будет читаться как «1,00». По мере того, как нам удастся активировать другие счетчики попаданий для каждой ветви, стрелка начнет двигаться к «8,00» (каждый бит в 8-битной карте попадает), но, вероятно, никогда не достигнет этого предела. Вместе эти значения могут быть полезны для сравнения охвата нескольких разных заданий фаззинга, основанных на одном и том же инструментированном двоичном файле.



```
— stage progress —
now trying : havoc
stage execs : 24/128 (18.75%)
total execs : 34.9k
exec speed : 1205/sec
```

Stage progress (Статус текущего этапа) – эта категория сообщает вам о том через какую стадию проходит фаззер в текущий момент. Now trying – главное поле этой категории, которая указывает на текущую стадию. Она может быть одной из:

calibration — этап перед фаззингом, на котором проверяется путь выполнения для обнаружения аномалий, установления базовой скорости выполнения и т. д. Выполняется очень быстро всякий раз, когда делается новая находка.

trim L/S — еще один этап предварительного фаззинга, на котором тестовый пример обрезается до кратчайшей формы, которая по-прежнему создает тот же путь выполнения. Длина (L) и шаг (S) выбираются исходя из общего размера файла.

bitflip L/S - детерминированные перевороты битов. В любой момент времени переключается L битов, просматривая входной файл с S-битными приращениями. Текущие варианты L/S: 1/1, 2/1, 4/1, 8/8, 16/8, 32/8.

arith L/8 - детерминированная арифметика. Фаззер пытается вычесть или добавить небольшие целые числа к 8-, 16- и 32-битным значениям. Шаг всегда равен 8 битам.

interest L/8 - перезапись детерминированного значения. У фаззера есть список известных «интересных» 8-, 16- и 32-битных значений, которые можно попробовать. Шаг составляет 8 бит.

extras - детерминированная инъекция терминов из словаря. Это может отображаться как «user» или «auto», в зависимости от того, использует ли фаззер пользовательский словарь (-х) или автоматически созданный. Вы также увидите «over» или «insert», в зависимости от того, перезаписывают ли словарные слова существующие данные или вставляются путем смещения оставшихся данных в соответствии с их длиной.

havoc — своего рода цикл фиксированной длины со сложенными случайными настройками. Операции, которые предпринимаются на этом этапе, включают перестановку битов, перезапись случайными и «интересными» целыми числами, удаление блоков, дублирование блоков, а также различные операции, связанные со словарем (если словарь предоставляется в первую очередь).

splice — стратегия последнего средства, которая срабатывает после первого полного цикла очереди без новых путей. Он эквивалентен «havoc», за исключением того, что splice сначала объединяет два случайных входа из очереди в некоторой произвольно выбранной средней точке.

sync — стадия, используемая только при установке -m или -s. Настоящего фаззинга не происходит, но инструмент сканирует выходные данные других фаззеров и при необходимости импортирует тестовые примеры. В первый раз это может занять несколько минут или около того.

Теперь разберём значения остальных полей. Stage execs - индикатор прогресса выполнения для текущего этапа. Total execs - глобальный счетчик выполненных операций, из значения данной строки и всего времени выполнения в последствии вычисляется средняя скорость операций в секунду (значение следующей строки). Exec speed – собственно, скорость протекания операций в секунду. Эти данные нужны в том числе и для того, чтобы фаззер мог предупредить вас о медленно протекающих процессах.

```
— findings in depth —
favored paths : 6 (100.00%)
new edges on  : 6 (100.00%)
total crashes  : 503 (1 unique)
total tmouts   : 0 (0 unique)
```

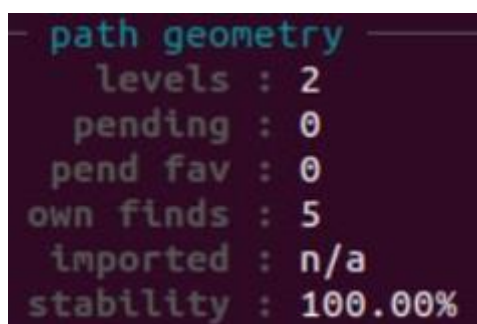
Findings in depth (Нахождения в глубине) - этот раздел включает в себя количество путей, которые больше всего нравятся фаззеру на основе встроенного в код алгоритма минимизации, и количество тестовых случаев, которые фактически привели к лучшему покрытию. Также есть дополнительные, более подробные счетчики сбоев и тайм-аутов.

Обратите внимание, что счетчик времени ожидания несколько отличается от счетчика зависаний; он включает все тестовые случаи, которые превысили тайм-аут, даже если они не превысили его с запасом, достаточным для классификации ожидания как зависания.

```
— fuzzing strategy yields —
bit flips : 1/224, 0/218, 1/206
byte flips : 0/28, 0/22, 0/10
arithmetics : 1/1554, 0/283, 0/71
known ints : 1/126, 0/533, 0/405
dictionary : 0/0, 0/0, 0/0
havoc : 2/19.7k, 0/11.5k
trim : 70.73%/9, 0.00%
```

Fuzzing strategy yields (Валидность стратегии фаззинга) - раздел, в котором отслеживается, сколько путей мы получили в результате пропорционально количеству попыток выполнения для каждой из стратегий фаззинга, обсуждавшихся ранее. Это служит для убедительной проверки предположений о полезности различных подходов, используемых afl-fuzz.

Статистика стратегии обрезки в этом разделе немного отличается от остальных. Первое число в этой строке показывает долю удаленных байтов из входных файлов; второй соответствует количеству исполнителей, необходимых для достижения этой цели. Наконец, третье число показывает долю байтов, которые, хотя и невозможно было удалить, были признаны неэффективными и были исключены из некоторых более дорогостоящих шагов детерминированного фаззинга.



```
path geometry
  levels : 2
  pending : 0
  pend fav : 0
  own finds : 5
  imported : n/a
  stability : 100.00%
```

Path geometry (Геометрия пути). Первое поле в этом разделе отслеживает глубину пути, достигнутую в процессе управляемого фаззинга. По сути: начальные тестовые случаи, предоставленные пользователем, считаются «уровнем 1». Тестовые случаи, которые могут быть получены из этого с помощью традиционного фаззинга, считаются «уровнем 2»; те, которые получены с использованием их в качестве входных данных для последующих раундов фаззинга, относятся к «уровню 3»; и так далее. Таким образом, максимальная глубина является приблизительным показателем того, какую ценность вы получаете от подхода, основанного на

инструментарии, используемого afl-fuzz. Следующее поле показывает количество входных данных, которые еще не прошли фаззинг. Та же статистика предоставляется для «избранных» записей, до которых фаззер действительно хочет добраться в этом цикле очереди (непривилегированным записям, возможно, придется подождать пару циклов, чтобы получить свой шанс). Затем у нас есть количество новых путей, найденных во время этого раздела фаззинга и импортированных из других экземпляров фаззинга при выполнении параллельного фаззинга; и степень, в которой идентичные входные данные иногда вызывают переменное поведение в тестируемом бинарном файле.

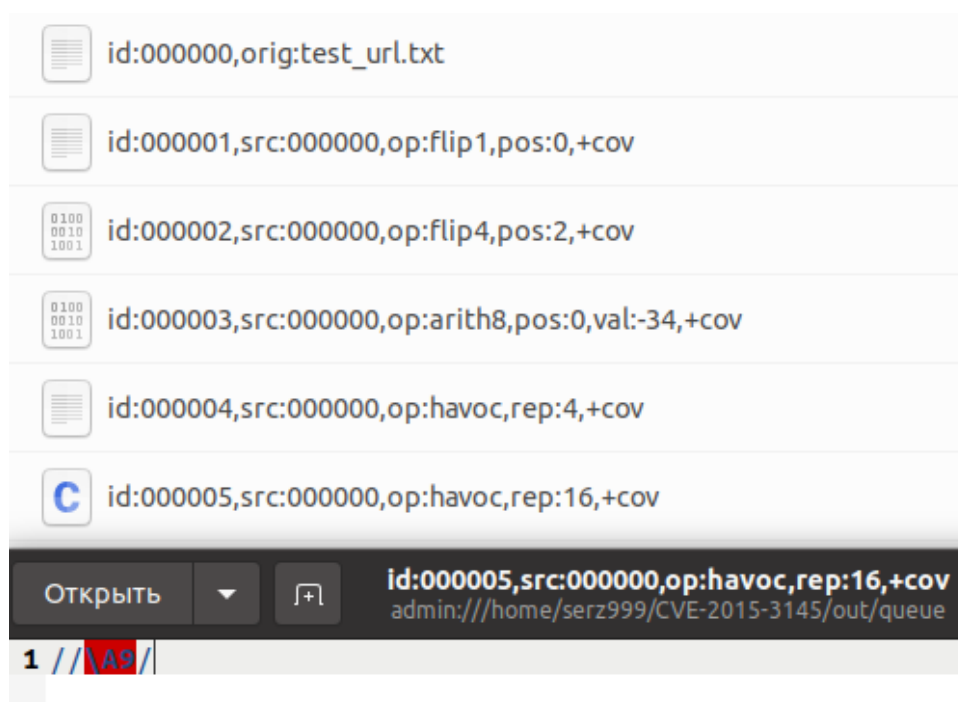


CPU load (Загрузка процессора). Этот крошечный виджет показывает очевидную загрузку ЦП в локальной системе. Он рассчитывается путем взятия количества процессов в «работоспособном» состоянии и последующего сравнения его с количеством логических ядер в системе. Если значение отображается зеленым цветом, вы используете меньше ядер ЦП, чем доступно в вашей системе, и, вероятно, можете использовать распараллеливание для повышения производительности. Если значение отображается красным цветом, возможно, ваш ЦП перегружен, и запуск дополнительных фаззеров может не дать вам никаких преимуществ.

2.5 Анализ вывода программы в директорию out/

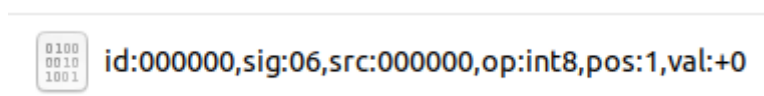
Процесс фаззинга будет продолжаться до тех пор, пока вы не нажмете Ctrl+C. В выходном каталоге создаются три подкаталога, которые обновляются в режиме реального времени:

queue/ — тестовые примеры для каждого пути выполнения, а также все начальные файлы, заданные пользователем (тестовый корпус). Например, в нашем случае:



Можно пронаблюдать как в конечном этапе был изменён начальный тестовый вход “/xxx/”.

crashes/ - уникальные тест-кейсы, в результате которых тестируемая программа получает экстренный сигнал ОС (например, SIGSEGV, SIGILL, SIGABRT). Записи сгруппированы по полученному сигналу. Пример содержания для нашего случая:



Можно сделать вывод, что программа поймала сигнал 06 – SIGABRT.

hangs/ - уникальные тест-кейсы, которые приводят к тайм-ауту тестируемой программы. В нашем случае зависания не были обнаружены.

Сбои и зависания считаются «уникальными», если связанные пути выполнения включают какие-либо переходы состояний, не замеченные в ранее зарегистрированных ошибках. Если к одной ошибке можно добраться несколькими способами, в начале процесса будет некоторая инфляция счетчика, но она должна быстро исчезнуть.

3. Вывод

Так мы ознакомились с тем, как можно использовать AFL для тестирования программ на практике. По сути, фаззинг — это способ обнаружения новых ошибок, и AFL максимально упрощает его. AFL — очень мощный инструмент, который почти не требует усилий в использовании. Это позволяет делать процесс фаззинга всего лишь за несколько шагов, в то время как он заботится обо всем сам в фоновом режиме. Эта технология может помочь в обнаружении багов о которых разработчики даже не подозревали, что делает её очень полезным и удобным инструментом для автоматического тестирования приложений, их последующего выпуска и эффективной эксплуатации.