СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В современном мире информационных технологий повышение безопасности программного обеспечения является одной из ключевых проблем. С увеличением сложности программных продуктов и их широким распространением во всех сферах жизни общества, вопрос выявления и устранения уязвимостей становится особенно актуальным.

Классические методы тестирования сталкиваются с рядом проблем, таких как невозможность предсказания всех сценариев использования ПО, сложности в обнаружении трудноуловимых ошибок и т.д. Фаззинг, как метод автоматизированного тестирования программ на наличие ошибок и уязвимостей путем подачи неожиданных или случайно сгенерированных данных на вход, показывает высокую эффективность в решении данной проблемы. Однако, несмотря на свои преимущества, фаззинг сталкивается с рядом сложностей, таких как выбор оптимальных стратегий генерации входных данных и анализ большого объема результатов тестирования.

Актуальность применения фаззинга определяется постоянно растущими требованиями к безопасности программного обеспечения и необходимостью эффективного и своевременного обнаружения потенциальных уязвимостей. Методы фаззинга, и в частности инструменты вроде AFL++, предлагают возможность автоматизации процесса поиска ошибок, что значительно повышает шансы на их обнаружение до того, как они будут эксплуатированы пользователями. Важность фаззинга усиливается в контексте непрерывно возрастающего числа угроз информационной безопасности и роста сложности программных систем, что делает традиционные методы тестирования недостаточно эффективными.

Целью данной работы является исследование и анализ методов фаззинга, с акцентом на использование инструмента AFL++ для выявления ошибок в программном обеспечении.

В рамках работы планируется детально изучить теоретические основы фаззинга, его внутреннее устройство и алгоритмы, а также провести практическое тестирование с использованием fuzz-goat [**???**] для демонстрации возможностей и эффективности метода. Особое внимание будет уделено анализу результатов фаззинга, выявлению и классификации обнаруженных уязвимостей, а также оценке возможностей AFL++ как инструмента для улучшения безопасности программного обеспечения.

В рамках достижения поставленной цели работы определены следующие **задачи**:

A. Изучение теоретических аспектов фаззинга как методики обнаружения ошибок в программном обеспечении, включая историческое развитие, основные подходы и классификацию фаззеров.
B. Анализ внутреннего устройства и алгоритмов фаззинга, освещение принципов генерации входных данных и механизмов обнаружения ошибок.
C. Освещение особенностей и возможностей AFL++, включая сравнение с другими инструментами фаззинга и детальное рассмотрение его функциональности.
D. Проведение практического фаззинга на примере fuzz-goat с использованием AFL++.
E. Выявление и анализ обнаруженных уязвимостей в ходе фаззинга fuzz–goat.
F. Оценка эффективности AFL++ и фаззинга в целом как инструментов повышения безопасности программного обеспечения, формулирование выводов о преимуществах и ограничениях методики.

ГЛАВА 1. ФАЗЗИНГ, КАК МЕТОДИКА ОБНАРУЖЕНИЯ УЯЗВИМОСТЕЙ В ПО

Фаззинг-тестирование - это тип тестирования безопасности, который позволяет обнаружить ошибки кодирования и лазейки в программном обеспечении, операционных системах или сетях [**???**].

Фаззинг включает в себя ввод огромного количества случайных данных в тестируемое программное обеспечение, чтобы заставить??? его дать сбой. Одним из ключевых преимуществ фаззинга является его способность эффективно сканировать программное обеспечение на предмет ошибок обработки данных, таких как переполнение буфера, уязвимости для SQL-инъекций, кросс-сайтового скриптинга и других видов уязвимостей, которые могут быть эксплуатированы злоумышленниками.

Фаззинг выполняется с помощью фаззера - программы, которая автоматически вводит псевдослучайные данные в программу и обнаруживает ошибки. Фаззинг-тестирование обычно выполняется автоматически. Оно обнаруживает наиболее серьезные ошибки или дефекты безопасности и является экономически эффективным методом тестирования. Фаззинг также является одним из самых распространенных методов, используемых хакерами для обнаружения уязвимостей системы [1].

1.1. Классификация фаззеров

По методу генерации новых входных значений фаззеры подразделяются на [**???**]:

- Mutation-Based Fuzzers: Это один из типов фаззинга, при котором фаззер имеет некоторые знания о входном формате тестируемой программы: на основе существующих выборок данных инструменты фаззинга на основе мутаций генерируют новые варианты (мутанты), которые он использует для фаззинга. Прим.: AFL++.
- Generation-Based Fuzzers: Фаззер генерирует входные данные с нуля. Например, использует входную модель, предоставленную пользователем, для генерации новых входных данных. Прим.: Peach Fuzzer [1].

По цели фаззинга [**???**]:

- Protocol-based Fuzzers. Фокусируются на тестировании безопасности сетевых протоколов и программ, обрабатывающих конкретные файловые форматы (изображения, архивы и документы и другое) Прим.: Boofuzz.
- Library and Framework Fuzzers. Предназначаются для тестирования функций из готовых библиотек и пакетов. Прим.: LibFuzzer, Jazzer.
- Binary Fuzzers. Осуществляют проверку безопасности исполняемого бинарного кода. Прим.: AFL++ (QEMU mode).
- API Fuzzers. Предназначаются для тестирования безопасности веб-сервисов и API. Прим.: Restler [6].

1.2. Недостатки fuzzing-тестирования

Несмотря на очевидные преимущества, фаззинг также имеет недостатки [**???**]:

- Само по себе fuzzing тестирование не может дать полную картину общей угрозы безопасности или ошибок.
- Fuzzing тестирование менее эффективно для борьбы с угрозами безопасности, которые не вызывают сбоев программы, например, с некоторыми вирусами, червями, троянами и т. д.
- Fuzzing тестирование может обнаружить только простые неисправности или угрозы.
- Для эффективной работы потребуется значительное время [7].

Тем не менее, фаззинг остается одним из самых мощных инструментов в арсенале специалистов по кибербезопасности для обнаружения и устранения уязвимостей в программном обеспечении. Применение этой методики позволяет не только улучшить качество разрабатываемых программ, но и значительно повысить уровень защиты информационных систем от внешних атак.

ГЛАВА 2. ВНУТРЕННЕЕ УСТРОЙСТВО И АЛГОРИТМЫ ФАЗЗИНГ-ТЕСТИРОВАНИЯ

Глава посвящена подробному рассмотрению внутреннего устройства и алгоритмов, лежащих в основе фаззинг-тестирования. Особое внимание уделяется классификации методов фаззинга, которые различаются по уровню доступа к

информации о тестируемой программе. Это разделение играет ключевую роль в выборе стратегии тестирования и определяет эффективность обнаружения уязвимостей. В главе также описываются различные методы fuzzing-тестирования, каждый из которых имеет свои особенности и предпочтительные области применения.

2.1. Базовые этапы фаззинг-тестирования

Далее приведем.... [**???**].

1. *Определение целевой системы*. Выбор программного обеспечения или системы, которая будет подвергнута тестированию.
2. Определение входных данных:??? Анализ типов данных, обрабатываемых системой в нормальных условиях работы.
3. Генерация fuzzing данных: Создание или модификация данных для генерации нестандартных входных значений, которые будут использоваться в тестах.
4. Проведение теста с fuzzing данными: Подача сгенерированных данных в систему и наблюдение за её реакцией.
5. Мониторинг поведения системы: Наблюдение за системой во время тестирования для выявления аномалий, сбоев и других нештатных ситуаций.
6. Фиксация обнаруженных дефектов: Запись всех найденных проблем и уязвимостей в журнал дефектов для дальнейшего анализа.
7. Анализ и устранение дефектов: Оценка выявленных уязвимостей и разработка решений для их исправления, направленных на повышение безопасности и надежности тестируемой системы [8].

На рис.2.1 приведён принцип работы фаззера.

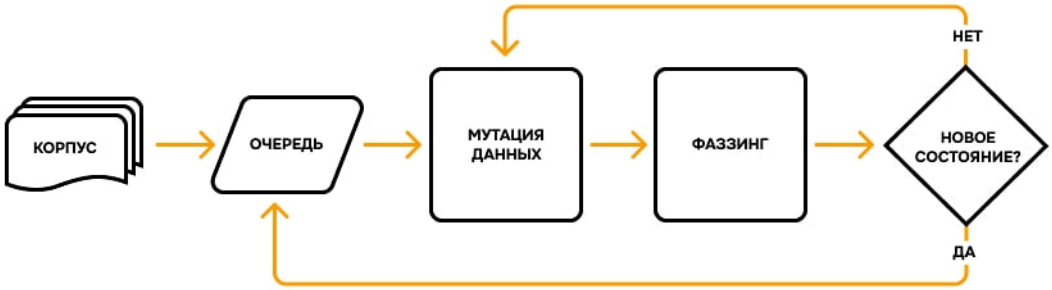

Рис.2.1. Принцип работы фаззера

2.2. Классификация методов фаззинга по уровню доступа к информации

Выделяют три вида фаззинга [**???**]: *фаззинг чёрного ящика, фаззинг серого ящика и фаззинг белого ящика*, в зависимости от того, сколько информации требуется фаззеру от тестируемой программы во время исполнения.

Это разделение весьма условно, на практике, например, фаззеры белого ящика часто используют некоторые аппроксимации [**???**]. Термин *«черный ящик»*, обычно использующийся при тестировании программного обеспечения, в случае с фаззингом обозначает аналогичное: фаззер во время своей работы не получает никакой информации от тестируемой программы, он может взаимодействовать с программой только через ввод/вывод программы. Большинство традиционных фаззеров относят к этой категории.

Фаззинг белого ящика — противоположность чёрного. Здесь программа анализирует внутреннее устройство тестируемой программы. Такой фаззер использует метод, который теоретически может исследовать все пути выполнения в тестируемой программе. В отличие от фаззинга черного ящика, фаззингу белого ящика требуется информация от тестируемой программы и она используется для руководства генерацией тестов.

В частности, начиная исполнение с заданным конкретным входом, фаззер белого ящика сначала собирает символьные ограничения для всех условных операторов на пути исполнения. Следовательно, после одного исполнения такой фаззер объединяет все символьные ограничения с помощью конъюнкции для формирования ограничения пути. Затем фаззер белого ящика последовательно отрицает

одно из ограничений и решает новое ограничение пути [**???**]. Накладные расходы на фаззинг белого ящика обычно значительно превосходят расходы на фаззинг черного ящика. Это связано с тем, что реализации динамического символьного исполнения часто используют динамические инструменты и SMT-решатели, что весьма трудоёмко. Фаззинг серого ящика — что-то среднее между первыми двумя видами.

Фаззер серого ящика может получать некоторую информацию о внутренней структуре тестируемой программы и/или её исполнении. Фаззеры серого ящика полагаются на приблизительную информацию, чтобы увеличить скорость и иметь возможность тестировать программу на большем количестве входных данных [3].

На рис.2.2 представлено сравнение подходов к тестированию ПО.

Подход	**Знания о тестируемом объекте**	**Доступные данные**	**Применимость**
White-box	Полные	Исходный код	Приложения, осуществляющие парсинг, валидацию данных Приложения, в исходном коде которых мало строк
Grey-box	Неполные	Скомпилированная программа	Локальные приложения Приложения, в исходном коде которых много строк
Black-box	Минимальные	Внешний интерфейс	Работающие удаленно приложения Встроенное ПО

Рис.2.2. Подходы к тестированию ПО [9]

2.3. Методы генерации данных fuzzing-тестирования

2.3.1. Случайное тестирование

Генерация начинается со случайных входных данных в качестве аргументов, а затем используются некоторые дополнительные знания для получения новых входных данных. Одним из основных представителей этой категории методов является случайное тестирование с обратной связью, которое улучшает генерацию

тестов за счёт полученной с предыдущих итераций обратной связи, которая направляет процесс создания входных данных на последующих итерациях.

Это позволяет снизить количество повторяющихся и/или недопустимых входных данных, используемых для тестирования. Инструменты для генерации тестов, использующие методы случайного тестирования, обладают преимуществами масштабируемости и простоты реализации, но используют много ресурсов на генерацию недопустимых, бесполезных (не улучшающих покрытие) входных данных ввиду случайной природы методов. Добавление обратной связи позволяет нивелировать данный недостаток [3].

2.3.2. Тестирование на основе поиска

При тестировании на основе поиска задача генерации тестов решается с помощью алгоритмов поиска, например, генетических алгоритмов. В таком случае генерация входных данных формулируется как проблема поиска, множество возможных входных значений формирует пространство решений, а метрика, которую хочется максимизировать, например, покрытие кода, кодируется как функция приспособленности. Таким образом, поиск будет осуществляться с помощью выбранной функции приспособленности, которая представляет из себя эвристику, оценивающую, насколько близко текущее решение к оптимальному. Руководствуясь функцией приспособленности, алгоритм поиска итеративно выдаёт лучшие решение до тех пор, пока либо не будет найдено оптимальное решение, либо не будет выполнено условие остановки (например, истечение выделенного на генерацию времени) [3].

2.3.3. Символьное тестирование

Тестирование с помощью символьного исполнения — это аналитический подход, основанный на правилах вывода. Он использует символьные переменные как входные данные программы и представляет значения программных переменных как символьные выражения, а путь исполнения — выражением над символьными переменными (ограничением пути). Символьное исполнение применяется для поиска всех возможных путей исполнения программы. Данный подход способен находить пути исполнения программы, которые вызывают ошибки, даже без компиляции. Для решения ограничений пути, как правило, применяется SMT-решатель, который вычисляет уже конкретные значения переменных, необходимых для

того, чтобы поток исполнения пошёл по данному пути. Несмотря на очевидные преимущества и силу, этот подход обычно реализуется в мелком масштабе, так как в крупных проектах наблюдается значительный рост всех возможных путей исполнения, что требует серьёзных вычислительных ресурсов. Именно по этой причине этот подход только недавно стал применяться для генерации тестов, хотя идея символьного исполнения появилась в 70-х годах [3].

ГЛАВА 3. ИНСТРУМЕНТ ФАЗЗИНГ-ТЕСТИРОВАНИЯ AFL++

AFL++ - это улучшенная версия популярного инструмента фаззинга AFL (American Fuzzy Lop), разработанная сообществом для повышения его эффективности и функциональности. *AFL* — это ориентированный на поиск ошибок инструмент анализа ПО, который использует обширный список типов инструментации кода для получения информации о покрытии и множество генетических алгоритмов мутации для автоматического обнаружения различных тестовых примеров, которые вызывают новые внутренние состояния в бинарном коде ПО.

Фаззер AFL поддерживает инструментацию, реализующуюся на этапе компиляции из исходного кода с помощью оберток afl-gcc/afl-g++. afl-gcc/afl-g++ подменяет вызываемую команду на обертку, переписывающую ассемблерный код, сгенерированный компилятором. Для фаззинга уже скомпилированных бинарных файлов используется qemu mode — это патч к QEMU, эмулирующий запуск одного процесса и реализующий требуемую для анализа инструментацию кода [8].

AFL является одним из самых известных фаззеров серого ящика, он поддерживает фаззинг программ, написанных на C, C++ и Objective C, скомпилированных с помощью и GCC, и CLang, но его часто расширяют и портируют.

AFL требует от пользователя предоставить пример команды, запускающей тестируемое приложение, и хотя бы один небольшой пример входных данных. Входные данные могут быть переданы в тестируемую программу либо через стандартный ввод, либо в виде входного файла, указанного в командной строке процесса. Чтобы максимизировать производительность фаззинга, American fuzzy lop ожидает, что тестируемая программа будет скомпилирована с помощью служебной программы, которая оснащает (инструментирует) код вспомогательными функциями, которые отслеживают поток управления.

3.1. Инструментирование

Инструментирование – это процесс внедрения дополнительного кода или инструкций в исходный код программы с целью сбора информации о её выполнении. Это позволяет фаззеру:

- Отслеживать, какие части кода выполняются (покрытие кода).
- Определять, какие новые ветви кода (пути выполнения) были исследованы благодаря новым или мутированным входным данным.
- Собирать статистику о выполнении для последующего анализа.

AFL представляет собой фаззер серого ящика , то есть он внедряет инструменты для измерения покрытия кода в целевую программу во время компиляции и использует метрику покрытия для управления генерацией новых входных данных. В случаях, когда это невозможно, также поддерживается тестирование «черного ящика» [2].

3.2. Измерение покрытия

AFL подсчитывает количество раз, которое данное выполнение целевой программы проходит через каждое ребро в графе управления потоком целевой программы. Ребро в графе управления потоком (control-flow graph, CFG) программы представляет собой переход между двумя узлами (которые представляют собой отдельные блоки инструкций) [**???**].

AFL поддерживает глобальный набор пар (словарь) которые были произведены любым выполнением до этого момента. Входные данные считаются "интересными"и добавляются в очередь, если они производят пару, которой ещё нет в словаре.

Инструментарий, внедряемый в скомпилированные программы, фиксирует покрытие ветвей (рёбер), а также подсчеты срабатываний условных переходов. Точки ветвления в контексте программирования — это места в коде, где происходит условный переход, который может изменить путь выполнения программы. Например, это могут быть операторы if, switch, циклы, или даже вызовы функций — все, что может привести к изменению последовательности исполняемых инструкций [4].

Код, вставляемый в точках ветвления, по существу включает следующее:

```
1.cur_location ← ⟨COMPILE_TIME_RANDOM⟩
2.foreach prev_location do
3.   shared_mem[cur_location ⊕ prev_location] ++
4.   prev_location ← cur_location ≫ 1
5.end
```

Рис.3.1. Код, измеряющий покрытие, внедряемый в точки ветвления [**???**]

cur_location —это переменная, которая используется как идентификатор текущего места в коде, где выполняется операция инструментирования. Значение cur_location генерируется случайно на этапе компиляции и уникально для каждой точки ветвления. Оно используется для того, чтобы отслеживать, была ли достигнута эта точка ветвления при выполнении программы.

shared_mem:??? shared_mem — это массив, который используется как словарь покрытия для отслеживания, какие точки ветвления (и переходы между ними) были выполнены во время исполнения программы. Ключем является xor (prev_location, cur_location), а значением счетчик попаданий для данного кортежа (перехода). Каждый раз, когда исполнение программы проходит через этот переход, счетчик увеличивается

prev_location помогает отслеживать предыдущее состояние исполнения программы и выявлять изменения в путях исполнения при каждой новой итерации фаззинга.

Размер словаря выбран таким образом, чтобы коллизии возникали редко для почти всех предполагаемых целей. Коллизии могут возникать, например, когда несколько различных входных данных приводят к одной и той же точке ветвления (branch) в программе или к одному и тому же состоянию программы. На рис.3.2 представлена зависимость количества коллизий от количества ветвлений.

В то же время размер словаря достаточно мал, чтобы анализ словаря занимал всего лишь несколько микросекунд и чтобы словарь легко помещался в кэш второго уровня (L2 cache) [4].

Такая КАКАЯконкретно??? форма покрытия предоставляет значительно более глубокое представление о пути выполнения программы, в частности, она тривиально различает следующие трассы исполнения:

A -> B -> C -> D -> E (кортежи: AB, BC, CD, DE) и

A -> B -> D -> C -> E (кортежи: AB, BD, DC, CE)

Branch cnt	Colliding tuples	Example targets
1,000	0.75%	giflib, lzo
2,000	1.5%	zlib, tar, xz
5,000	3.5%	libpng, libwebp
10,000	7%	libxml
20,000	14%	sqlite
50,000	30%	•

Рис.3.2. Распределение количества коллизий кортежей при различном количестве ветвлений для разных целей фаззинга [**???**]

3.3. Обнаружение нового поведения

Фаззер поддерживает глобальный словарь кортежей, замеченных в предыдущих исполнениях, эти данные можно быстро сравнить с отдельными трассами и обновить [**???**].

Когда измененный вход генерирует трассировку выполнения, которая содержит новые кортежи, соответствующий входной файл будет сохранен и направлен для дополнительной обработки позже. Входы, которые не инициируют новые переходы состояний локального масштаба во время отслеживания выполнения (например, не генерируют новые кортежи), будут отброшены, даже если их общий поток управления уникален.

В дополнение к обнаружению новых кортежей фаззер также учитывает приблизительное количество кортежей. Они разделены на несколько частей [**???**]: 1, 2, 3, 4-7, 8-15, 16-31, 32-127, 128+.

Эта упрощенная классификация помогает фаззеру быстро определить "интересные"изменения в потоке управления программы без необходимости в точном подсчете каждого выполнения. Вместо того, чтобы отслеживать каждое выполнение отдельно, фаззер смотрит на общую картину и обнаруживает изменения, которые выходят за пределы ожидаемых или типичных паттернов выполнения программы.

3.4. Изменение очереди ввода

Тестовые случаи мутации, которые генерируют новые переходы состояний в программе, добавляются во входную очередь и служат отправной точкой для

будущих циклов фаззинга. Они дополняют, но не могут автоматически заменить существующие открытия. В отличие от более жадных генетических алгоритмов, этот подход позволяет инструменту постепенно исследовать различные непересекающиеся и, возможно, взаимно несовместимые функции базового формата данных, как показано на рис.3.3 [4].

На рис.3.2 представлена зависимость количества коллизий от количества ветвлений.

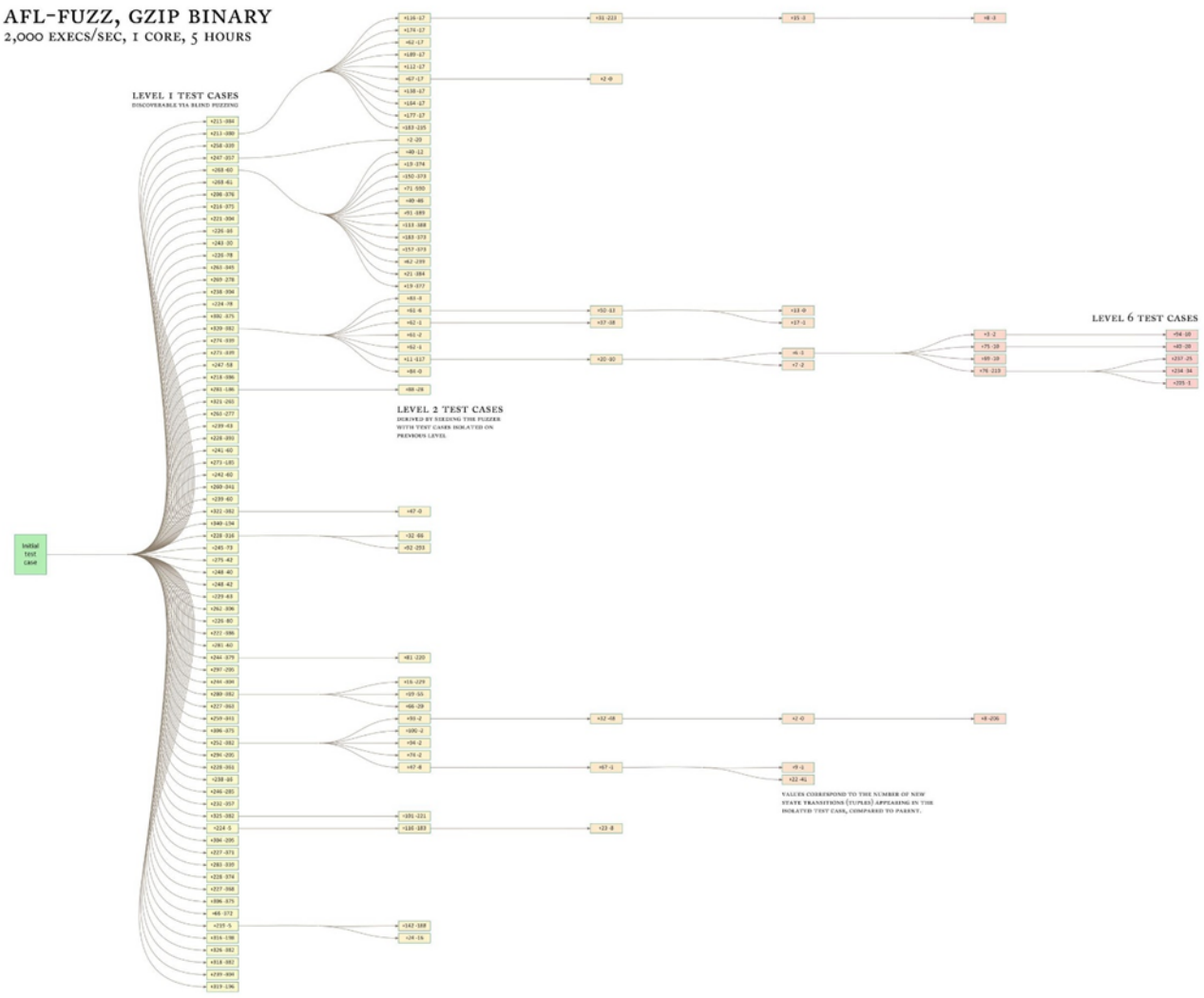

Рис.3.3. Визуализация дерева тестирования AFL-Fuzz [**???**]

В общем, AFL поддерживает очередь, и каждый раз, когда файл берется из этой очереди, в нее вносится множество мутаций, и проверяется, не обнаружит ли новые пути после операции.

3.5. Отбраковка корпуса

Корпус (corpus) - это коллекция тестовых данных или файлов, которые используются для проведения тестирования программы или системы [**???**]. Корпус включает в себя как исходные тестовые данные, так и те, которые были сгенерированы или изменены в ходе тестирования.

Подход к пошаговому исследованию состояний, описанный выше, означает, что некоторые тестовые случаи, синтезированные позднее, могут иметь покрытие ветвей, которое является строгим надмножеством покрытия, предоставленного их предками. Чтобы оптимизировать процесс фаззинга, AFL периодически переоценивает очередь, используя быстрый алгоритм, который выбирает меньшее подмножество тестовых примеров, которые по-прежнему охватывают каждый кортеж, виденный на данный момент, и характеристики которых делают их особенно благоприятными для инструмента.

Сгенерированный корпус "предпочтительных"записей обычно в 5-10 раз меньше, чем исходный набор данных. Непредпочтительные записи не удаляются, но они пропускаются с различными вероятностями при обнаружении в очереди:

- Если в очереди есть новые, еще не прошедшие фаззинг, предпочтительные записи, 99% непредпочтительных записей будут пропущены, чтобы достичь предпочтительных.
- Если нет новых предпочтительных записей:
  - Если текущая непредпочтительная запись уже прошла через фаззинг, она будет пропущена в 95% случаев.
  - Если она еще не прошла ни одного фаззинга, вероятность пропуска уменьшается до 75%.

Это обеспечивает разумный баланс между скоростью цикла очереди и разнообразием тестовых случаев [4].

3.6. Стратегии фаззинга, предоставляемые AFL++

Для генерации новых входных данных AFL применяет различные мутации к существующим данным. Эти мутации в основном нечувствительны к формату входных данных целевой программы; они обычно обрабатывают входные данные как простой массив бинарных данных.

Сначала AFL применяет детерминированную последовательность мутаций к каждому входному файлу. Они применяются в различных местах входных данных и включают:

- Инвертирование (т.е. отрицание или инвертирование) от 1 до 32 бит.
- Инкрементирование и декрементирование 8-, 16- и 32-битных целых чисел в кодировках как little-endian, так и big-endian.
- Перезапись частей входных данных "примерно двумя десятками 'интересных' значений включая ноль, максимальные и минимальные знаковые и беззнаковые целые различной ширины, также в кодировках little- и big-endian.
- Замена частей входных данных данными, взятыми из "словаря"пользовательских или автоматически обнаруженных токенов (например, магические байты или ключевые слова в текстовом формате).

После применения всех доступных детерминированных мутаций AFL переходит к стадии "havoc на которой подряд применяются от 2 до 128 мутаций. Эти мутации включают описанные выше детерминированные мутации, а также:

- Перезапись байтов случайными значениями.
- Операции над многобайтовыми "блоками":
  - Удаление блоков.
  - Дублирование блоков.
  - Установка каждого байта в блоке в одно значение.

Если AFL проходит через всю очередь, не генерируя ни одного входа, который достигает нового покрытия кода, он начинает "склеивание". Склеивание берет два входа из очереди, обрезает их в произвольных позициях, соединяет их вместе и применяет к результату стадию "havoc"[2].

3.6.1. Стратегия фаззинга: Bit flips

Первая и наиболее элементарная стратегия, используемая afl включает в себя выполнение последовательных, упорядоченных переворотов битов. Переход всегда составляет один бит; количество бит, переворачиваемых подряд, варьируется от одного до четырех. Для большого и разнообразного массива входных файлов наблюдаемые результаты следующие [**???**]:

- Переключение одного бита: 70 новых путей на миллион сгенерированных входных данных

- Переключение двух битов подряд: 20 дополнительных путей на миллион сгенерированных входных данных
- Переключение четырех битов подряд: 10 дополнительных путей на миллион входных данных [5].

```
#define FLIP_BIT(_ar, _b) do {
  u8 *_arf = (u8 *)(_ar);
  u32 _bf = (_b);
  _arf[(_bf) >> 3] ^= (128 >> ((_bf) & 7));
} while (0)
```

Рис.3.4. Реализация алгоритма bit flip в AFL++ [**???**]

Этот макрос переворачивает конкретный бит в массиве. Он использует _ar для указания на массив, а _b для указания на бит, который нужно перевернуть. Переворот осуществляется путём применения операции XOR к соответствующему байту и биту в этом байте.

3.6.2. Стратегия фаззинга: Byte flips

Естественное продолжение подхода пошагового переворота битов, этот метод основан на битовых переворотах шириной 8, 16 или 32 бита с постоянным переходом на один байт. Эта стратегия обнаруживает около 30 дополнительных путей на миллион входных данных, в дополнение к тому, что могло бы быть запущено при более коротких переключениях битов. В AFL++ byte flips реализован аналогично bit flips [5].

3.6.3. Стратегия фаззинга: Simple arithmetics

Этап состоит из трех отдельных операций [**???**]. Сначала фаззер пытается выполнить вычитание и сложение для отдельных байтов. Второй проход включает просмотр 16-битных значений с использованием обоих порядковых значений, но увеличивая или уменьшая их только в том случае, если операция также повлияла бы на самый старший байт (в противном случае операция просто дублировала бы результаты 8-битного прохода). Заключительный этап следует той же логике, но для 32-битных целых чисел [5].

Рассмотрим часть кода, отвечающую за мутацию 8-ми битных значений:

```
  for (i = 0; i < (u32)len; ++i) {
  u8 orig = out_buf[i];
  if (!skip_eff_map[i]) continue;
  for (j = 1; j <= ARITH_MAX; ++j) {
    u8 r = orig ^ (orig + j);
    if (!could_be_bitflip(r)) {
      out_buf[i] = orig + j;
      if (common_fuzz_stuff(afl, out_buf, len)) { goto
         abandon_entry; }
    }
    out_buf[i] = orig - j;
    if (!could_be_bitflip(r)) {
      if (common_fuzz_stuff(afl, out_buf, len)) { goto
         abandon_entry; }
    }
    out_buf[i] = orig;
  }
}
```

Рис.3.5. Реализация алгоритма simple arithmetic в AFL++ [**???**]

Этот код проходит через каждый байт входных данных, пытаясь прибавить и вычесть к нему значения от 1 до ARITH_MAX. Мутация применяется только если результат операции не может быть достигнут с помощью простого битфлипа, что позволяет сосредоточиться на более сложных и интересных случаях. После каждой мутации вызывается функция common_fuzz_stuff, которая запускает тестируемое приложение с мутированными данными и проверяет результаты на наличие ошибок.

Эта стратегия помогает обнаруживать ошибки, связанные с некорректной обработкой числовых значений, такие как переполнения или неправильные граничные условия, что делает её важной частью процесса фаззинга.

3.6.4. Стратегия фаззинга: Known integers

AFL полагается на жестко запрограммированный набор целых чисел, выбранных из-за их явно повышенной вероятности запуска граничных условий в типичном коде (например, -1, 256, 1024, MAX_INT-1, MAX_INT) [**???**]. Фаззер использует переход на один байт, чтобы последовательно перезаписать существующие данные во входном файле одним из примерно двух десятков "интересных"значений, используя оба порядковых номера (записи имеют ширину 8, 16 и 32 бита).

Эффективность на этом этапе составляет от 2 до 5 дополнительных путей на миллион попыток [5].

3.6.5. Стратегия фаззинга: Stacked tweaks

Когда детерминированные стратегии исчерпаны для конкретного входного файла, фаззер продолжает выполнять бесконечный цикл рандомизированных операций, которые состоят из последовательности [**???**]:

- Однобитовые перевороты,
- Попытки установить "интересные"байты, то есть байты, связанные с критическими частями формата данных.
- Сложение или вычитание небольших целых чисел в байты, слова или dw-слова (оба в конце),
- Полностью случайные однобайтовые наборы,
- Блокирует удаление,
- Блокирует дублирование с помощью перезаписи или вставки,
- Блокирует memset [5].

```
  stack_max = 1 << (1 + rand_below(afl, afl->havoc_stack_pow2)
     );
for (afl->stage_cur = 0; afl->stage_cur < afl->stage_max; ++
   afl->stage_cur) {
  u32 use_stacking = 1 + rand_below(afl, stack_max);
  afl->stage_cur_val = use_stacking;
  for (i = 0; i < use_stacking; ++i) {
    // Применение мутации...
  }
}
```

Рис.3.6. Реализация алгоритма stacked tweaks в AFL++

stack_max определяет максимальное количество мутаций, которое может быть применено в одном цикле фаззинга. Это значение адаптируется в зависимости от общего "показателя мощности"(havoc_stack_pow2), что позволяет динамически регулировать интенсивность мутаций в зависимости от контекста.

Внутри основного цикла определяется **use_stacking**, которое указывает, сколько именно мутаций будет применено к текущим входным данным. Это значение выбирается случайным образом в пределах от 1 до stack_max.

Далее идёт вложенный цикл, в котором непосредственно выполняется последовательное применение мутаций. Каждая мутация выбирается случайным образом из заранее определённого массива возможных мутаций (mutation_array), и её эффект накладывается на текущие входные данные.

Таким образом, стратегия "stacked tweaks"позволяет AFL++ генерировать комплексные тестовые случаи, применяя несколько мутаций к одному набору входных данных за один цикл тестирования.

3.6.6. Стратегия фаззинга: Test case splicing

Стратегия «последнего шанса», включающая в себя извлечение из очереди двух разных входных файлов, которые отличаются по крайней мере в двух местоположениях, и объединение их в случайном месте посередине перед отправкой этого временного входного файла с помощью короткого запуска алгоритма "stacked tweaks-[**???**]. Эта стратегия обычно обнаруживает около 20% дополнительных путей выполнения, которые вряд ли сработают при использовании только предыдущей операции.

```
/* Первоначально, если мы модифицировали in_buf для havoc,
 очистим это... */
if (in_buf != orig_in) {
  in_buf = orig_in;
  len = afl->queue_cur->len;
}
/* Выбираем случайную запись в очереди и переходим к ней.
 Не склеиваем с самим собой. */
do {
  tid = rand_below(afl, afl->queued_items);
} while (unlikely(tid == afl->current_entry || afl->
   queue_buf[tid]->len < 4));
/* Получаем тестовый случай */
afl->splicing_with = tid;
target = afl->queue_buf[tid];
new_buf = queue_testcase_get(afl, target);
/* Находим подходящее место для склеивания, где-то между
первым и последним отличающимся байтом. Отказываемся, если
 разница составляет всего один байт или около того. */
locate_diffs(in_buf, new_buf, MIN(len, (s64)target->len), &
   f_diff, &l_diff);
if (f_diff < 0 || l_diff < 2 || f_diff == l_diff) { goto
   retry_splicing; }
/* Разделяем где-то между первым и последним
 отличающимся байтом. */
split_at = f_diff + rand_below(afl, l_diff - f_diff);
/* Выполняем операцию. */
len = target->len;
afl->in_scratch_buf = afl_realloc(AFL_BUF_PARAM(in_scratch),
    len);
memcpy(afl->in_scratch_buf, in_buf, split_at);
memcpy(afl->in_scratch_buf + split_at, new_buf + split_at,
   len - split_at);
in_buf = afl->in_scratch_buf;
```

Рис.3.7. Реализация алгоритма test case splicing в AFL++ [**???**]

Сначала выбирается случайная запись из очереди входных данных, исключая текущую, чтобы избежать склеивания файла с самим собой.

Затем определяется подходящее место для склеивания, основываясь на различиях между текущим файлом и выбранным файлом. Это делается для обеспечения того, чтобы склеенные части были максимально разнообразными и могли породить новые пути выполнения в тестируемой программе.

После этого выбранные файлы склеиваются в выбранной точке, и полученный blob данных подвергается дальнейшим мутациям по стратегии "havoc".

Этот метод особенно полезен для преодоления "тупиков"в процессе фаззинга, когда другие стратегии не приводят к нахождению новых ошибок.

ГЛАВА 4. ПРАКТИЧЕСКОЕ ПРИМЕНЕНИЕ AFL++

Рассмотрим процесс фаззинга на практике. В качестве объекта тестирования будет использоваться программа Fuzzgoat [**???**] Данное приложение на языке C содержит преднамеренно введенные уязвимости для анализа эффективности фаззеров и инструментов статического анализа.

4.1. Фаззинг-тестирование

Перед началом тестирования следует установить инструмент фаззинга AFL++, что предполагает скачивание и компиляцию исходного кода AFL с последующей установкой [**???**].

После установки AFL можно приступать к сборке Fuzzgoat с использованием make, при условии, что afl-gcc находится в PATH.

Запуск фаззинга осуществляется командой:

```
afl-fuzz -i in -o out ./fuzzgoat @@,
```

где:

- `-i in` указывает на директорию с начальными тестами.
- `-o out` обозначает директорию для сохранения результатов фаззинга.
- `./fuzzgoat` — исполняемый файл тестируемой программы.
- `@@` — местозаполнитель, который AFL заменяет на имена файлов из тестового набора.

Результаты фаззинга, такие как количество проведенных тестов, скорость тестирования и найденные краши, можно наблюдать в реальном времени в пользовательском интерфейсе AFL. Все найденные ошибки будут сохранены в директорию out/default/crashes.

На рис.4.1 представлена панель мониторинга фаззинг-тестирования.

На этой панели отображаются результаты процесса тестирования (рис.4.2).

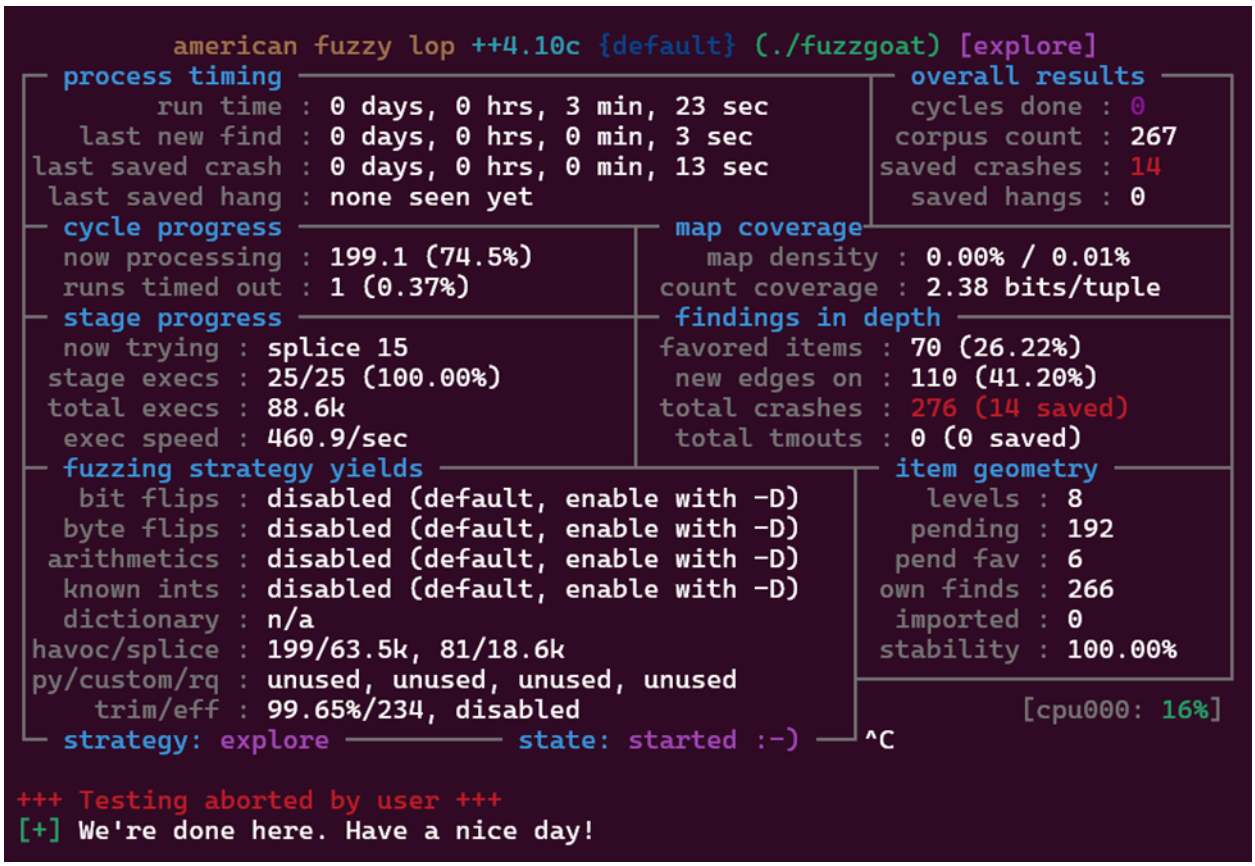

Рис.4.1. Панель мониторинга фаззинг-тестирования

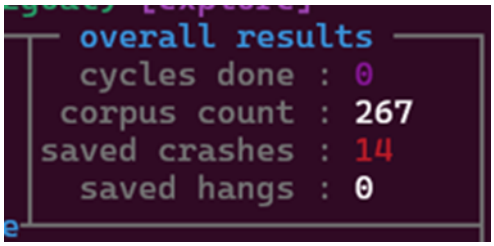

Рис.4.2. Текущие результаты фаззинг-тестирования

- **cycles done:**??? Это количество полных циклов фаззинга, которое было выполнено. Один цикл обычно представляет собой полное исполнение всех тестов в корпусе данных (наборе уникальных тестовых случаев).
- **corpus count:** Количество уникальных тестов, которые в данный момент находятся в "корпусе"— наборе данных для тестирования. AFL++ использует этот корпус как исходный материал для генерации новых входных данных.
- **saved crashes:** Количество уникальных сбоев, которые были обнаружены.
- **saved hangs:** Количество случаев, когда тестируемая программа "зависала"или не отвечала в течение определенного времени, установленного AFL++.

4.2. Анализ выявленных ошибок

В процессе фаззинга были обнаружены следующие ошибки (рис.4.3):

```
alexander1703@Alexander:~/fuzzing/fuzzgoat$ ls  out/default/crashes/
README.txt
id:000000,sig:06,src:000000+000003,time:4425,execs:1706,op:splice,rep:3
id:000001,sig:11,src:000000+000003,time:4736,execs:1834,op:splice,rep:5
id:000002,sig:06,src:000037,time:24397,execs:10195,op:havoc,rep:2
id:000003,sig:06,src:000037,time:28786,execs:12124,op:havoc,rep:2
id:000004,sig:06,src:000034,time:40753,execs:17263,op:havoc,rep:1
id:000005,sig:06,src:000034,time:62615,execs:26557,op:havoc,rep:2
id:000006,sig:11,src:000034,time:63483,execs:26891,op:havoc,rep:2
id:000007,sig:06,src:000073,time:74256,execs:31516,op:havoc,rep:4
id:000008,sig:11,src:000073,time:90221,execs:38422,op:havoc,rep:3
id:000009,sig:06,src:000169,time:101808,execs:43688,op:havoc,rep:2
id:000010,sig:11,src:000178,time:127337,execs:54994,op:havoc,rep:3
id:000011,sig:06,src:000161+000217,time:150531,execs:65561,op:splice,rep:6
id:000012,sig:11,src:000246+000000,time:179202,execs:78699,op:splice,rep:1
id:000013,sig:11,src:000220,time:186806,execs:82140,op:havoc,rep:4
id:000014,sig:06,src:000279,time:257831,execs:112585,op:havoc,rep:4
id:000015,sig:11,src:000040+000033,time:512428,execs:224439,op:splice,rep:5
```

Рис.4.3. Ошибки, выявленные в результате тестирования

Рассмотрим как AFL именует файлы с ошибками: id:000015,sig:11,src:000040+0000

- **id:000015:**??? Это уникальный идентификатор ошибки.
- **sig:11:** Это номер сигнала, который был сгенерирован операционной системой при краше. Сигнал 11 в UNIX-подобных системах — это SIGSEGV, который означает нарушение доступа к памяти (segmentation fault).
- **src:000040+000033:** Это идентификаторы тестовых кейсов из корпуса, которые были использованы для генерации текущего входного файла. В данном случае файл был получен в результате "склеивания"(splice) двух входных файлов с идентификаторами 40 и 33.
- **time:512428:** Время в микросекундах, которое прошло с начала фаззинга до момента обнаружения данной ошибки.
- **execs:224439:** Количество выполнений (исполнений тестов), которое было произведено до обнаружения этой ошибки.
- **op:splice:** Операция, которая была применена к входному файлу, в данном случае "splice"означает, что AFL++ взял части из двух разных файлов корпуса и соединил их вместе, чтобы создать новый тестовый кейс.
- **rep:5:** Количество повторов (репликаций) данной операции, прежде чем был обнаружен краш. Это может означать, что ошибка возникла не с первого раза, а после нескольких попыток с различными мутациями.

Чтобы подробнее узнать о найденных ошибках нужно ввести команду

```
./fuzzgoat out/default/crashes/id:000011,...,
```

где:

– ./fuzzgoat — исполняемый файл тестируемой программы

– out/default/crashes/id:000011,... - путь до файла

Рассмотрим некоторые из ошибок, которые были найдены (рис.4.4):

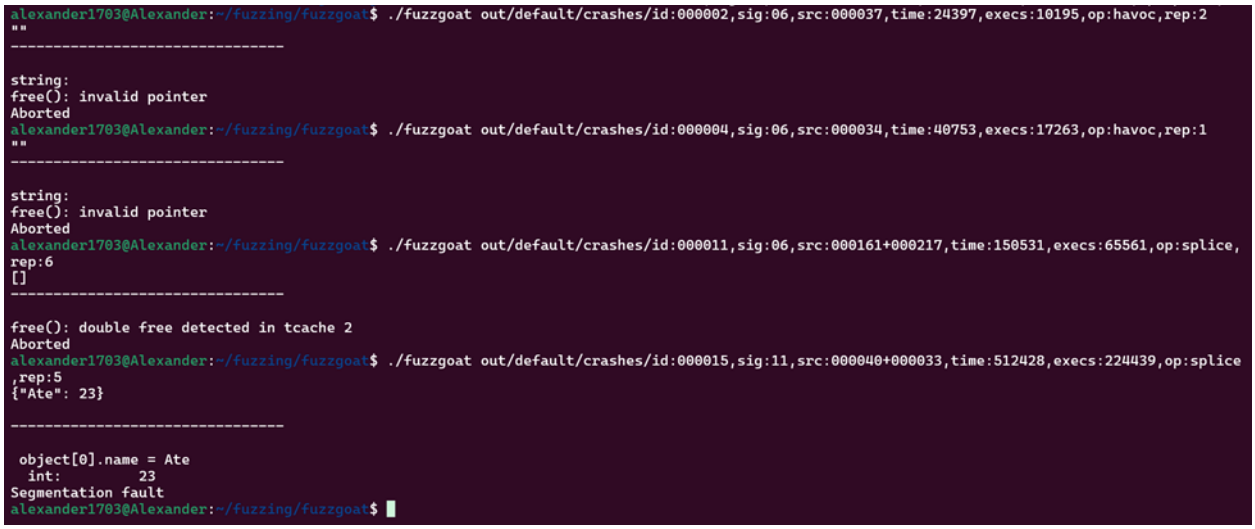

Рис.4.4. Информация об ошибках, выявленных в процессе тестирования

free(): invalid pointer

Попытка освободить память, которая не была выделена через malloc() или аналогичные функции, или уже была освобождена ранее. В данном случае, такая ошибка произошла дважды (id:000002 и id:000004):

free(): double free detected in tcache 2

Программа пытается освободить ту же область памяти более одного раза. Ошибка произошла при тесте с id:000011, что показывает, что определенные входные данные приводят к повторному освобождению уже освобожденной памяти.

Segmentation fault

Ошибка доступа к памяти вне выделенного сегмента. Обычно возникает, когда программа пытается читать или писать в память, к которой у нее нет доступа, или когда указатель на память не инициализирован. Ошибка обнаружена при тестировании с id:000015.

К сожалению, точное место ошибки AFL не возвращает. AFL работает с уже скомпилированным кодом и не имеет прямого доступа к исходному коду программы, что делает невозможным указание точной строки или функции, где произошла ошибка. Фаззер отслеживает лишь факт возникновения сбоя, без анализа причин его возникновения на уровне исходного кода. Для определения

конкретной причины и места сбоя необходимо использовать дополнительные инструменты, например профилировщики.

ЗАКЛЮЧЕНИЕ

ПОЛНОСТЬЮ дописываем (переписываем) с указанием также того, что сделал автор: рассмотрел различные виды фаззинга, изучил работу AFL++, развернул и на примере пофаззил, выявил, что не так все хорошо и далее общие слова (если Ваши...)

Фаззинг, включая AFL++, является мощным инструментом для обнаружения уязвимостей в программном обеспечении с низкими затратами на внедрение и автоматизацией процесса тестирования. Этот метод позволяет обнаруживать широкий спектр ошибок. Благодаря автоматическому созданию и вводу большого количества случайных или псевдослучайных данных фаззинг позволяет проводить тестирование программного обеспечения быстро и эффективно.

Однако у фаззинга есть и ограничения. Во-первых, он имеет ограниченное покрытие кода, что может привести к пропуску некоторых ошибок, особенно если они находятся за пределами тестового покрытия. Во-вторых, некоторые типы ошибок, такие как логические ошибки, могут быть сложными для обнаружения с помощью фаззинга. Также важно отметить, что для достижения оптимальных результатов фаззинг требует дополнительной настройки, включая выбор правильных стратегий мутации и анализ результатов.

В целом, фаззинг, включая инструменты типа AFL++, является хорошим дополнением к методам тестирования безопасности программного обеспечения. Он позволяет быстро обнаруживать уязвимости с низкими затратами на внедрение, что делает его привлекательным инструментом для разработчиков и исследователей. Однако для обеспечения полной безопасности программного обеспечения рекомендуется использовать фаззинг в сочетании с другими методами тестирования, такими как статический анализ кода и ручное тестирование.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Фаззинг-тестирование: QA Bible: https://vladislaveremeev.gitbook.io/qa_bible/vidy-metody-urovni-testirovaniya/fazzing-testirovanie-fuzz-testing
2. Wikipedia: American Fuzzy Lop (software): https://en.wikipedia.org/wiki/American_Fuzzy_Lop_(software)
3. Исследование инструментов фаззинга для генерации модульных тестов на Java - Александра Осипова: https://dspace.spbu.ru/bitstream/11701/32418/1/Osipova_report.pdf
4. More about AFL - afl-1: https://afl-1.readthedocs.io/en/latest/about_afl.html
5. Binary fuzzing strategies: what works, what doesn't - lcamtuf old blog: https://lcamtuf.blogspot.com/2014/08/binary-fuzzing-strategies-what-works.html
6. Как провести фаззинг REST API с помощью RESTler - Habr: https://habr.com/ru/companies/swordfish_security/articles/793514/l
7. Учебное пособие по фаззингу - guru99: https://www.guru99.com/ru/fuzz-testing.html
8. Обзор различных средств фаззинга как инструментов динамического анализа программного обеспечения. Мишечкин, М. В. - Молодой Ученый: https://moluch.ru/archive/186/47575/
9. Фаззинг на пальцах. Часть 1: идея, техника и мера - Habr: https://habr.com/ru/companies/ussc/articles/771778/