МИНОБРНАУКИ РОССИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО»

Институт компьютерных наук и кибербезопасности

Высшая школа технологий искусственного интеллекта

Доклад по дисциплине «Методы тестирования ПО»

Основные задачи технологии тестирования Фаззинг

Студент: _____ Салимли Айзек Мухтар Оглы

Преподаватель: _____ Курочкин Михаил Александрович

«____»_____ 20___ г.

Санкт-Петербург, 2025

Содержание

Введение **3**

1 Фаззинг **4**

2 Этапы фаззинга **5**

3 Метод серого ящика **6**

4 Санитайзеры **7**

5 Типы фаззеров **8**
5.1 На основе грамматики 8
5.2 На основе мутаций 9
5.3 На основе покрытия кода 10
5.4 Построение грамматики по данным 11

6 Проблемы фаззинга **11**

Заключение **12**

Введение

В данном докладе рассматривается применение технологии фаззинг (Fuzzing) для тестирования программного обеспечения на наличие уязвимостей, когда вместо ожидаемых входных данных программе передаются случайные или специально сформированные данные. Рассмотрены основные типы, этапы фаззинга и их эффективность, что такое метод серого ящика и зачем для фаззинга нужны санитайзеры.

1 Фаззинг

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 (англ. random testing), часто используемого для проверки проблем безопасности в программном обеспечении и компьютерных системах. В качестве входных данных при этом могут выступать обрабатываемые приложением файлы, информация, передающаяся по сетевым протоколам, функции прикладного интерфейса и т. д.

2 Этапы фаззинга

В основном этапы фаззинга состоят из:

1. Анализ исследуемого приложения
2. Разработка фаззера (опционально)
3. Генерация данных
4. Сам, фаззинг
5. Анализ результатов

3 Метод серого ящика

На стыке методов структурного и функционального тестирования находится метод «серого ящика». При тестировании данным методом исследователь не имеет полной спецификации программы и исходных кодов, как это бывает при тестировании методом «белого ящика», однако знаний о системе больше чем при тестировании методом «черного ящика». Фаззинг берет лучшее от двух существующих подходов. Мы не ограничиваем себя конечным количеством тестовых случаев и постоянно генерируем случайные входные данные для нашего кода, но при этом каждая новая последовательность использует информацию из предыдущих попыток для максимизации результата. Сразу возникает резонный вопрос — могут ли случайные данные что-то протестировать в действительности Предположим, мы хотим проверить компилятор C/C++ и посылаем на вход сгенерированные последовательности символов. Совершенно очевидно, что мы получим много ошибок от компилятора на стадии лексикографического анализа и парсера выражений, но при этом едва затронем оптимизацию и генерацию кода.

Рис. 1: Серый ящик.

4 Санитайзеры

Санитайзеры – это инструменты для динамического тестирования, помогающие в поиске самых разных ошибок в программах. Повод для использования санитайзеров: переполнение буферов (глобальных, на стеке или в куче), использование после освобождения, утечки памяти, обращение к неинициализированным переменным. Они также позволяют обнаруживать состояние гонки для потоков, ситуации взаимной блокировки, обращения по нулевому указателю, деление на ноль (куда же без него), переполнения для типов данных и некорректные битовые сдвиги. Санитайзеры очень помогают во время отладки. Для их использования достаточно скомпилировать исходники с включенной инструментацией, которая добавит специальные команды в исполняемый файл. По ним можно будет следить за ходом выполнения программы и состоянием памяти. При обнаружении ошибок будет сгенерирован отладочный вывод и программа завершит работу. Таким образом, мало просто сгенерировать фаззером некорректные входные данные, которые обрушат программу. Ошибку следует устранить, а для этого надо собрать максимум информации. Именно в этом и помогают санитайзеры. Их совместное использование повышает эффективность тестирования.

5 Типы фаззеров

Существует несколько основных типов фаззеров, они отличаются подходами к генерации входных данных. Некоторым для работы требуется стартовый набор примеров, другим нужны правила, по которым эти примеры могут быть выведены. Важную роль также играет случайность, которая позволяет фаззерам создавать новые входные данные.

5.1 На основе грамматики

Таким фаззерам для работы требуется определенный набор правил — грамматика для построения входных данных. Как только фаззеру будут известны эти правила, он сможет генерировать новые комбинации на их основе. При этом можно позволить себе иногда отклоняться от грамматики и не всегда следовать ей, подавая на вход тестируемой программы некорректные данные. Подобные фаззеры генерируют хорошие, достоверные последовательности, доля случайности в них относительно невелика. Однако важно понимать, что писать такие фаззеры — дело трудоемкое. Во-первых, грамматику следует определить самостоятельно (в редких случаях можно взять уже готовую). Во-вторых, качество работы фаззера будет напрямую зависеть от той грамматики, что ему передали.

5.2 На основе мутаций

Такой тип фаззеров на каждом этапе случайным образом изменяет входные данные из предыдущих попыток. При этом для старта ему требуется набор репрезентативных входов (корпус), который будет использоваться для дальнейших мутаций. В качестве корпуса можно взять данные пользователей или существующие тесты. Во время своей работы такой фаззер слегка изменяет готовые последовательности (переставляя биты и байты), комбинирует и сочетает их вариации и подает в программу. При этом никаких дополнительных усилий от разработчика тут не требуется (разве что написание специальных алгоритмов мутаций), но и качество входных данных у таких фаззеров обычно среднее.

5.3 На основе покрытия кода

Подобные фаззеры устроены по принципу генетического алгоритма и стремятся максимизировать покрытие тестового кода. С практической точки зрения это один из самых эффективных на сегодня типов фаззеров. На этой основе работает libFuzzer. LibFuzzer - это внутрипроцессный механизм фаззинга, управляемый охватом. LibFuzzer связан с тестируемой библиотекой и передает нечеткие входные данные в библиотеку через определенную точку входа фаззинга (также известную как «целевая функция»); Затем фаззер отслеживает, какие области кода достигаются, и генерирует мутации в корпусе входных данных, чтобы максимизировать покрытие кода.

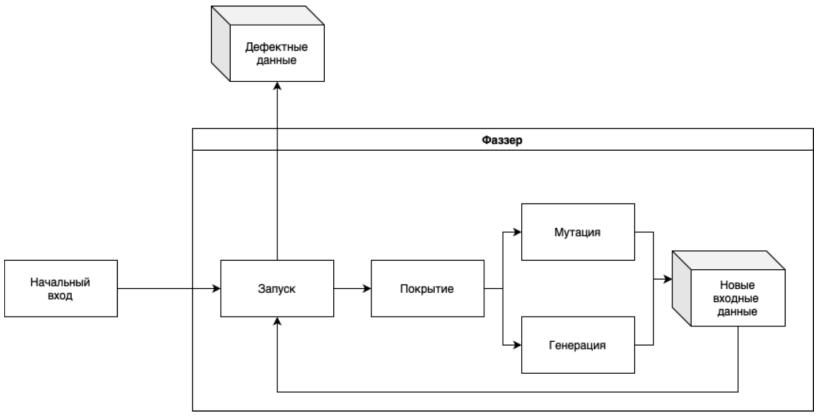

Рис. 2: Схема фаззера на основе покрытия кода.

5.4 Построение грамматики по данным

Также есть способ сочетать подходы на основе мутаций и на основе грамматики. Специалисты по обработке данных и машинному обучению могут попытаться построить грамматику на основе уже имеющейся информации. Однако оценить результат на выходе таких фаззеров зачастую непросто.

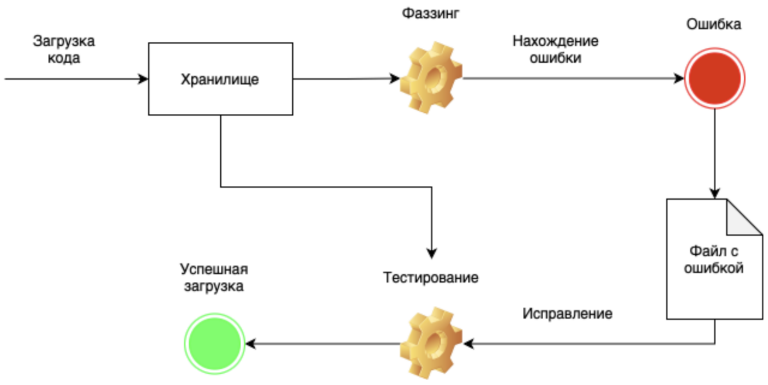

Рис. 3: Схема построения грамматики по данным.

6 Проблемы фаззинга

Основные проблемы фаззинга:

- Уязвимости в архитектуре

  Сложно предугадать, где найдёшь следующую уязвимость

- В результате фаззинга не всегда происходит падение

  Особенно при pool corruption

- Не все найденные уязвимости эксплуатабельны

  Таких воспроизводимых падений может быть очень много.

"Pool corruption ситуация когда обнаруживается повреждение или неправильное управление памятью, связанное с выделением и освобождением памяти в программных приложениях. Это может произойти, когда программа неправильно управляет памятью, что приводит к потенциальной уязвимости или сбоям в работе программы.

Заключение

Обнаружение даже одной уязвимости в популярном приложении за короткий срок в автоматическом режиме может сильно помочь для тестирования и выявления критических ошибок. Долгое время считалось, что фаззинг является слишком тяжеловесным подходом к обнаружению программных дефектов, и полученные результаты не оправдывают затраченных усилий и ресурсов. Однако, современные тенденции развития индустрии производства программного обеспечения позволяют сократить трудоемкость использования фаззинга. Тестирование безопасности в общем, и фаззинг в частности, нужны нам для того, чтобы, после выпуска продукта на рынок не подвергать пользователей атакам злоумышленников нашедших уязвимости, и как следствие, не подвергаться значительным финансовым затратам.