В тему SDL и фаззинга в частности - хочу подсветить пару возможностей современного инструментария, которые увидел и разобрал лишь недавно.

1. Автоматизация создания wrapper`ов для функций со сложным набором входных параметров (Structure Aware Fuzzing):

Предположим вам нужно профаззить не тривиальную для фаззинга функцию вида

XML * xml_parser (char* input)

а функцию с достаточно сложным набором входных параметров вида (на примере функции из postgres):

```
/*
 * Tests special kind of segment for in/out of polygon.
 * Special kind means:
 *    - point a should be on segment s
 *    - segment (a,b) should not be contained by s
 * Returns true if:
 *    - segment (a,b) is collinear to s and (a,b) is in polygon
 *    - segment (a,b) s not collinear to s. Note: that doesn't
 *      mean that segment is in polygon!
 */
```

static bool touched_lseg_inside_poly(Point *a, Point *b, LSEG *s, POLYGON *poly, int start)

Для фаззинга таковых функций существуют различные подходы, берущие начало из генерационного фаззинга (различные генераторы, в т.ч. на bnf-грамматиках и на использовании libprotobuf). Основной минус этих подходов в том, что порожденный сэмпл, позволивший открыть новый путь, на следующей итерации будет смутирован как обычный рандомный байтовый массив, а не как набор корректных структур. То есть имеется некий "разрыв связи" сэмплом, как порождением фаззера, и сэмплом/набором структур, поступающим на вход анализируемой функции. Мы с вами это неоднократно обсуждали, также можно обратиться вот к этому абзацу:

An advantage and disadvantage of using this library is that the input splitting happens dynamically, i.e. you don't need to define any structure of the input. This might be very helpful in certain cases, but would also make the corpus to be no longer in a particular format. For example, if you fuzz an image parser and split the fuzz input into several parts, the corpus elements will no longer be valid image files, and you won't be able to simply add image files to your corpus.

Чем это плохо? Например тем, что в этой парадигме не будут работать DSE-решатели Crusher (порождение байтового массива в генераторе и конвертацию его в реальный вход функции скорее всего не удастся представить в виде единой ассемблерной трассы, а также достаточно сложно порождать структуру вида массив переменной длины массивов переменной длины).

Что с этим можно делать? Пока что у нас нет (и не факт что появится для всех 100% случаев) системы, которая генерировала бы подобные врапперы автоматически. Но есть возможность помочь ей руками с помощью концепции нарезки "штампов" из порождаемого фаззером байтового массива. Что важно:

- код нарезки штампов в структуры на входе сложной функции встраивается в код фаззинг-цели, а соответственно это единый ассемблер - соответственно DSE точно будет работать, поскольку этап нерезки для него это просто ещё один участок кода выполняемой программы, а не какой-то внешний генератор;

- типовой метод нарезки штампов есть уже под c/c++ (**https://github.com/google/fuzzing/blob/master/docs/split-inputs.md#fuzzed-data-provider**), Java (https://blog.code-intelligence.com/engineering-jazzer), Python (https://github.com/google/atheris#fuzzeddataprovider). Это всё наработки гугла в рамках libfuzzer/llvm, но именно C-шный хидер (https://github.com/llvm/llvm-project/blob/main/compiler-rt/include/fuzzer/FuzzedDataProvider.h) можно просто линковать при работе в llvm-инфрастуктуре (не обязательно использовать именно libfuzer, главное - clang/llvm);

- **сходный механизм последние 3 месяца делает Николай (**@dhyannataraj**) из PostgresPro** (мы его уже тестируем, скоро вероятно он будет опубликован с tutorial). Этот механизм будет более глубоким в том смысле, что облегчит порождение сложных штампов (для сложных типов данных);

- создание штампов под обычные функции требует минимального обучения, но может существенно повысить глубину и честность покрытия фаззинг-тестами функций-операторов.

Итого: настоятельно рекомендуется ознакомиться и попробовать, можно совместно. Structure Aware Fuzzing очень актуальная штука, особенно в формате совмещения с DSE.

2. Фаззинг управляемого кода с целью нахождения ошибок логики (на примере Python) - зачем это нужно:

Основная цель фаззинга это поиск неизведанного вместо проверки известного (unit-тесты). В частности под неизведанным мы можем понимать нахождение не только buffer overwrite (потенциальная RCE) или (Div by Zero, потенциальный DOS), но и нахождение обходных путей в коде (traversal paths) и в целом любых странных ситуаций . Главное - придумать критерий "странности" и поместить некий abort() в точку проверки данного критерия - тогда фаззер пробившись в то место, куда он однозначно пробиться не должен, задетектить аборт и выведет сэмпл, приводящий к такой ситуации. Мои любимые примеры:

- ошибки логики – обходной путь вместо введения логина + пароля. Можно найти вставив abort() туда, куда поток выполнения никак попасть не должен при условии подачи рандомных данных;

- низкая энтропия шифрогенератора (см. Zerologon в Win AD). Можно определить вставив код расчета энтропии после шифрогенератора.

и т.д.

Примерно в данном направлении работает штука под название PropertyBasedTesting - попытка автоматизировать Unit-тесты. Хаотично мутируем параметры и проверяем assert`ом, что никогда не попадём туда, куда не следует. Пожалуйста ознакомьтесь с вводной частью и реализацией в мощной python-библиотеке анализа Hypothesis (https://hypothesis.readthedocs.io/en/latest/index.html). Общая идея во многом совпадает со сказанным выше, целью является не поиск крэша с перезаписью адреса возврата, а нарушение условий:

Hypothesis lets you write tests which instead look like this:

- For all data matching some specification.
- Perform some operations on the data.
- **Assert something about the result.**

This is often called property-based testing, and was popularised by the Haskell library Quickcheck.

It works by generating arbitrary data matching your specification and checking that your guarantee still holds in that case. If it finds an example where it doesn't, it takes that example and cuts it down to size, simplifying it until it finds a much smaller example that still causes the problem. It then saves that example for later, so that once it has found a problem with your code it will not forget it in the future.

Writing tests of this form usually consists of deciding on guarantees that your code should make - properties that should always hold true, regardless of what the world throws at you. Examples of such guarantees might be:

- Your code shouldn't throw an exception, or should only throw a particular type of exception (this works particularly well if you have a lot of internal assertions).
- If you delete an object, it is no longer visible.
- If you serialize and then deserialize a value, then you get the same value back.

Библиотека изначально хороша всем, за одним исключением - мутации рандомны, нет учета покрытия. Но! Теперь python-фаззер Atheris поддерживает создание врапперов посредством парадигм Hypothesis. То есть вы можете организовать враппер для входа в сложную python-функцию (Structure Aware Fuzzing) с помощью:

- парадигмы FuzzedDataProvider , обсуждавшейся выше (https://github.com/google/atheris#fuzzeddataprovider);

- парадигмы Hypothesis (https://github.com/google/atheris#use-with-hypothesis);

- получить честный Structure Aware Fuzzing с обратной связью (сертификаторы радуются);

- искать логические ошибки в своём коде.

Итого: настоятельно рекомендуется ознакомиться и попробовать, можно совместно.