Символьные предикаты безопасности в гибридном фаззинге

Алексей Вишняков
Илай Кобрин
Андрей Федотов

28 июня 2022 г.

ИСП РАН

Актуальность

- Развивающееся программное обеспечение неизбежно содержит ошибки
- Индустрия следует безопасному циклу разработки ПО (SDL) для своевременного обнаружения ошибок во время разработки
- Современное ПО включает в себя массу зависимостей с открытым исходным кодом
- Важно повышать безопасность открытого ПО путем поиска и исправления в нем ошибок

Гибридный фаззинг

- Фаззинг — поиск аварийных завершений, зависаний и нехватки памяти путем запуска программы на мутированных входных данных
  - Обратная связь по покрытию
- Динамическая символьная интерпретация — интерпретация трассы инструкций программы, где входным байтам сопоставляются свободные символьные переменные
  - Открытие новых путей (гибридный фаззинг)
  - Поиск ошибок (предикаты безопасности)
- Фаззер и символьный интерпретатор быстрее увеличивают покрытие, чем 2 фаззера*

* fuzzbench.com/reports/experimental/2021-07-03-symbolic

Цели работы

- Разработать метод автоматизированного поиска ошибок в бинарном коде методами динамической символьной интерпретации
- Метод должен применяться в контексте гибридного фаззинга
- Найти ошибки в проектах с открытым исходным кодом символьными предикатами безопасности

Динамическая символьная интерпретация

Sydr — инструмент динамической символьной интерпретации

- DynamoRIO — динамическая бинарная инструментация
- Triton — движок динамической символьной интерпретации
- Bitwuzla — SMT-решатель
- Каждый входной байт моделируется свободной **символьной переменной**
- Исследуется один путь выполнения программы
- Интерпретация инструкций порождает SMT формулы
- **Символьное состояние** отображает регистры и память в SMT формулы
- **Предикат пути** содержит ограничения пройденных переходов

Предикат безопасности

Предикат безопасности — дополнительные ограничения на предикат пути, описывающие условие возникновения ошибки (например, равенство делителя нулю)

Пример обнаружения ошибки выхода за левую границу массива:

Символьное состояние	**Инструкция**	**Множество формул**	**Предикат пути**
$rax = \phi_1, rbx = \phi_2$	—	$\varnothing$	*true*
$rax = \phi_1, rbx = \phi_3$	`add rbx, 2`	$\phi_3 = \phi_2 + 2$	*true*
$rax = \phi_1, rbx = \phi_3$	`cmp rbx, 2048` `jge .out`	$\phi_3 = \phi_2 + 2$	$\phi_3 < 2048$
$rax = \phi_1, rbx = \phi_3$	`cmp rbx, 0` `jl .out`	$\phi_3 = \phi_2 + 2$	$\phi_3 < 2048 \wedge \phi_3 \geq 0$
$rax = \phi_4, rbx = \phi_3$	`add rax, rbx`	$\phi_3 = \phi_2 + 2, \phi_4 = \phi_1 + \phi_3$	$\phi_3 < 2048 \wedge \phi_3 \geq 0$
$rax = \phi_4, rbx = \phi_3$	`mov [rbp+rax-0xe], 1`	$\phi_3 = \phi_2 + 2, \phi_4 = \phi_1 + \phi_3$	$\phi_3 < 2048 \wedge \phi_3 \geq 0$
Предикат безопасности $\phi_3 < 2048 \wedge \phi_3 \geq 0 \wedge \phi_4 < 0$			

Выход за границы массива

- Проверяем разыменование символьного адреса (который зависит от пользовательских данных)
- Границы определяем из теневого стека и кучи
- Не всегда можно определить обе границы
- Вычисляем базовый адрес массива эвристически из константной части символьного выражения адреса:
  - `[rdx + rax]`, где `rax` — константная база массива, а `rdx` — символьный индекс
- Оборачиваем функции копирования (`memcpy`, `memmove`, `memset` и т.д.) для обнаружения переполнения буфера

Целочисленное переполнение

- Целочисленное переполнение часто встречается в бинарном коде
- Проверка каждой арифметической операции замедляет анализ и приводит к ложным срабатываниям
- **Источник** — арифметическая инструкция
- **Сток** — опасное место использования переполненного значения (условные переходы, разыменование указателя, аргументы функций)
- Предикаты безопасности для беззнакового (CF) и знакового (OF) переполнения истинны, когда соответствующий флаг равен 1
- Знаковость определяется обратным слайсингом по инструкциям и анализом отобранных условных переходов (например, `jl` — знаковый переход)

Метод автоматизированного поиска ошибок символьными предикатами безопасности

- Гибридный фаззинг: фаззер libFuzzer и инструмент динамической символьной интерпретации Sydr
- Минимизация корпуса для получения меньшего числа файлов, дающих то же покрытие
- Проверка предикатов безопасности на каждом файле из минимизированного корпуса
  - Деление на нуль
  - Выход за границы массива
  - Целочисленное переполнение
- Верификация срабатываний предикатов безопасности на санитайзерах:
  - Срабатывание Sydr подтверждается на той же строке санитайзерами
  - Санитайзеры выдали новые предупреждения, которых не было на изначальном файле из корпуса

OSS-Sydr-Fuzz — фаззинг open source

OSS-Sydr-Fuzz — это наш форк Google OSS-Fuzz для фаззинга с помощью Sydr+libFuzzer (насчитывает **25+** проектов).
Более **25** найденных ошибок в открытых проектах, в том числе в PyTorch, TensorFlow, Tarantool.

Окружение для фаззинга описывается в Dockerfile:

- 3 версии исполняемого файла для каждой фаззинг цели
  - С инструментацией libFuzzer и санитайзерами
  - Для Sydr без инструментации, но с отладочной информацией (для верификации срабатывания со строчкой кода)
  - Для сбора покрытия llvm-cov
- Начальные корпуса входных файлов
- Словари

Более того, репозиторий содержит конфигурационные файлы для запуска гибридного фаззинга.

Найденные ошибки с помощью символьных предикатов безопасности

FreeImage

1. Беззнаковое целочисленное переполнение в `fseek` (BMP):

```
unsigned off_head, off_setup, off_image, i, temp;
...
fseek(ifp, off_setup + 792, SEEK_SET);
```

2. Знаковое целочисленное переполнение в `fseek` (TIFF):

```
fseek(ifp, offset + length - 4, SEEK_SET);
```

3. Неявное преобразование в `parse_tiff` (`thumb_offset` — `long`):

```
int parse_tiff(int base);
...
parse_tiff(thumb_offset + 12);
```

4. Беззнаковое целочисленное переполнение при вычислении ширины:

```
width = raw_width - left_margin - (get4() & 7);
```

5. Беззнаковое переполнение в условном переходе (смена потока управления):

```
if (*len * tagtype_dataunit_bytes[
        (*type <= LIBRAW_EXIFTAG_TYPE_IFD8) ? *type : 0] > 4)
    fseek(ifp, get4() + base, SEEK_SET);
```

xlnt: выход за границы массива

xlnt — библиотека для управления электронными таблицами XLSX

```
sector_chain
compound_document::follow_chain(sector_id start,
                                const sector_chain &table)
{
    auto chain = sector_chain();
    auto current = start;
    while (current >= 0)
    {
        chain.push_back(current);
        current = table[static_cast<std::size_t>(current)];
    }
    return chain;
}
```

xlnt: целочисленное переполнение

Беззнаковое целочисленное переполнение:

```
in_->seekg(static_cast<std::ptrdiff_t>(sector_data_start() +
        sector_size() * static_cast<std::size_t>(id)));
std::vector<byte> sector(sector_size(), 0);
```

1. Значение `sector_size()` очень велико — вызов конструктора `std::vector<byte> sector(sector_size(), 0)` приводит к аварийному завершению
2. Были найдены входные данные, где значение `id` равняется -1. Далее в том же файле исходного кода в функции `read_directory` происходит выход за границу массива: `entries_[static_cast<std::size_t>(current_entry_id)]`

unbound: беззнаковое целочисленное переполнение

unbound — проверяющий, рекурсивный, кэширующий распознаватель DNS

```
int sign = 0;
uint32_t i = 0;
uint32_t seconds = 0;
for(*endptr = nptr; **endptr; (*endptr)++) {
    switch (**endptr) {
    ...
    case '9':
        i *= 10;
    ...
}
```

hdp: деление на нуль

HDF — библиотека для работы с файлами формата hdf, который используется для хранения научных данных

```
int32 buf_size;
/* we are bounded above by VDATA_BUFFER_MAX */
buf_size = MIN(total_bytes, VDATA_BUFFER_MAX);
/* make sure there is at least room for one record
   in our buffer */
chunk = buf_size / hsize + 1;
```

miniz: целочисленное переполнение

miniz — библиотека сжатия данных без потерь (используется в PyTorch)

```
if (cdir_size < pZip->m_total_files * MZ_ZIP_CENTRAL_DIR_HEADER_SIZE)
    return mz_zip_set_error(pZip, MZ_ZIP_INVALID_HEADER_OR_CORRUPTED);
```

В результате переполнения проверка на корректность формата проходит, что может привести к ошибочной работе программы

Заключение

- Разработан метод поиска ошибок в бинарном коде методами динамической символьной интерпретации
- С помощью символьных предикатов безопасности в 5 проектах с открытым исходным кодом было обнаружено 11 новых ошибок:
  - 1 деление на нуль
  - 1 выход за границу массива
  - 7 целочисленных переполнений
  - 1 целочисленное переполнение, приводящее к аварийному завершению программы
  - 1 целочисленное переполнение, приводящее к переполнению буфера на куче

Спасибо за внимание! Вопросы?

e-mail: vishnya@ispras.ru

сайт: vishnya.xyz