Поиск ошибок целочисленного переполнения методами динамической символьной интерпретации

Кобрин Илай Александрович

25 мая 2022 г.

МГУ им. М.В. Ломоносова, кафедра системного программирования

Введение

- **Символьная интерпретация** — метод автоматического тестирования программ, при котором происходит интерпретация программы, где конкретным значениям переменных, зависящих от входных данных, сопоставляются символьные переменные, принимающие произвольные значения.
- **Фаззинг** — метод автоматического тестирования программ, при котором программе на вход подается множество наборов входных данных, после чего анализируется реакция программы и генерируются новые входные данные.
- Совместная работа фаззинга и символьной интерпретации называется **гибридным фаззингом**.

Построение предиката безопасности

Предикат безопасности — дополнительные условия на предикат пути, которые позволяют обнаружить ошибку в программе.

Символьное состояние	Инструкция	Множество формул	Предикат пути
$rax = \phi_1, rbx = \phi_2$	—	$S = \varnothing$	$\prod = true$
$rax = \phi_1, rbx = \phi_3$	`add rbx, 2`	$S = \{\phi_3 = \phi_2 + 2\}$	$\prod = true$
$rax = \phi_1, rbx = \phi_3$	`cmp rbx, 2048` `jge .out`	$S = \{\phi_3 = \phi_2 + 2\}$	$\prod = (\phi_3 < 2048)$
$rax = \phi_1, rbx = \phi_3$	`cmp rbx, 0` `jl .out`	$S = \{\phi_3 = \phi_2 + 2\}$	$\prod = (\phi_3 < 2048) \wedge (\phi_3 \geq 0)$
$rax = \phi_1, rbx = \phi_3$	`mov [rax + rbx], 1`	$S = \{\phi_3 = \phi_2 + 2, \phi_4 = \phi_1 + \phi_3\}$	$\prod = (\phi_3 < 2048) \wedge (\phi_3 \geq 0)$

Предикат безопасности

$$\prod_{security} = (\phi_3 < 2048) \wedge (\phi_3 \geq 0) \wedge (\phi_4 = 0)$$

Постановка задачи

Необходимо реализовать метод поиска ошибок целочисленного переполнения в бинарном коде с помощью динамической символьной интерпретации, который

- Строит предикат безопасности для ошибок целочисленного переполнения;
- Находит источники и стоки ошибок;
- Находит ошибки для различных размеров целочисленных типов
- Определяет знаковость результата арифметической операции, приводящей к переполнению, когда это возможно;
- Генерирует входные данные, приводящие к проявлению ошибки;
- В определенных случаях подбирает такие входные данные, чтобы последствия ошибки переполнения вероятнее приводили к аварийному завершению.

Известные решения

Разработанный метод был реализован на базе инструмента Sydr, использующего Triton и DynamoRIO.

Примеры других известных решений:

- KLEE
- Mayhem
- Google Sanitizers
- SAVIOR
- IntScope

Схема проверки предиката безопасности

- Составляем предикат безопасности
- Применяем слайсинг к предикату пути
- Конъюнкция предиката безопасности и получившегося предиката пути
- Проверка выполнимости конъюнкции с помощью Z3
- Если SAT - печатаем предупреждение и сохраняем файл

Понятие источника и стока

- Из-за слишком частых арифметических инструкций проверять на переполнение каждую слишком расточительно, а также может привести к большому количеству ложных срабатываний.
- **Источник** — арифметическая инструкция, в которой потенциально может произойти переполнение значения
- **Сток** — место, в котором используется переполненное ранее значение.
- Сначала находим потенциальный **источник**; если нашли **сток**, в котором он используется, проверяем на переполнение.
- Создаем два предиката для беззнакового и знакового переполнений, где проверяем равенство флагов `CF` и `OF` единице.

Виды sink'ов

Мы выделяем следующие места, где переполнение может привести к наиболее неприятным последствиям:

- Условные переходы
- Разыменование указателя
- Аргументы функций

Особенно опасно переполнение в аргументах `malloc`, `memcpy` и т.п. Для таких функций проверяем все символьные аргументы, для произвольных функций - только первые три аргумента.

Определение знаковости

- На уровне ассемблера не всегда возможно определить, является ли значение sink'а знаковым или беззнаковым.
- Для обнаружения знаковости используем следующий алгоритм:
  - Идем в обратном порядке по всем условным переходам, в которых были использованы переменные, которые есть в sink'е
  - Из найденных условных переходов узнаем знаковость
- Также можем определить знаковость, если соответствующие входные данные были обработаны функцией `strto*l`.

Дополнительные условия на предикат безопасности

- В аргументе `*alloc` функций хотим переполнить значение так, чтобы результат был меньше изначального конкретного значения (но не нулем)
- В аргументе `memcpy` и т.п. хотим переполнить так, чтобы результат был больше изначального конкретного значения

Пример Integer Overflow to Buffer Overflow

Входные данные: +00000000002

Ответ: +01073741825

Разрядность: 32 бита

```
1  int main() {
2      int size;
3      fscanf(stdin, "%d", &size);
4      if (size <= 0) return 1;
5      size_t i;
6      int *p = malloc(size * sizeof(int));
7      if (p == NULL) return 1;
8      for (i = 0; i < (size_t)size; i++) {
9          p[i] = 0;
10     }
11     printf("%d\n", p[0]);
12     free(p);
13 }
```

Juliet Dynamic

Для проверки эффективности работы предикатов была сделана тестовая система на основе набора тестов Juliet.

- Собираем и запускаем тесты, собираем результаты TP, TN, FP, FN на основе вывода Sydr
- Проверяем на основе сгенерированного файла корректность результата с помощью санитайзеров
- Выводим отдельно результаты на основе вывода Sydr и результаты, верифицированные санитайзерами

Результаты Juliet Dynamic

CWE	P=N	Текстовые ошибки			Верификация		
		TPR	TNR	ACC	TPR	TNR	ACC
Integer Overflow	2580	99.92%	90.89%	95.41%	98.10%	90.89%	94.50%
Integer Underflow	1922	99.90%	91%	95.45%	97.45%	91%	94.22%
Int Overflow to BOF	188	100%	100%	100%	100%	100%	100%

FreeImage

С помощью предикатов безопасности было найдено переполнение целого типа в программе FreeImage при вызове функции `fseek`:

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

xlnt

unbound

Заключение

- Реализованы 4 предиката безопасности
- Получена точность 95.59% в Juliet Dynamic на 11 CWE
- Найдено несколько ошибок в FreeImage