Обнаружение ошибок, возникающих при использовании динамической памяти после её освобождения*

[2] *С.А. Асрян <asryan@ispras.ru>*
[1,3,5,6] *С.С. Гайсарян <ssg@ispras.ru>*
[1] *Ш.Ф. Курмангалеев <kursh@ispras.ru>*
[4] *А.М. Агабалян <anna.aghabalyan@ispras.ru>*
[4] *Н.Г. Овсепян <narekhnh@ispras.ru>*
[4] *С.С. Саргсян <sevaksargsyan@ispras.ru>*
[1] *Институт системного программирования им. В.П. Иванникова РАН, 109004, Россия, г. Москва, ул. А. Солженицына, д. 25*
[2] *Институт проблем информатики и автоматизации НАН РА, Республика Армения, Ереван, 0014, ул. П.Севака, 1*
[3] *МГУ имени М.В. Ломоносова, факультет ВМК, 2119991 ГСП-1 Москва, Ленинские горы, 2-й учебный корпус*
[4] *Ереванский государственный университет Республика Армения, г. Ереван, 0025, ул. Алека Манукяна, 1*
[5] *Московский физико-технический институт, 141700, Московская область, г. Долгопрудный, Институтский пер., 9*
[6] *Национальный исследовательский университет «Высшая школа экономики» 101000, Россия, г. Москва, ул. Мясницкая, д. 20*

Аннотация. Существенная часть программного обеспечения написана на языках программирования C/C++. Программы на этих языках часто содержат ошибки: использования памяти после освобождения (Use After Free, UAF), переполнения буфера (Buffer Overflow) и др. В статье предложен метод обнаружения ошибок UAF, основанный на динамическом анализе. Для каждого пути выполнения программы предлагаемый метод проверяет корректность операций создания и доступа, а также освобождения динамической памяти. Поскольку применяется динамический анализ, поиск ошибок производится только в той части кода, которая была непосредственно выполнена. Используется символьное исполнение программы с применением решателей SMT (Satisfiability Modulo Theories) [12]. Это позволяет сгенерировать данные, обработка которых приводит к обнаружению нового пути выполнения.

* Работа поддержана грантом РФФИ № 17-01-00600

Ключевые слова: динамический анализ программ; покрытие кода; use-after-free.

DOI: 10.15514/ISPRAS-2018-30(3)-1

Для цитирования: Асрян С.А., Гайсарян С.С., Курмангалеев Ш.Ф., Агабалян А.М., Овсепян Н.Г., Саргсян С.С. Обнаружение ошибок, возникающих при использовании динамической памяти после освобождения. Труды ИСП РАН, том 30, вып. 3, 2018 г., стр. 7-20. DOI: 10.15514/ISPRAS-2018-30(3)-1

1. Введение

В программном обеспечении могут содержаться такие ошибки, как:

- использование динамической памяти после ее освобождения (Use After Free, UAF),
- переполнение буфера или кучи (buffer/heap overflow).

Поскольку огромная часть программного обеспечения используется в критически-важных областях человеческой деятельности, наличие ошибок может привести к серьезным последствиям. Существует ряд инструментов, помогающих решить эту проблему, используя методы статического [1, 2] и динамического анализа [3, 4, 5, 6, 7].

Статический анализ предоставляет возможность исследования программного кода без его выполнения. Недостатком этого метода является отсутствие информации о состоянии программы (регистры, трасса программы, входные данные и т.д.) во время выполнения. Это приводит к большому количеству ложных срабатываний. Поэтому данный метод в большинстве случаев используется до применения динамического анализа для выявления фрагментов программы, содержащих потенциальные ошибки.

Для поиска ошибок UAF инструмент [1] выполняет анализ, похожий на анализ доступных выражений (выражение $x+y$ является доступным в точке p, если вдоль любого пути от входной точки до точки p данное выражение вычисляется, а между этими вычислениями значения x и y остаются неизменными [11]). Производится обход всех путей в программе, чтобы обеспечить выполнение условия «определение объектов до их использования». В случаи неудовлетворения данного условия считается, что было выполнено ошибочное использование памяти и выводится ошибка.

Инструмент GUEB [2] основан на исследовании бинарного кода программы. Процесс анализа разделяется на два основных этапа. На первом этапе отслеживаются операции обращения к куче и присваивания адресов для проведения анализа набора данных (какой указатель к какому элементу кучи относится). На этом этапе информация {адрес, размер} сохраняется в множествах *alloc_set* и *free_set* при создании и освобождении памяти соответственно.

На втором этапе выполняется поиск ошибок UAF. Используя собранную информацию для каждой точки программы, инструмент строит множество

access_heap, которое содержит все элементы {адрес, размер} кучи, доступные в этой точке. Если пересечение *access_heap* и *free_set* является непустом множеством, считается, что найдена ошибка UAF.

Одной из причин популярности динамического анализа является возможность исследования программ во время выполнения. Благодаря этому возможен доступ к значениям регистров и содержимому памяти. Инструмент Avalanche [3] реализует итеративный анализ исполняемого кода программы, основанный на динамической бинарной трансляции. В процессе анализа инструмент вычисляет входные данные анализируемой программы с целью автоматического обхода всех достижимых путей в программе и обнаружения аварийных завершений программы.

Инструменты DangNull [4] и FreeSentry [5] фокусируются на обнаружении и обнулении указателей на динамическую область программы после их освобождения, предотвращая появление ошибок. Оба инструмента используют статическую инструментацию программ.

Undangle [6] также предотвращает ошибки использования памяти после освобождения. Этот инструмент помечает возвращаемые значения каждой функции распределения памяти и использует анализ помеченных данных для отслеживания этих меток. Далее при освобождении памяти проверяются, какие ячейки памяти ассоциированы с соответствующей меткой, и определяются висячие указатели (указатель с ненулевым значением, ссылающийся на освобожденный область памяти).

Инструмент Mayhem [7] основан на методе поиска ошибок в бинарном коде, объединяющем офлайн- и онлайн-подходы к символьному выполнению программы. Офлайн-подход предполагает последовательное исследование путей программы: при каждом новом запуске инструмент покрывает только один путь выполнения. Недостатком является повторное выполнение общего начального фрагмента пути при каждом запуске программы. Онлайн-подход, в свою очередь, исследует все возможные пути выполнения программы одновременно, что приводит к нехватке памяти в определенный момент времени.

Объединение этих двух подходов заключается в следующем: при достижении граничного значения расхода памяти создаются контрольные точки, исследование некоторых путей останавливается с сохранением информации о текущем состоянии выполнения, контекста символьного выполнения и конкретных входных данных. После освобождения ресурсов (завершились исследование некоторых путей), восстанавливается одна из контрольных точек (с использованием сохраненных данных воспроизводится конкретное выполнение до контрольной точки). Далее выполняется загрузка контекста символьного выполнения и начинается анализ нового пути. Данный подход позволяет избежать повторного символьного выполнения программы до места создания контрольной точки.

В данной статье мы будем рассматривать подход, основанный на динамическом анализе программы с использованием динамической инструментации [8, 9]. В работе описывается метод обнаружения ошибок UAF, который проверяет корректность использования указателей для всех возможных путей выполнения программы. Этот метод основан на алгоритме покрытия кода, используемом в SAGE [10], и использует инфраструктуру динамического анализатора Triton [9].

В рамках данной работы была выполнена модификация алгоритма покрытия кода, используемого в Triton, что привело к значительному росту производительности, а также была добавлена поддержка анализа программ, работающих с файловыми входными данными, которая отсутствовала в реализации Triton.

Второй раздел статьи посвящена описанию алгоритма покрытия кода программы в Triton и предлагаемой модификации. В третьем разделе рассматривается исходная реализация обнаружения ошибок UAF и ее объединение с динамическим покрытием кода. В четвертом разделе представлены результаты.

2. Алгоритм покрытия кода

2.1 Покрытие кода в Triton

В данной статье используется алгоритм увеличения покрытия кода программы, разработанный в компании Microsoft и используемый в инструменте SAGE [10]. Этот алгоритм частично реализован в Triton. Он состоит из двух этапов:

- выбор начальных входных данных и сборка ограничений для каждого пути выполнения программы;
- получение новых входных данных с помощью решения логических выражений, состоящих из ограничений, собранных на предыдущем этапе.

Рассмотрим пример программы на рис. 1.

```
void top(char input[4]) {
  int cnt=0;
  if (input[0] == 'b') cnt++;
  if (input[1] == 'a') cnt++;
  if (input[2] == 'd') cnt++;
  if (input[3] == '!') cnt++;
  if (cnt >= 3) abort(); // error
}
```

Рис. 1. Пример программы из статьи [10]
Fig. 1. An example of a program from [10]

Чтобы можно было исследовать все пути этой программы, на вход она должна получить строку “bad!”. Чтобы получить нужные данные, алгоритм начинает

свою работу, запуская программу на начальной входной строке, которая помещается в список входных данных. После первого запуска программы получается набор ограничений < $i0 \neq$ b, $i1 \neq$ a, $i2 \neq$ d, $i3 \neq$! >, где $i0$, $i1$, $i2$, $i3$ представляют ячейки памяти *input* [0], *input* [1], *input* [2] и *input* [3] соответственно.

В ходе работы алгоритма с помощью решения этих ограничений для каждого элемента из списка входных данных генерируются дочерние данные, удовлетворяющие этим ограничениям, которые далее помещаются в список входных данных. Для каждого элемента из этого списка программа заново запускается, и работа алгоритма возобновляется.

Этот процесс продолжается до тех пор, пока все элементы из списка входных данных не будут поочередно рассмотрены (псевдокод алгоритма приведен на рис. 3). Применив алгоритм для программы на рис. 1 с начальной входной строкой “good”, мы получим набор решений, представленный на рис. 2.

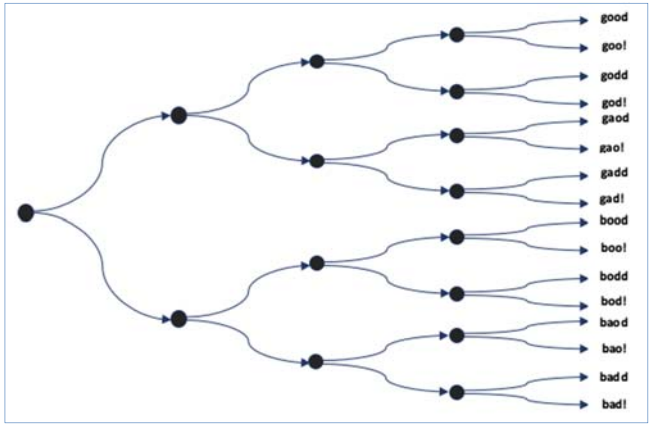

Рис. 2. Входные данные после каждой итерации алгоритма
Fig. 2. Input data after each iteration of the algorithm

```
1. runCodeCoverage(inputSeed):
2.   takeSnapshot()
3.   inputSeed.bound = 0
4.   inputList = {inputSeed}
5.   while inputList is not empty:
6.     input = getInputFromList(inputList)
7.     convertMemoryToSymbolic(input)
8.     childInputs = computeNewInputs(input)
9.     while input childInputs is not empty:
10.      inputList.append(input)
11.  if len(inputList)>0 and snapshotEnabled()
12.      restoreSnapshot()
```

```
1. computeNewInputs(input):
2.   // solve constraints using SMT solver
3.   childInputs = {}
4.   pc = ComputePathConstraint(input)
5.   for i in range(input.bound, pc.length):
6.     if (pc[0..(i-1)]) and not(pc[i]):
7.       I is solution for pc :
8.       newIn=updateWithoutOverwrite(input,I)
9.       newIn.bound = i
10.      childInputs.append(newIn)
11.  return childInputs
```

Рис. 3. Псевдокод алгоритма покрытия кода программы
Fig. 3. Pseudo code of a program code coverage algorithm

Поскольку работа данного алгоритма требует неоднократного запуска программы, в Triton реализована возможность сохранения состояния программы. Это позволяет значительно улучшить производительность выполнения. Кроме того, в Triton отсутствует часть алгоритма SAGE [10], предназначенная для уменьшения набора входных данных. Поэтому инструмент неоднократно запускает анализируемую программы на входных данных, которые не открывают новых путей. В описываемой работе в алгоритм покрытия кода был добавлен новый функционал, который позволяет достичь значительного роста производительности.

2.2 Модификация алгоритма покрытия

В оригинальной реализации алгоритма SAGE [10] в Triton после каждой итерации программа всегда получала на вход последний элемент из списка входных данных, не учитывая при этом количество открытых базовых блоков программы с помощью данного элемента. Это приводило к тому, что вместе с обработкой входных данных, которые имеют воздействие на покрытие кода программы, рассматриваются и те входные данные, с помощью которых не были открыты никакие новые пути в программе.

Поскольку в ходе работы алгоритма для каждого входного элемента генерируются ее дочерние данные, количество элементов в списке входных данных значительно увеличивается. Следовательно, для эффективного выполнения алгоритма требуется определение приоритетов для сгенерированных входных данных.

В предлагаемой модификации алгоритма каждому элементу из списка входных данных мы присваиваем вес, который представляет из себя количество базовых блоков программы, открытых этим элементом. В начале работы алгоритма входным данным присваивается нулевой вес. Во время первой итерации алгоритма подсчитываются весовые значения начальных входных данных.

После каждой итерации весовые значения обновляются следующим образом: весовые значения уже рассмотренных элементов передаются их дочерним элементам (входные данные, которые получились с помощью решения логических уравнений). Таким образом, применяется иерархический обход входных данных. Перед очередным запуском программы из списка выбирается элемент с наибольшим весом. Это позволяет значительно упростить дерево решений, как показано на рис. 4.

На рисунке видно, что после добавления весовых значений количество рассматриваемых входных данных уменьшилось почти вдвое, что в свою очередь приводит к существенному увеличению производительности работы алгоритма (на некоторых тестах производительность выросла почти на 90%).

Еще одним недостатком Triton была поддержка программ, принимающих на вход только аргументы командной строки. Для расширения набора

анализируемых программ нами была добавлена поддержка программ, использующих файлы как источник входных данных. Кроме того, была добавлена возможность определения конкретных диапазонов входных данных, которые в ходе анализа будут помечены как символьные.

Описанный подход к подсчету весовых значений не является единственным возможным, поэтому в дальнейших исследованиях будут рассматриваться и другие варианты определения этих значений.

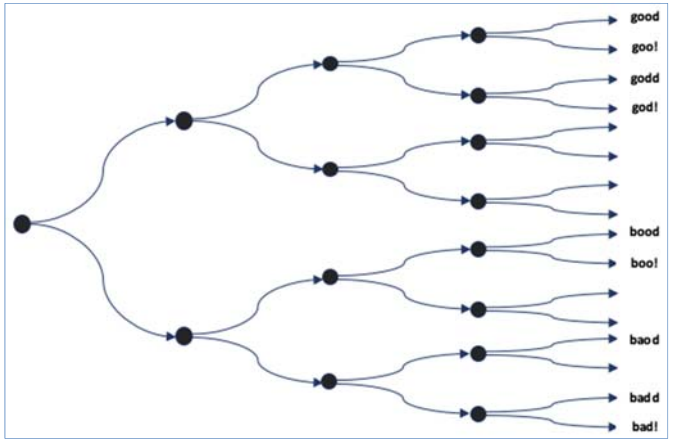

Рис. 4. Входные данные после каждой итерации алгоритма после добавления весов
Fig. 4. Input data after each iteration of the algorithm, after adding weight values

3. Поиск ошибок

Динамический анализ программы основан на исследовании программного обеспечения в процессе выполнения. Это дает возможность исследования программ с учетом определенных условий выполнения, а также позволяет использовать конкретные значения указателей. Одним из недостатков динамического анализа является требование наличия качественного покрытия кода. Однако в большинстве случаев для найденных ошибок возможна генерация входных данных, которые позволяют воспроизвести ошибку. Ошибка UAF характеризуется возникновением двух последовательных событий:

- создание висящих указателей (dangling pointers);
- доступ к памяти с использованием висящего указателя.

На рис. 5 приведён пример UAF. После проверки условия на строке 3 происходит освобождение памяти на который указывает переменная ptr (строка 4), затем управление переходит на строку 12, вследствие чего происходит повторное освобождение памяти.

3.1 Алгоритм поиска ошибок в Triton

Используя инструментацию программы, алгоритм отслеживает функции выделения (*malloc*) и освобождения (*free*) памяти. В начале работы алгоритма создаются два множества *(allocSET* и *freeSET*) для отслеживания участков памяти, которые были выделены и освобождены во время выполнения программы. Элементами данных множеств являются парами, имеющие вид («*адрес*, *размер*»).

Каждый раз при вызове *malloc/free* множества *allocSET* и *freeSET* обновляются путем добавления или удаления элементов с соответствующим адресом и размером выделенной памяти. При вызове функции *malloc* новый элемент («*адрес*_2, *размер*_2») добавляется в множество *allocSET* и удаляется из второго множества если данный элемент присутствует в множестве *freeSET* (т.е. есть совпадение как по адресу, так и по размеру).

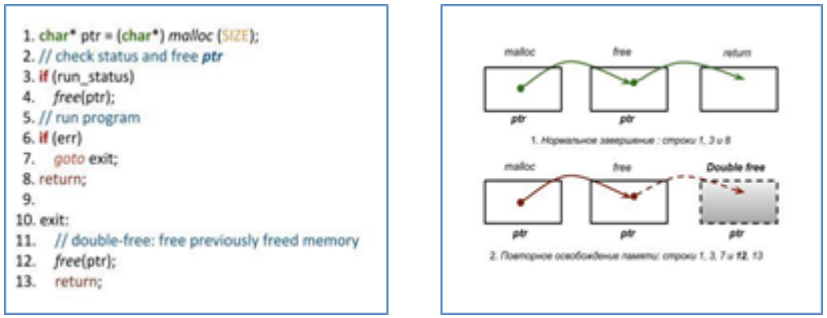

Рис. 5. Использование памяти после освобождения
Fog. 5. An example of UAF

Если данный элемент совпадает только по адресу, то перед обновлением множеств, выполняются дополнительные действия для обработки значений адреса и размера (если *размер*_2 < *размер*_1, то в *freeSET* добавляется элемент «*адрес*_2 + *размер*_2, *размер_1* - *размер*_2»). При вызове функции *free* соответствующий элемент перемещается из множества *allocSET* в множество *freeSET*.

Во время выполнения инструкций, осуществляющих доступ к памяти, проверяется наличие указателя в обоих множествах. Ошибки UAF фиксируются в двух случаях: элемент найден в множестве *freeSET* и его нет в множестве *allocSet*; один и тот же элемент встречается в *freeSET* больше одного раза. Работа алгоритма проиллюстрирована на рис. 6.

3.2 Предлагаемый метод

Для повышения эффективности поиска ошибок предлагается объединить два вышеописанных алгоритма. Объединенный метод позволяет искать ошибки UAF на разных путях программы, которые получаются из-за внедренных и

нетривиальных проверок в коде. На рис. 7 приведены примеры программ, на которых обнаружение повторного освобождения памяти невозможно без использования информации о покрытии кода (из-за присутствия условных переходов, которые будут выполнены только при выполнении программы с определенными входными данными).

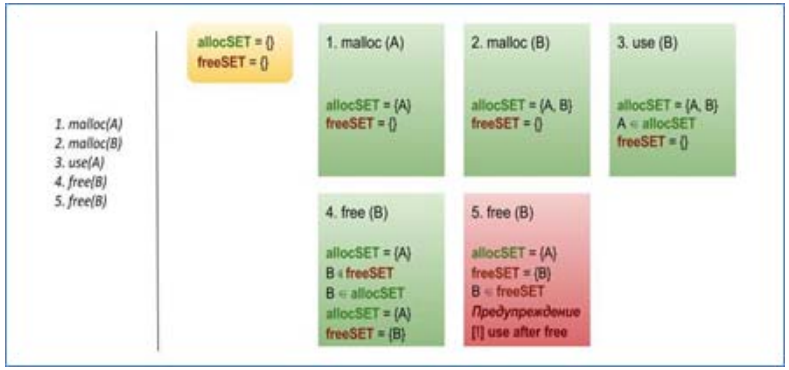

Рис. 6. Пример работы алгоритма
Fig. 6. An example of operation of the algorithm

```
1.  struct readFile {
2.    char* buffer;
3.    int   status;
4.  };
5.
6.  int search_key_in_text(char* str, char* key) {
7.    if (key != NULL && !strstr(str, key))
8.      return -1;
9.    else
10.     return 0;
11.
12.
13. int main(int argc, char* argv[]) {
14.   struct readFile rf;
15.   rf.buffer = (char*)malloc(sizeof(char));
16.   int fd = open(argv[1], O_RDONLY | O_CREAT);
17.   read(fd, rf.buffer, 50);
18.   printf("File cont : %s\n", rf.buffer);
19.   if (strlen(argv[2]) > 64 && !checkAttr(argv[2])) {
20.     rf.status = 11;
21.     free(rf.buffer);
22.   }
23.   if (!search_key_in_text(rf.buffer, argv[2])) {
24.     // do some stuff
25.   }
26.   free(rf.buffer); // Use after free
27. }
```

```
1.  int myatoi(char *p) {
2.    int i = 0;
3.    while (*p != '\x00' && *p >= '0' && *p <= '9') {
4.      i *= 10;
5.      i += *(unsigned char *)p - '0';
6.      p++;
7.    }
8.    return i;
9.  }
10.
11. int main(int argc, char *argv[]) {
12.   char* str = (char*) malloc (SIZE);
13.   int num;
14.   if (argc != 2) {
15.     printf("Usage: %s <string>\n", argv[0]);
15.     exit(-1);
17.   }
18.   strcpy(str, argv[1]);
19.
20.   num = myatoi(str);
21.   if (num >= 33 && ! (num % 2)) {
22.     printf("ok\n");
23.     free(str)
24.   }
25.   if (num > 1024 ) {
26.     printf("Number is out of range\n");
27.     free(str); // Use after free
28.   }
29.   return 0;
30. }
```

*Рис. 7. Ошибки повторного освобождения памяти. Первый пример основан на ошибке UAF в программе libssh. На первом примере ошибка может возникнуть на строке **26**, а на втором примере на строке **27***
Fig. 7. Errors of double memory freeing. The first example is based on the UAF error in the libssh program. In the first example, an error may occur on line 26, and in the second example on line 27

Для примеров, приведенных на рис. 7, ошибка UAF произойдет, если выполнение программ достигнет строк 21 и 23 для первой и второй программы соответственно. Использование предлагаемого метода позволяет найти входные данные для достижения нужного блока в коде (строки 20-21 и 22-23, рис. 7) и потом проверить данную часть на наличие ошибок UAF.

3.3 Сравнение подходов динамического анализа

В табл. 1 подход, описываемый в данной статье, сравнивается в подходами, применяемыми в Mayhem и Triton.

Табл. 1. Результаты сравнения
Table 1. Results of comparison

	Предлагаемый подход	**Mayhem**	**Triton UAF**
Использование покрытия кода	+	+	-
Офлайн-подход символьного выполнения	+	+	+
Онлайн-подход символьного выполнения	-	+	-
Приоритеты обрабатываемых входных данных	+	+	-
Поддержка файлов, в качестве входных данных	+	+	-

4. Результаты

Предлагаемый метод был протестирован на синтетических тестах, в том числе и на приведённых примерах (рис. 1, 7), результаты тестирования приведены на рис. 8. Данные результаты показывают, что на синтетических тестах, по сравнению с реализацией Triton, производительность выросла примерно на 80%. Запуск анализа на реальных программах показал, что в большинстве случаев количество символьных уравнений становится настолько большим, что алгоритм покрытия кода Triton не может решить полученные уравнения для всех путей.

Для проверки предложенного подхода нами были специально внесены ошибки UAF в исходный код реальных проектов. Были исследованы проекты: gvgen из пакета graphviz, jasper из пакета libjasper-runtime и gif2rgb из пакета giflib. На данных проектах ошибки были найдены на различных уровнях встраивания. В случае программы gvgen встраивание было выполнено за пределами одной функции, максимальная глубина составляло три уровня (функции). Код встраивания в данном случае представлял с собой условное выражение,

связанное с входными данными, и код самой ошибки. Выполнение этого условия приводило к воспроизведению данной ошибки.

В проектах jasper и gif2rgb из-за сложности полученных символьных уравнений встраивание было выполнено в пределах только одной функции. Код встраивания был непосредственно кодом ошибки. Также были выделены отдельные участки кода из реальных программ содержащие UAF на которых данный подход смог найти соответствующие ошибки.

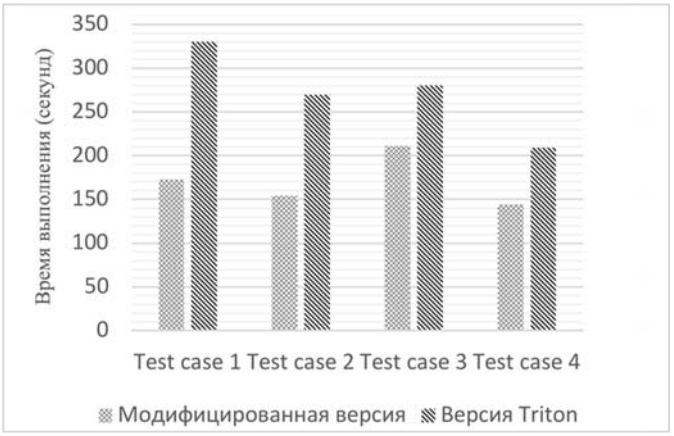

Рис. 8. Результаты сравнения относительно времени выполнения анализа
Fig. 8. Comparison results relative to the analysis execution time

5. Заключение

В статье представлен метод обнаружения ошибок UAF, возникающих при неправильной обработке указателей на динамическую память. Метод реализован с помощью инфраструктуры Triton [9] на базе алгоритма [10] и алгоритма обнаружения ошибок UAF. После проведенных модификаций и улучшений существующей реализации был получен прирост производительности выполнения анализа.

Список литературы

[1]. D. Dewey, B. Reaves, P. Trainor. Uncovering Use-After-Free Conditions in Compiled Code. In Proc of the 10th International Conference on Availability, Reliability and Security (ARES), 2015, pp. 90-99

[2]. J. Feist, L. Mounier, ML. Potet. Statically detecting use after free on binary code. Journal of Computer Virology and Hacking Techniques, vol. 10, issue 3, 2014, pp 211-217

[3]. И.К. Исаев, Д.В. Сидоров, А.Ю. Герасимов, М.К. Ермаков. Avalanche: Применение динамического анализа для автоматического обнаружения ошибок в программах, использующих сетевые сокеты. Труды ИСП РАН, том 21, 2011 г., стр. 55-70.

[4]. B. Lee, Ch. Song, Y. Jang, T. Wang. Preventing Use-after-free with Dangling Pointers Nullification. In Proc of the Network and Distributed System Security Symposium, 2015, https://www.ndss-symposium.org/ndss2015/ndss-2015-programme/preventing-use-after-free-dangling-pointers-nullification/, дата обращения 05.05.2018

[5]. Yves Younan. FreeSentry: Protecting Against Use-After-Free Vulnerabilities Due to Dangling Pointers. In Proc of the Network and Distributed System Security Symposium, 2015, https://www.ndss-symposium.org/ndss2015/ndss-2015-programme/freesentry-protecting-against-use-after-free-vulnerabilities-due-dangling-pointers/, дата обращения 05.05.2018

[6]. J. Caballero, G. Grieco, M. Marron, A. Nappa. Undangle: Early Detection of Dangling Pointers in Use-After-Free and Double-Free Vulnerabilities. In Proceedings of the 2012 International Symposium on Software Testing and Analysis, 2012, pp. 133-143

[7]. Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert and David Brumley. Unleashing MAYHEM on Binary Code. In Proceedings of the 2012 IEEE Symposium on Security and Privacy, 2012, pp. 380-394

[8]. Pin – A Dynamic Binary Instrumentation Tool, https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool, дата обращения 05.05.2018

[9]. Triton – Dynamic Binary Analysis Framework, https://triton.quarkslab.com/, дата обращения 05.05.2018

[10]. P. Godefroid, M. Y. Levin, D. Molnar. Automated Whitebox Fuzz Testing. In Proceedings of NDSS'2008 (Network and Distributed Systems Security), 2008, pp. 151-166.

[11]. A. Aho, J. Ullman, R. Sethi, M. S. Lam. Compilers: Principles, Techniques, and Tools. Addison Wesley; 2nd edition, September 10, 2006, 1000 p.

[12]. Leonardo de Moura, Nikolaj Bjørner. Z3: an efficient SMT solver. In Proceedings of the 14th international conference on Tools and algorithms for the construction and analysis of systems, 2008, pp. 337-340

Dynamic Detection of Use After Free Bugs

[2] *S.A Asryan <asryan@ispras.ru>*
[1,3,5,6] *S.S. Gaissaryan <ssg@ispras.ru>*
[1] *Sh. F. Kurmangaleev <kursh@ispras.ru>*
[4] *A.M. Aghabalyan <anna.aghabalyan @ispras.ru>*
[4] *N.H. Hovsepyan <narekhnh@ispras.ru>*
[4] *S.S Sargsyan <sevaksargsyan@ispras.ru>*
[1] *Ivannikov Institute for System Programming of RAS,*
25, Alexander Solzhenitsyn str., Moscow, 109004, Russia
[2] *Institute for Informatics and Automation Problems of NAS RA,*
1, P. Sevak str., Yerevan, 0014, Republic of Armenia,
[3] *Lomonosov Moscow State University,*
GSP-1, Leninskie Gory, Moscow, 119991, Russia
[4] *Yerevan State University*
1, Alex Manoogian str., Yerevan, 0025, Republic of Armenia
[5] *Moscow Institute of Physics and Technology (State University),*
9 Institutskiy per., Dolgoprudny, Moscow Region, 141700, Russia
[6] *National Research University Higher School of Economics (HSE)*
11 Myasnitskaya str., Moscow, 101000, Russia

Abstract. The article describes new method of use after free bug detection using program dynamic analysis. In memory-unsafe programming languages such as C/C++ this class of bugs mainly accurse when program tries to access specific area of dynamically allocated memory that has been already freed. This method is based on combination of two basic components. The first component tracks all memory operations through dynamic binary instrumentation and searches for inappropriate memory access. It preserves two sets of memory address for all allocation and free instructions. Using both sets this component checks whether current memory is accessible through its address or it has been already freed. It is based on dynamic symbolic execution and code coverage algorithm. It is used to maximize the number of execution paths of the program. Using initial input, it starts symbolic execution of the target program and gathers input constraints from conditional statements. The new inputs are generated by systematically solving saved constraints using constraint solver and then sorted by number of basic blocks they cover. Proposed method detects use after free bugs by applying first component each time when second one was able to open new path of the program. It was tested on our synthetic tests that were created based on well-known use after free bug patterns. The method was also tested on couple of real projects by injecting bugs on different levels of execution.

Keywords: program dynamic analysis; use after free bug; dynamic symbolic execution; code coverage; instrumentation.

DOI: 10.15514/ISPRAS-2018-30(3)-1

For citation: Asryan S.A., Gaissaryan S.S., Kurmangaleev Sh.F., Aghabalyan A.M., Hovsepyan N.H., Sargsyan S.S. Dynamic detection of Use After Free bugs. Trudy ISP RAN/Proc. ISP RAS, vol. 30, issue 3, 2018. pp. 7-20 (in Russian). DOI: 10.15514/ISPRAS-2018-30(3)-1

References

[1]. D. Dewey, B. Reaves, P. Trainor. Uncovering Use-After-Free Conditions in Compiled Code. In Proc of the 10th International Conference on Availability, Reliability and Security (ARES), 2015, pp. 90-99

[2]. J. Feist, L. Mounier, ML. Potet. Statically detecting use after free on binary code. Journal of Computer Virology and Hacking Techniques, vol. 10, issue 3, 2014, pp 211-217

[3]. Ildar Isaev, Denis Sidorov, Alexander Gerasimov, Mikhail Ermakov. Avalanche: Using dynamic analysis for automatic defect detection in programs based on network sockets. Trudy ISP RAN/Proc. ISP RAS, vol. 21, 2011, pp. 55-70 (in Russian).

[4]. B. Lee, Ch. Song, Y. Jang, T. Wang. Preventing Use-after-free with Dangling Pointers Nullification. In Proc of the Network and Distributed System Security Symposium, 2015, https://www.ndss-symposium.org/ndss2015/ndss-2015-programme/preventing-use-after-free-dangling-pointers-nullification/, accessed at 05.05.2018

[5]. Yves Younan. FreeSentry: Protecting Against Use-After-Free Vulnerabilities Due to Dangling Pointers. In Proc of the Network and Distributed System Security Symposium, 2015, https://www.ndss-symposium.org/ndss2015/ndss-2015-programme/freesentry-protecting-against-use-after-free-vulnerabilities-due-dangling-pointers/, accessed at 05.05.2018

[6]. J. Caballero, G. Grieco, M. Marron, A. Nappa. Undangle: Early Detection of Dangling Pointers in Use-After-Free and Double-Free Vulnerabilities. In Proceedings of the 2012 International Symposium on Software Testing and Analysis, 2012, pp. 133-143

[7]. Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert and David Brumley. Unleashing MAYHEM on Binary Code. In Proceedings of the 2012 IEEE Symposium on Security and Privacy, 2012, pp. 380-394

[8]. Pin – A Dynamic Binary Instrumentation Tool, https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool, accessed at 05.05.2018

[9]. Triton – Dynamic Binary Analysis Framework, https://triton.quarkslab.com/, accessed at 05.05.2018

[10]. P. Godefroid, M. Y. Levin, D. Molnar. Automated Whitebox Fuzz Testing. In Proceedings of NDSS'2008 (Network and Distributed Systems Security), 2008, pp. 151-166.

[11]. A. Aho, J. Ullman, R. Sethi, M. S. Lam. Compilers: Principles, Techniques, and Tools. Addison Wesley; 2nd edition, September 10, 2006, 1000 p.

[12]. Leonardo de Moura, Nikolaj Bjørner. Z3: an efficient SMT solver. In Proceedings of the 14th international conference on Tools and algorithms for the construction and analysis of systems, 2008, pp. 337-340