Методы и инструменты выявления уязвимостей при разработке безопасного программного обеспечения

Батраева И. А., Шишков Н.А., Селезнев А.Д.
batraevaia@info.sgu.ru, nik.shishkov2004@mail.ru ,
aleksandrselezny0v@yandex.ru
Саратовский государственный университет имени Н.Г. Чернышевского

Аннотация. В статье рассматриваются проблемы, связанные с разработкой программ, не содержащих код с неопределенным поведением и проверкой уже разработанных программ на такие ошибки. Приведены примеры, возникновения ошибок и методы и инструменты, используемые для их поиска.

Ключевые слова: безопасное программное обеспечение, статический анализ, динамический анализ

Одной из серьезных проблем при разработке программного обеспечения стала проблема создания безопасного кода. Под таким кодом подразумевается код, который не содержит уязвимостей, способствующих утечке данных, а также не содержит фрагментов, приводящих к неопределенному поведению программы (например, гарантировано не будет деления на ноль, переполнения разрядной сетки и т.п.)

Небезопасный код возникает зачастую как результат плохо сформулированных технических заданий, ошибок программистов, уязвимостей в сторонних библиотеках, которые используются в проектах. Основным и идеальным средством разработки такого кода можно считать безопасный компилятор, удовлетворяющий введенному в действие с 1 апреля 2024 года ГОСТу Р 71206-2024 «Защита информации. Разработка безопасного программного обеспечения. Безопасный компилятор языков С/С++» [1].

Однако, стандартов на другие языки программирования в настоящее время еще нет, а компилятор для С/С++ в настоящее время только разрабатывается [2], поэтому для анализа и поиска уязвимостей уже написанного ПО используются инструменты анализа кода. Основные два принципа исследования кода – это статический и динамический анализ.

Статический анализ заключается в проверке исходного кода без его выполнения. Инструменты статического анализа ищут потенциальные уязвимости, ошибки и проблемы в коде, такие как неинициализированные переменные, утечки памяти, неправильные логические условия, выявляют некорректную реализацию криптографической защиты, возможность инъекции, слабый контроль доступа и использование устаревших алгоритмов защиты.

Наиболее известными представителями группы статических анализаторов являются: Clang Static Analyzer, Coverity, Klocwork Insight, Svace, PVS-Studio.

Рассмотрим метод обнаружения уязвимостей с использованием clang static analyzer версии 17.0 на примере библиотеки LibTIFF 4.0.3, которая используется при просмотре формата tiff

1) Собрать исходные файлы библиотеки с помощью анализатора командой scan-build-17 make, данный анализатор покажет предупреждения о возможных ошибках в коде (см. рис.1):

```
tiffcrop.c:5653:15: warning: Division by zero [core.DivideZero]
 5653 |          x1 = TIFFhowmany(iwidth, owidth);
      |               ^~~~~~~~~~~~~~~~~~~~~~~~~~
tiffcrop.c:160:61: note: expanded from macro 'TIFFhowmany'
  160 | #define TIFFhowmany(x, y) ((((uint32)(x))+(((uint32)(y))-1))/((uint32)(y)))
      |                                                             ^~~~~~~~~~~~~~
tiffcrop.c:5662:12: warning: Value stored to 'orientation' is never read [deadcode.DeadStores]
 5662 |           orientation = ORIENTATION_PORTRAIT;
      |           ^             ~~~~~~~~~~~~~~~~~~~~
```

Рис. 1. Процесс сборки через анализатор.

2) Результаты сборки проекта привести к формату Html с помощью команды scan-view-17 /path/to/result. Этот формат позволяет использовать фильтрацию ошибок по их типу и создает ссылки на проблемные участки кода с их кратким описанием (см. рис.2.)

Bug Type	Quantity	Display?
All Bugs	**128**	☑
API		
Argument with 'nonnull' attribute passed null	2	☑
Logic error		
Assigned value is garbage or undefined	1	☑
Dereference of null pointer	7	☑
Division by zero	10	☑
Result of operation is garbage or undefined	2	☑

Bug Group	Bug Type	File	Function/Method	Line	Path Length			
API	Argument with 'nonnull' attribute passed null	tools/tiffcrop.c	writeSingleSection	7107	37	View Report	Report Bug	Open File
API	Argument with 'nonnull' attribute passed null	tools/tiffcrop.c	writeCroppedImage	7786	40	View Report	Report Bug	Open File
Logic error	Uninitialized argument value	contrib/iptcutil/iptcutil.c	main	556	46	View Report	Report Bug	Open File
Logic error	Result of operation is garbage or undefined	tools/tiffcmp.c	PrintIntDiff	426	39	View Report	Report Bug	Open File
Logic error	Result of operation is garbage or undefined	libtiff/tif_luv.c	multiply_ms	1216	19	View Report	Report Bug	Open File
Logic error	Division by zero	libtiff/tif_write.c	TIFFWriteEncodedTile	406	15	View Report	Report Bug	Open File
Logic error	Division by zero	libtiff/tif_read.c	TIFFStartTile	1005	24	View Report	Report Bug	Open File
Logic error	Division by zero	libtiff/tif_write.c	TIFFWriteEncodedTile	408	17	View Report	Report Bug	Open File
Logic error	Division by zero	libtiff/tif_write.c	TIFFWriteScanline	119	23	View Report	Report Bug	Open File
Logic error	Division by zero	libtiff/tif_read.c	TIFFStartTile	1002	22	View Report	Report Bug	Open File
Logic error	Division by zero	libtiff/tif_write.c	TIFFWriteRawStrip	319	20	View Report	Report Bug	Open File

Рис. 2. Отчёт от clang по найденным ошибкам.

3) Для анализа кода отфильтруем ошибки по расположению файлов и откроем в данном случае отчет о делении на 0 в файле libtiff/write.c на строке 119 (см. рис.3.):

Logic error	Division by zero	libtiff/tif_read.c	TIFFStartTile	1002	22	View Report	Report Bug	Open File
Logic error	Division by zero	libtiff/tif_read.c	TIFFStartTile	1005	24	View Report	Report Bug	Open File
Logic error	Division by zero	libtiff/tif_write.c	TIFFWriteRawStrip	319	20	View Report	Report Bug	Open File
Logic error	Division by zero	libtiff/tif_write.c	TIFFWriteScanline	119	23	View Report	Report Bug	Open File
Logic error	Division by zero	libtiff/tif_write.c	TIFFWriteEncodedTile	408	17	View Report	Report Bug	Open File
Logic error	Division by zero	libtiff/tif_write.c	TIFFWriteEncodedTile	406	15	View Report	Report Bug	Open File

Рис. 3. Нахождение интересующего файла в отчёте.

4) Комментарии в расшифровке (см. рис. 4) описывают условия возникновения ошибки:

```
if (strip >= td->td_stripsperimage && imagegrew)
```
16 ← Assuming 'strip' is >= field 'td_stripsperimage' →
```
        td->td_stripsperimage =
```
20 ← The value 0 is assigned to field 'td_stripsperimage' →
```
            TIFFhowmany_32(td->td_imagelength,td->td_rowsperstrip);
```
17 ← Taking true branch →

18 ← Assuming the condition is false →

19 ← '?' condition is false →
```
(strip % td->td_stripsperimage) * td->td_rowsperstrip;
```
21 ← Division by zero

Рис.4. Комментарии анализатора по найденной ошибке.

Ошибка является уязвимостью CVE-2014-8130, которая позволяет удаленным злоумышленникам вызвать отказ в обслуживании (ошибка деления на ноль и падение приложения) с помощью созданного изображения TIFF, неправильно обрабатываемого функцией TIFFWriteScanline.

Из недостатков статических анализаторов можно выделить слабую диагностику утечек памяти и параллельных ошибок. Связано это с тем, что для их корректного выявления необходимо виртуально выполнить часть программы, поэтому анализатор может обнаруживать лишь простые случаи. Более эффективным способом выявления утечек памяти и параллельных ошибок является использование инструментов динамического анализа.

Динамический анализ предполагает анализ исполняемого файла, а не исходного кода программы, а значит в большинстве случаев не нужен доступ к исходному коду, что является огромным плюсом динамического анализатора. Так же у динамического анализатора практически отсутствует ложные срабатывания, так как он констатирует факт возникновения ошибки, а не возможность возникновения.

Инструменты динамического анализа могут обнаруживать проблемы, связанные с управлением памятью, доступом к данным и другими аспектами выполнения программы.

Наиболее известными разработками в области динамического анализа являются Valgrind, Address Sanitizer, Chess и созданный в ИСП РАН анализатор помеченных данных «Блесна». Эти продукты оценивают программу по таким характеристикам как используемые ресурсы (время выполнения программы или ее отдельных модулей, количество внешних запросов, количество используемой оперативной памяти и других ресурсов), степень покрытия кода тестами, наличие программных ошибок типа деления на ноль, разыменования нулевого указателя, утечки памяти и т.п.

Рассмотрим динамическое тестирование программы с использованием Valgrind 3.21 на следующем примере:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
void run_test(int i)
{
  int delta = 123;
  char* mem = malloc(1024);
  strcpy(mem, "i = ");
  printf("%s %d\n", mem, i + delta);
  /* free(mem); */
}

void main()
{
  int i;
  for(i = 0; i < 10; i++) run_test(i);
}
```

В программе допущена типовая ошибка начинающего программиста – пропущен оператор освобождения памяти. Зачастую, разработчики не обращают на нее внимания, считая, что, когда программа закончит работу, память будет освобождена. Тем не менее, есть ситуации, когда это может

привести к неустойчивой работе программы. Анализатор valgrind позволит отследить эту утечку при отладке программы, показав следующе сообщение (см. рис. 5):

```
==1948== HEAP SUMMARY:
==1948==     in use at exit: 10,240 bytes in 10
blocks
==1948==   total heap usage: 11 allocs, 1 frees,
11,264 bytes allo...
==1948==
==1948== LEAK SUMMARY:
==1948==    definitely lost: 10,240 bytes in 10
blocks
```

Рис.5. Вывод анализатора Valgrind при запуске на нашем тестовом примере

Видно, что размещено было 11 блоков памяти, а по окончании работы программы 10 из них так и не были освобождены. Вызов Valgrind с опцией `--leak-check=full` для дополнительной информации покажет точное место утечки памяти (см. рис. 6).

```
==2047== 10,240 bytes in 10 blocks are definitely
lost in loss recor...
==2047==    at 0x4C2AF1F: malloc (in
/usr/lib/valgrind/vgpreload_mem...
==2047==    by 0x400561: run_test (vgcheck.c:8)
==2047==    by 0x4005AF: main (vgcheck.c:18)
```

Рис. 6. Вывод рис.6 с опцией leak-check=full

Таким образом, можно сделать вывод, что использование методов и инструментов обнаружения неопределенного поведения кода при обучении программистов повысит качество их подготовки, так как более наглядно покажет, к чему могут привести такие ошибки при работе программы

Список литературы

[1]. ГОСТ Р 71206-2024 «Защита информации. Разработка безопасного программного обеспечения. Безопасный компилятор языков C/C++»// URL: https://www.gostinfo.ru/catalog/Details/?id=7524417

[2]. Дунаев П.Д., Синкевич А.А., Гранат А.М., Батраева И.А., Миронов С.В., Шугалей Н.Ю. Разработка безопасного компилятора на основе Clang. Труды Института системного программирования РАН, том 36, вып. 4, 2024, стр. 27-40. DOI: 10.15514/ISPRAS-2024-36(4)-3.