Лабораторная работа №3
Динамический анализ кода программ

Цель работы

Получить навыки динамического анализа кода программ с помощью инструментальных средств.

Использование Valgrind для анализа ошибок работы с памятью

Для работы с Valgrind проект должен быть собран в отладочной конфигурации. В противном случае будет невозможно определить место ошибки. Результаты работы выводятся в текстовом виде.

Valgrind включает в себя несколько компонентов. С точки зрения разработки безопасного программного обеспечения наиболее важными являются

-4 Memcheck — инструмент для поиска ошибок при использовании динамически выделяемой памяти;

-5 Helgrind и DRD — инструмент для отслеживания состояния гонки и других ошибок в многопоточном коде;

-6 SGCheck — экспериментальный инструмент для поиска ошибок использования памяти на стеке и в глобальной области.

Рассмотрим возможности модулей Memcheck и SGCheck. Этот модуль может выявлять следующие проблемы:

-7 чтение или запись по неправильным адресам памяти -- за границами выделенных блоков памяти;

-8 использование не инициализированных значений, в том числе и для переменных выделяемых на стеке;

-9 ошибки освобождения памяти, например, когда блок памяти уже был освобожден в другом месте;

-10 использование "неправильной" функции освобождения памяти, например использование delete для памяти, выделенной с помощью new [];

-11 передача некорректных параметров системным вызовам, например указание неправильных указателей для операций чтения из буфера, указанного пользователем;

-12 пересечение границ блоков памяти при использовании операций копирования/перемещения данных между двумя блоками памяти.

Для этих ошибок данные выдаются по мере их обнаружения, и обычно они выглядят следующим образом:

```
Mismatched free() / delete / delete []
   at 0x40043249: free (vg_clientfuncs.c:171)
   by 0x4102BB4E: QGArray::~QGArray(void) (tools/qgarray.cpp:149)
   by 0x4C261C41: PptDoc::~PptDoc(void) (include/qmemarray.h:60)
   by 0x4C261F0E: PptXml::~PptXml(void) (pptxml.cc:44)
 Address 0x4BB292A8 is 0 bytes inside a block of size 64 alloc'd
   at 0x4004318C: operator new[](unsigned int) (vg_clientfuncs.c:152)
   by 0x4C21BC15: KLaola::readSBStream(int) const (klaola.cc:314)
   by 0x4C21C155: KLaola::stream(KLaola::OLENode const *) (klaola.cc:416)
   by 0x4C21788F: OLEFilter::convert(QCString const &) (olefilter.cc:272)
```

В первой строке приводится описание соответствующей ошибки, а затем идет стек вызова функций, приведших к появлению данной ошибки. В том случае, где это необходимо (как в нашем примере), выдается также адрес блока памяти и место где этот блок памяти был выделен.

При окончании работы программы Valgrind выдает сводную таблицу, описывающую количество найденных ошибок, а также выделение памяти в программе, например:

```
ERROR SUMMARY: 2569904 errors from 493 contexts (suppressed: 17962 from 9)
malloc/free: in use at exit: 85,066,939 bytes in 313,004 blocks.
malloc/free: 10,552,914 allocs, 10,239,910 frees, 565,747,810 bytes allocated.
For counts of detected errors, rerun with: -v
searching for pointers to 313,004 not-freed blocks.
checked 117,623,772 bytes.
```

И в самом конце отчета, выдается сводная таблица по каждому из типов ошибок работы с памятью:

```
LEAK SUMMARY:
   definitely lost: 2,260 bytes in 47 blocks.
   indirectly lost: 1,680 bytes in 66 blocks.
     possibly lost: 2,703,124 bytes in 13,791 blocks.
   still reachable: 82,359,875 bytes in 299,100 blocks.
        suppressed: 0 bytes in 0 blocks.
```

Definitely lost означает, что valgrind нашел область памяти, на которую нет указателей, т.е. программист не освободил память, при выходе указателя за область видимости. Possibly lost показывает, что найден указатель, указывающий на часть области памяти, но Valgrind не уверен в том, что указатель на начало области памяти до сих пор существует (это может происходить в тех случаях, когда программист вручную управляет указателями). Still reachable обычно означает, что Valgrind нашел указатель на начало не освобожденного блока памяти, что во многих случаях связано с выделением глобальных переменных и т.п. вещей. Обычно эта информация показывается только при указании опции --show-reachable со значением yes.

Между двумя этими таблицами выдаются данные по каждой из найденных ошибок работы с памятью, вида:

```
756 bytes in 27 blocks are definitely lost in loss record 1,077 of 1,267
   at 0x4022AB8: malloc (vg_replace_malloc.c:207)
   by 0x7C485DA: __libc_res_nsend (res_send.c:425)
   by 0x7C47276: __libc_res_nquery (res_query.c:171)
   by 0x7C47B5B: __res_nquery (res_query.c:223)
   by 0x834A618: LookUp::LookUp(std::string&) (LookUp.cpp:83)
   by 0x8637C29: ClientThread::Connect(LogFileEntry&) (ClientThread.cpp:359)
   by 0x86394D5: ClientThread::Run() (ClientThread.cpp:215)
   by 0x80FD839: Thread::StartRunLoop(void*) (Thread.cpp:315)
   by 0x4163FD9: start_thread (pthread_create.c:297)
   by 0x43843AD: clone (in /usr/lib/debug/libc-2.7.so)
```

Первой строкой идет описание ошибки, вместе с указанием номера блока в списке потенциально потерянных блоков памяти, а также размером "потерянного" блока памяти. "Важность" ошибки соответствует описанию в итоговой таблице. После строки описания, приводится стек вызовов функций, которые привели к возникновению "потерянного" блока памяти.

Этот список достаточно подробен для того, чтобы обнаружить точное место возникновения данной утечки памяти.

Проверка утечек памяти в модуле Memcheck по умолчанию отключена. Для ее включения необходимо использовать опцию --leak-check=yes

Использование санитайзеров компилятора GCC для анализа ошибок работы с памятью

Компиляторы проекта GCC, например gcc и g++, имеют встроенные динамического анализа, называемые санитайзерами. Санитайзеры — это система инструментации кода, позволяющая выявлять дефекты программ, связанные с ошибками памяти и гонками. При компиляции с опцией санитайзера компиляторы добавляют в машинный код программы дополнительные инструкции так, что программа работает как обычно и выполняет свою работу, но при этом в процессе выполнения осуществляет проверки памяти.

Чтобы воспользоваться санитайзерами, надо компилировать и компоновать с флагом -fsanitize=[название]. Санитайзеров несколько, каждый ориентирован на свой тип ошибок. Компилятор GCC для выявления ошибок памяти поддерживает санитайзер address, а для выявления только утечек памяти — санитейзер leak. Следует отметить, что санитйзер address также определяет утечки памяти, но при наличии других ошибок программа прерывается и информация по утечкам не выводится. Таким образом, при использовании санитайзера address получить информацию по утечкам памяти можно только если других ошибок в программе нет. Санитайзер leak дает информацию только по утечкам и не проверяет других проблем с памятью. Поэтому его можно использовать для обнаружения утечек памяти даже при наличии других ошибок.

Санитайзер undefined используется для поиска ситуаций неопределенного поведения. Например деления на 0, арифметического переполнения,

При компиляции с санитайзером желательно использовать флаги **-O0** или **-Og** для отключения оптимизации. А также флаг **-g** для включения в откомпилированную программу информации о номерах строк исходного текста. Пример компиляции программы с санитайзером address:

```
g++ myprog.cpp -g -Og -fsanitize=address -o myprog
```

Запускается программа, скомпилированная с санитайзерами, как обычно, никаких специальных флагов не надо. Работает несколько медленнее обычной прошраммы. Если при многократных запусках изменяются адреса памяти, в которых происходят ошибки, то тогда надо выключить ASLR (рандомизация размещения адресного пространства программ). Для этого программу надо запускать с помощью утилиты setarch:

```
setarch -R ./myprog
```

где ./myprog — имя запускаемой программы.

Использование санитайзеров компилятора clang для анализа ошибок работы с памятью

Компиляторы clang и clag++ по параметрам аналогичны gcc и g++, однако сам функционал санитайзеров там выполнен неколько иначе. Поэтому результаты работы санитайзеров могут отличаться. Кроме того, в clang есть санитайзер memory для обнаружения использования неинициализированных переменных.

Задание к лабораторной работе

1. Клонировать репозиторий https://gitlab.com/winwood/rbpo.git
2. Выполнить сборку проекта *longnumbers* в отладочной конфигурации.
3. Выполнить динамический анализ полученной программы в помощью Valgrind для поиска проблем использования памяти.
4. Выполнить сборку проекта, используя санитайзер address компиляторов gcc и clang, и выполнить поиск проблем
5. Выполнить сборку проекта, используя санитайзер leak компиляторов gcc и clang, и выполнить поиск проблем
6. Выполнить сборку проекта, используя санитайзер undefined компиляторов gcc и clang, и выполнить поиск проблем
7. Проанализировать полученные отчеты об ошибках, сравнить результаты трех вариантов динамического анализа.
8. Внести изменения в программу для устранения дефектов в коде.
9. При необходимости повторить п. 2-6 до до полного отсутствия ошибок в отчетах анализа
10. Взять файл for_dynamic_testing.cpp из ЭИОС и выполнить анализ с помощью valgrind, g++ и clang++ следующих функций:
    - divide;
    - multiply;
    - fill_ones;
    - fill_rand и print_rand (совместно).
11. Для функции divide сделать тест, обнаруживающий переполнение буфера
12. Для функции multiply сделать тест, обнаруживающий арифметическое переполнение
13. Для функции fill_ones сделать тест, обнаруживающий использование указателя после выхода из функции
14. Для функций fill_rand и print_rand сделать тест, обнаруживающий утечку памяти
15. Сделать выводы по результатам п. 10-14