МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ

РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение высшего образования «Самарский национальный исследовательский университет имени академика С.П. Королева»

(Самарский университет)

Институт информатики, математики и электроники

Факультет информатики

Кафедра геоинформатики и информационной безопасности

КУРСОВОЙ ПРОЕКТ ПО ДИСЦИПЛИНЕ

«Инженерия программного обеспечения»

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

«Статический анализатор кода программ, написанных на С++, на уязвимости по информационной безопасности»

ЧАСТЬ 2

Разработка общего алгоритма. Разработка обработчика внедрения команд, обработчика внедрения SQL команд, обработчика проблем, связанных с динамическим выделением памяти, модуля тестирования и генерации кода.

|  |  |
| --- | --- |
| Студент \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  *(подпись)* | Е.А. Терентьев |
|  |  |
| Руководитель работы \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  *(подпись)* | Я.А. Мостовой |
|  |  |

Представлено к защите: 6 июня 2020 г

САМАРА 2020

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ

РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение высшего образования «Самарский национальный исследовательский университет имени академика С.П. Королева»

(Самарский университет)

Институт информатики, математики и электроники

Факультет информатики

Кафедра геоинформатики и информационной безопасности

**ЗАДАНИЕ НА КУРСОВОЙ ПРОЕКТ**

Студенту ***Терентьеву Е.А.*** группы 6411-100503D

Тема проекта: ***«Статический анализатор кода программ, написанных на С++, на уязвимости по информационной безопасности», часть № 2. Разработка общего алгоритма. Разработка обработчика внедрения команд, обработчика внедрения SQL команд, обработчика проблем, связанных с динамическим выделением памяти, модуля тестирования и генерации кода.***

|  |  |  |
| --- | --- | --- |
| Планируемые результаты освоения образовательной программы (компетенции) | Планируемые результаты | Содержание задания |
| ПК-5 | **Знать:**  этапы оценки и управления рисками, современные методики оценки рисков, смысловое содержание численных показателей оценок рисков.  **Уметь:**  определять необходимую и достаточную совокупность средств защиты информации, организационных мер направленных на снижение рисков ИБ, разработать архитектуру СИБ, направленную на снижение рисков информационной безопасности автоматизированных систем.  **Владеть:**  навыками определения приемлемого для организации уровня риска; идентификации, анализа и оценки рисков; ранжирование рисков; принятия решения по рискам и разработка плана реагирования на риски; реализации мероприятий и оценки эффективности реализованных мер. | Разработать статический анализатор кода, направленный на снижение рисков информационной безопасности систем, использующих ПО |
| ПК-12 | **Знать:**  критерии и методы оценки качества политики безопасности АС; основные методы управления информационной безопасностью; средства поддержки процессов управления информационной безопасностью АС.  **Уметь:**  проводить аудит информационной безопасности АС; применять методы синтеза системы защиты информации в АС; формировать политику информационной безопасности АС; вести мониторинг информационной безопасности АС.  **Владеть:**  навыками определения класса АС; организации обеспечения информационной безопасности АС; применения средств поддержки процессов управления информационной безопасностью АС. | Разработать политику безопасности, средства мониторинга и поддержки процессов информационной безопасности программного обеспечения при использовании статического анализатора кода |
| ПСК-7.5 | **Знать:**  задачи, решаемые подразделениями по защите информации, виды угроз безопасности информации.  **Уметь:**  координировать деятельность подразделений и специалистов по защите информации в организациях, в том числе на предприятии и в учреждении.  **Владеть:** навыками формализации постановки задач по защите информации, контроля хода выполнения задач. | Формализовать постановку задач и координировать деятельность специалистов -разработчиков статического анализатора по обеспечению защиты программного обеспечения от уязвимостей. |

Описание задач проекта, необходимые исходные данные, требования к объему материалов, представлению результатов, программному обеспечению, оформлению отчета приведены в «методических указаниях к курсовому проекту по ИПО»

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Дата выдачи задания | 6 февраля 2020 г. | | | |
| Срок представления на кафедру пояснительной записки | | | 6 июня 2020 г. | |
| Руководитель курсового проекта профессор кафедры ГИиИБ, д.т.н. | |  | | Я.А. Мостовой |
|  | | *(подпись)* | |  |
| Задание принял к исполнению  студент группы № 6411-100503D | |  | | Е.А. Терентьев |
|  | | *(подпись)* | |  |

РЕФЕРАТ

**Пояснительная записка к курсовому проекту:** 50 c., 30 рисунков, 4 таблицы, 10 источников, 2 приложения.

СТАТИЧЕСКИЙ АНАЛИТОР, УЯЗВИМОСТЬ, ИНФОРМАЦИОННАЯ БЕЗОПАСНОСТЬ, АНАЛИЗ КОДА, ОТЛАДОЧНЫЕ ТЕСТЫ.

Цель работы – разработка статического анализатора кода программ, написанных на C++, для анализа уязвимостей информационной безопасности, а также разработка средств отладки разрабатываемого анализатора и средств, подтверждающих его эффективность таким образом, чтобы все специально внесенные виды уязвимостей во фрагменты ПО в различных сочетаниях могли бы быть обнаружены анализатором.

В итоге были получены следующие результаты:

1. Выполнен анализ существующих статических анализаторов кода C++;
2. Выполнен обзор существующих методов и принципов, на которых основывается статический анализ кода;
3. Разработан и протестирован собственный статический анализатор кода C++, написанный на языке Python в среде разработке PyCharm.

Содержание

[РЕФЕРАТ 5](#_Toc42361679)

[Содержание 6](#_Toc42361680)

[ВВЕДЕНИЕ 7](#_Toc42361681)

[1 АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ 9](#_Toc42361682)

[1.1 Теоретические основы 9](#_Toc42361683)

[1.2 Анализ существующих реализаций 10](#_Toc42361684)

[1.3 Выводы и результаты 13](#_Toc42361685)

[2 ОПИСАНИЕ РАЗРАБОТАННОГО ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ 14](#_Toc42361686)

[2.1 Выбор средств разработки и программных средств 14](#_Toc42361687)

[2.2 Требования к разрабатываемой программе 14](#_Toc42361688)

[2.3 Структура программы 16](#_Toc42361689)

[2.4 Описание модулей и их составных частей 17](#_Toc42361690)

[2.4.1 Модуль первичного лексического анализа 17](#_Toc42361691)

[2.4.2 Модуль обработчиков уязвимостей 18](#_Toc42361692)

[2.4.3 Модуль пользовательского интерфейса 28](#_Toc42361693)

[2.4.4 Агрегирующий модуль 33](#_Toc42361694)

[2.4.5 Модуль тестирования и генерации тестового кода С++ 33](#_Toc42361695)

[2.5 Вывод и результаты 38](#_Toc42361696)

[3 ТЕСТИРОВАНИЕ 39](#_Toc42361697)

[4 ИНДИВИДУАЛЬНАЯ ЧАСТЬ 48](#_Toc42361698)

[1 Разработка общего алгоритма работы программы 48](#_Toc42361699)

[2 Разработка программы 49](#_Toc42361700)

[3 Разработка тестов 56](#_Toc42361701)

[ЗАКЛЮЧЕНИЕ 70](#_Toc42361702)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 71](#_Toc42361703)

[ПРИЛОЖЕНИЕ А 72](#_Toc42361704)

[ПРИЛОЖЕНИЕ Б 77](#_Toc42361705)

ВВЕДЕНИЕ

Одним из общепринятых и эффективных методов выявления дефектов является обзор кода. Его суть в совместном внимательном чтении исходного кода с целью обнаружения ошибок и дефектов. В этом процессе выявляются ошибки или участки кода, которые могут стать ошибочными в будущем. Обзор кода является эффективным, но очень трудозатратным методом, так как отнимает время от основной разработки у программистов, а также требует от них большой концентрации. Альтернативным решением являются инструменты статического анализа кода. Они обрабатывают исходные тексты программ и выдают программисту рекомендации обратить повышенное внимание на определенные участки кода. Несмотря на то, что существует большое количество статических анализаторов кода, зачастую в них опускается возможность обнаружения проблем, связанных с уязвимостями информационной безопасности. Это очень весомое опущение, так как состояния информационной безопасности разрабатываемой программы является одним из главных столпов, на котором стоит архитектура приложения. В связи с этим разработка статического анализатора, учитывающего возможности обнаружения уязвимостей информационной безопасности является важной и актуальной задачей.

Цель работы – разработка статического анализатора кода программ, написанных на C++, для анализа уязвимостей информационной безопасности, а также разработка средств отладки разрабатываемого анализатора и средств, подтверждающих его эффективность таким образом, чтобы все специально внесенные виды уязвимостей во фрагменты ПО в различных сочетаниях могли бы быть обнаружены анализатором.

В соответствии с поставленной целью были сформулированы и решены следующие задачи:

1. анализ предметной области;
2. разработка анализатора программ, написанных на С++, на уязвимости информационной безопасности;
3. программирование разработанных алгоритмов;
4. разработка проверочных тестов;
5. разработка генератора случайного тестового кода на С++ для отладки и тестирования разработанного статического анализатора;
6. автономная отладка программ анализатора;
7. комплексная отладка программ анализатора.

# АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ

## Теоретические основы

Принцип работы статических анализаторов основывается на разборе предоставленного к анализу кода без его выполнения.

В ходе исследования предметной области, мы выяснили, что существует два основных подхода при реализации статического анализатора.

Первый подход – использование регулярных выражений. В этом случае анализ кода ведется с помощью базы, в которой находятся шаблоны уязвимостей. Шаблоны представляют из себя регулярные выражения, составленные по сигнатуре уязвимости. Такой подход удобен в реализации и имеет право на существование, но его возможности ограничены. Так, например, с его помощью практически невозможно осуществлять комплексный анализ, то есть анализ уязвимостей, которые распределены по коду или пересекаются с другими уязвимостями.

Второй подход решает проблему комплексного анализа. В этом случае анализ начинается с разбиения кода анализируемой программы на лексемы – составные части, такие как функции, переменные и т.д. Далее из этих лексем строится объект, характеризующий анализируемую программу. После этого начинается анализ полученного объекта на наличие уязвимостей с помощью реализованных для этого алгоритмов. Данный вид анализа дает очень подробное представление о структуре анализируемой программы, но в нашем случае использование исключительно этого подхода является избыточным, так как он затрагивает более широкий класс решаемых задач помимо поставленных перед нашим анализатором, а, следовательно, ведет к нерациональному потреблению ресурсов.

После изучения двух подходов было решено использовать гибридный подход, то есть использование комплексного анализа с разбиением совместно с регулярными выражениями. Такой подход позволяет рационально использовать достоинства комплексного анализа в сочетании с механизмом регулярных выражений, обеспечивающего легкую масштабируемость анализатора, то есть возможность добавления новых шаблонов уязвимостей.

## Анализ существующих реализаций

В рамках анализа предметной области был произведен обзор существующих решений статического анализа. При исследовании конкретной реализации статического анализатора мы изучали его принцип работы, функциональные возможности, а также возможность добавления собственного функционала, позволяющего осуществлять поиск необходимых нам уязвимостей. Мы выделили три наиболее распространенных анализатора.

**CppСheck** — статический анализатор кода для языка C/C++, предназначенный для поиска ошибок, которые не обнаруживаются компиляторами. Главной целью проекта является сведение до минимума количества ложных срабатываний при поиске ошибок. Анализатор способен проверять нестандартные участки кода, включающие использование расширений компилятора, встраиваемый ассемблер и т. п.

Перечень некоторых обрабатываемых уязвимостей:

1. утечки памяти;
2. утечки ресурсов;
3. выход за границы массивов;
4. использование неинициализированных переменных;
5. использование устаревших функций;
6. разыменование нулевого указателя.

**PVS-Studio** — инструмент для выявления ошибок и потенциальных уязвимостей в исходном коде программ, написанных на языках С, C++, C# и Java. Работает в 64-битных системах на Windows, Linux и macOS и может анализировать код, предназначенный для 32-битных, 64-битных и встраиваемых ARM платформ. PVS-Studio выполняет статический анализ кода и генерирует отчёт, помогающий программисту находить и устранять ошибки. PVS-Studio выполняет широкий спектр проверок кода, но наиболее силён в поисках опечаток и последствий неудачного переиспользования кода методом «копировать-вставить».

Перечень некоторых обрабатываемых уязвимостей:

1. утечки памяти;
2. утечки ресурсов;
3. выход за границы массивов;
4. использование устаревших функций;
5. проверка операций ввода/вывода;
6. разыменование нулевого указателя.

**Clang Static Analyzer** - инструмент анализа исходного кода, который находит ошибки в программах на C, C ++ и Objective-C. Анализатор является проектом с полностью открытым исходным кодом и является частью проекта Clang. Как и остальная часть Clang, анализатор реализован в виде библиотеки C ++, которая может использоваться другими инструментами и приложениями.

Перечень некоторых обрабатываемых уязвимостей:

1. утечки памяти;
2. поиск неиспользуемых переменных;
3. проверка возвращаемого значения;
4. выход за границы массивов;
5. использование устаревших функций;
6. проверка операций ввода/вывода;
7. разыменование нулевого указателя.

Приведенные выше статические анализаторы могли бы быть использованы в качестве базиса, модернизация которого могла бы решить поставленные в курсовом проекте задачи. В силу причин, описанных в таблице 1, мы отказались от варианта с переиспользованием рассмотренных анализаторов.

Таблица 1 – Обоснования отказа от имеющихся анализаторов

|  |  |  |
| --- | --- | --- |
| Название анализатора | Расположение | Причина отказа от анализатора |
| СppCheck | http://cppcheck.source  forge.net/ | Отсутствие анализа уязвимостей многопоточности.  Цельная структура, анализатора, не позволяющая полноценно интегрировать свои версии шаблонов уязвимостей. |
| PVS-Studio | https://www.viva64.com  /ru/pvs-studio/ | Отсутствие анализа уязвимостей многопоточности.  Исходный код не является открытым, невозможно вносить свои изменения. |
| Clang Static  Analyzer | https://clang-analyzer.llvm.org/ | Отсутствует анализа уязвимостей многопоточности, уязвимостей, связанных с выделением памяти. |

## Выводы и результаты

В ходе проведенного анализа предметной области были изучены и выбраны подходы к реализации статического анализатора. Принят гибридный подход к разработке статического анализатора. Были исследованы существующие решения в области статического анализа, а также была рассмотрена возможность их переиспользования для добавления собственного функционала, позволяющего осуществлять поиск необходимых уязвимостей. В итоге было принято решение о создание собственного статического анализатора, который позволит решить задачи, поставленные в курсовом проекте.

# ОПИСАНИЕ РАЗРАБОТАННОГО ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

## Выбор средств разработки и программных средств

Для разработки нашего приложения был выбран язык Python3. Выбор был обусловлен двумя основными причинами:

1. наличие гибкой и функциональной библиотеки для работы с регулярными выражениями - Lib/re.py, а также расположенность языка для работы с ними;
2. наличие библиотеки для создания удобного и дружественного интерфейса - Lib/tkinter;

В качестве среды разработки выступила программа PyCharm, специально предназначенная для создания приложений на языке Python. Она имеет удобный интерфейс и большое количество полезных функций, позволяющих проводить отладку кода, связываться с удаленным репозиторием для синхронизации изменений в разрабатываемом приложение и т.д.

## Требования к разрабатываемой программе

Разрабатываемое приложение должно соответствовать ряду требований, которые были описаны в методическом указании к курсовому проекту, а также сформулированы в ходе предреализационного планирования.

1. Программа должна иметь удобный и понятный графический интерфейс.
2. Интерфейс программы должен содержать:
3. список загруженных на проверку программ;
4. список загруженных поисковых программ;
5. список загруженных тест примеров;
6. указатель на исследуемые уязвимости с изменением его состояния по мере проверки уязвимостей;
7. кнопку «произвести проверку» программы №… или последовательно все;
8. кнопку произвести проверку на тест- примерах №… или на всех.
9. выводимые результаты работы программы для каждой проверяемой программы или тест примера должны содержать виды уязвимостей с отметкой обнаружена или не обнаружена. В случае обнаружения указывается точка обнаружения.
10. Должен быть реализован поиск следующих уязвимостей:
11. переполнение буфера;
12. пренебрежение обработок ошибок;
13. проблемы, связанные с динамическим выделением памяти;
14. внедрение команд;
15. внедрение SQL команд;
16. состояние гонки;
17. ошибка форматной строки;
18. некорректный доступ к файлам;
19. читатели – писатели;
20. случайные числа криптографического характера;
21. переполнение целых чисел;
22. пренебрежение безопасным хранением данных;
23. утечка информации.
24. Программа должна быть проверена достаточным количеством ручных тестов, а также тестами, созданными автогенератором тестов.

## Структура программы

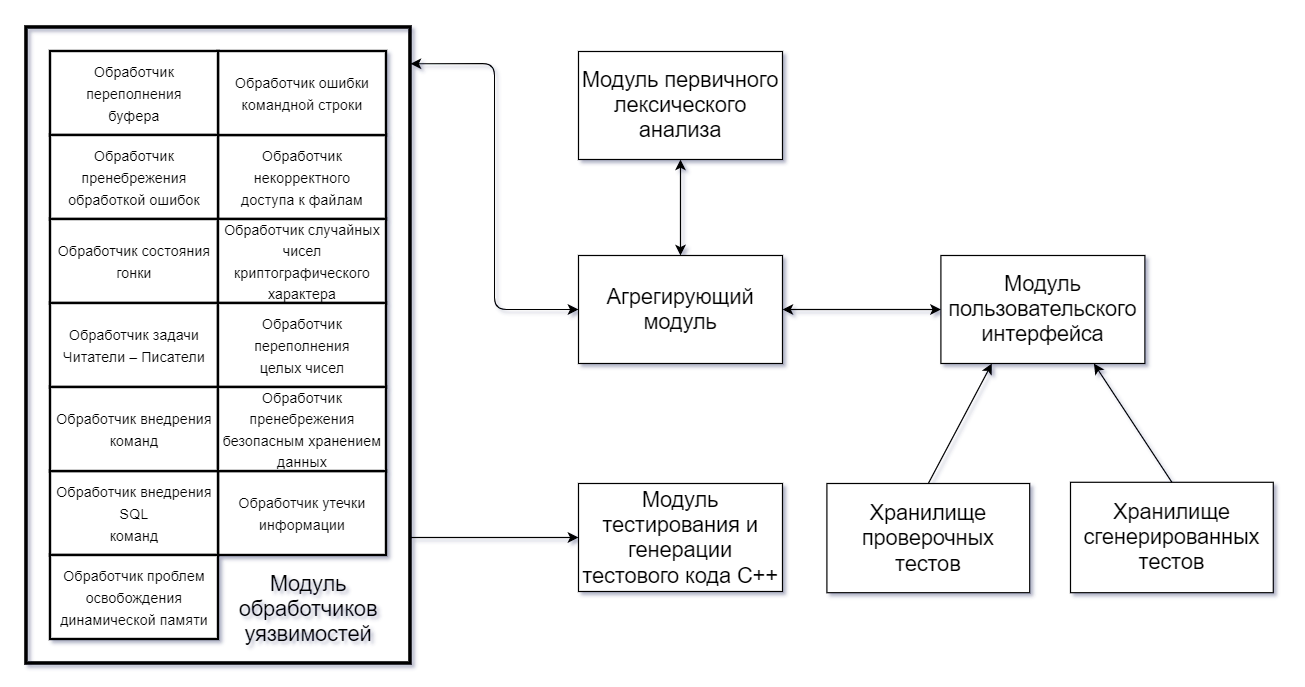
Разработанную программу можно разделить на пять основных модулей (рисунок 1).

1. Модуль первичного лексического анализа.
2. Модуль обработчиков уязвимостей, содержащий в себе 13 частей:

* обработчик переполнения буфера;
* обработчик пренебрежения обработкой ошибок;
* обработчик состояния гонки;
* обработчик задачи читатели-писатели;
* обработчик внедрения команд;
* обработчик внедрения SQL команд;
* обработчик проблем освобождения динамической памяти;
* обработчик ошибки командной строки;
* обработчик некорректного доступа к файлам;
* обработчик случайных чисел криптографического характера;
* обработчик переполнения целых чисел;
* обработчик пренебрежения безопасным хранением данных;
* обработчик утечки информации.

1. Модуль пользовательского интерфейса.
2. Агрегирующий модуль.
3. Модуль тестирования и генерации тестового кода C++.

Библиотекой уязвимостей является модуль обработчиков уязвимостей. При необходимости ее можно дополнить новыми обработчиками, добавив собственный класс обработчика, который будет подчиняться общим правилам создания обработчиков. На структурной схеме (рисунок 1) также можно увидеть два объекта: «Хранилище проверочных тестов» и «Хранилище сгенерированных тестов», это директории, в которых хранятся соответственно проверочные и сгенерированные тесты.

Рисунок 1 – Структурная схема ПО

## Описание модулей и их составных частей

### Модуль первичного лексического анализа

Как мы отметили раннее нами был выбран гибридный подход к статическому анализу кода. Модуль первичного лексического анализа обеспечивает начальный анализ кода, который позволяет получить объектную модель исследуемой программы. В эту модель входят все объявленные в анализируемом коде функции, их параметры, переменные и потоки. При анализе очень важно иметь полное представление о структуре анализируемой программы, так как, например, в разных функциях могут быть объявлены переменные или потоки с одинаковым именем, и это необходимо учитывать.

Для представления модели используются классы:

1. Variable – данный класс отвечает за хранение информации о найденной переменной. Он содержит полную декларацию переменной, номер строки, на которой она была объявлена, ее имя, тип и значение.
2. FunctionContext – данный класс отвечает за хранение информации о найденной функции. Класс содержит тип возвращаемого значения функции, ее имя, список ее потоков, параметров и переменных, которые являются объектами класса Variable, а также список всех строк, которые включает в себя функция.
3. Thread - данный класс отвечает за хранение информации о найденном потоке из стандартной библиотеки С++. Он содержит полную декларацию потока, номер строки, на которой он была объявлен, его имя, имя исполняемой функции и список переданных параметров.

После завершения первичного лексического анализа сформированная объектная модель анализируемой программы поступает к обработчикам уязвимостей. Полученная ими информация расширяет возможности анализа и делает его проще.

### Модуль обработчиков уязвимостей

Модуль обработчиков уязвимостей состоит из 13 классов, отвечающих за поиск и обработку конкретных уязвимостей.

Таблица 2 – Соответствие уязвимости, ее обработчика и краткого описания метода обнаружения

|  |  |  |
| --- | --- | --- |
| Уязвимость | Метод обнаружения | Разработанное ПО обнаружения |
| Переполнение буфера | Поиск объявленных буферов и функций, в которых они были использованы. | BufferOverflowHandler |
| Переполнение целых чисел | Поиск арифметических выражений, в которых используются объявленные целочисленные переменные и проверка этих выражений на переполнение. | IntegerOverflowHandler |
| Внедрение SQL-кода | Поиск уязвимых функций вызова SQL запросов с телом запроса, в котором отсутствует экранирование символа единичной кавычки. | SqlInjectionHandler |
| Внедрение команд | Поиск уязвимых функции, которые позволяют выполнить внедрение команд. | CommandsIntoductionHandler |
| Пренебрежение безопасным хранением данных | Поиск уязвимых функций, которые позволяют изменять уровень доступа к данным и задавать привилегии для определенных объектов  ОС. | DataStorageManagmentHandler |
| Утечка информации | Поиск уязвимых функций, которые могут привести к потере или утечке данных. | DataLeakHandler |
| Некорректный доступ к файлам | Поиск уязвимых функций, позволяющие проводить операций над файлами (создание ссылок, файлов, директорий, изменение файлов и т.д. | IncorrectFileAccessHandler |
| Случайные числа криптографического характера. | Поиск функций генерации случайных чисел, инициализированных константным значениями, а также других не криптостойких генераторов из стандартных библиотек С++. | RandomGeneratorHandler |
| Ошибка форматной строки | Поиск функции функций, использующих форматную строку. | FormatStringHandler |
| Состояние гонки | Поиск потоков, у которых используется одна и та же исполняемая функция. | RaceConditionHandler |
| Читатели-писатели | Поиск потоков, использующих одинаковые переменные в качестве параметров исполняемых функций, а также анализ этих исполняемых функций на наличие и структуру блоков синхронизации. | ReadersWritersHandler |
| Проблемы с динамическим выделением памяти | Поиск переменных, для которых была выделена, но не освобождена память. | MemoryLeakHandler |
| Пренебрежение обработкой ошибок | Поиск «try-catch» выражений с отсутствием обработки исключений. | EmptyCatchHandler |

Рассмотрим каждую уязвимость и ее обработчик в отдельности. Заранее отметим, что благодаря выбранному гибридному подходу с составлением модели анализируемой программы, пересечение и перекрытие уязвимостей между собой или между произвольным кодом не влияет на их обнаружение.

1. *Переполнения буфера*

Данная уязвимость возникает, например, при получении данных извне и дальнейшем копировании их во внутренний буфер без проверки размера копируемых данных. При необходимом объеме записываемых данных адрес возврата, находящийся в стеке, может быть перезаписан. При этом на его место можно перезаписать необходимый адрес, который указывает на вредоносный код.

Обработчик «BufferOverflowHandler» руководствуется следующим шаблоном при поиске уязвимости:

1. в коде ведется поиск всех объявленных переменных, которые являются буферами, то есть поиск массивов и указателей;
2. поиск функции, предназначенных для работы с буферами;
3. проверка того, используются ли найденные раннее переменные в обнаруженных потенциально опасных функциях.

Между объявлением переменных и небезопасной функцией, которая их использует может быть любое количество произвольного кода или других уязвимостей.

1. Пренебрежение обработкой ошибок

Безопасность подвергается серьезной угрозе, когда не происходит должного уделения внимания обработке ошибок. Опасность заключается в том, что в некоторых ситуациях программа может оказаться в некорректном состоянии, что ведет к отказу обслуживания, а, следовательно, создает угрозу безопасности.

Обработчик «EmptyCatchHandler» руководствуется следующим шаблоном при поиске уязвимости:

1. ведется поиск «try-catch» конструкции;
2. осуществляется проверка, является ли тело блока «catch» пустым, при этом обработчик учитывает возможность вложенных конструкций «try-catch» (рисунок 2), в этом случае будет осуществлен рекурсивный анализ таких конструкций.

Тело блока «catch» помимо произвольного кода или других уязвимостей может содержать в себе вложенную конструкцию «try-catch», а та в свою очередь еще одну вложенную конструкцию и так далее рекурсивно. Обработчик способен отслеживать пустые блоки «catch», находящиеся на любом конечном уровне рекурсивных вложений блоков «try-catch».

try {

#код программы

}

catch {

try {

#код программы}

catch {} #пустой блок «catch»

}

Рисунок 2 – вложенная конструкция «try-catch»

1. Переполнение целых чисел

Уязвимость в компьютерной арифметике, при которой вычисленное в результате операции значение не может быть помещено в n-битный целочисленный тип данных. Различают переполнение через верхнюю границу представления и через нижнюю.

Обработчик IntegerOverflowHandler руководствуется следующим шаблоном при поиске уязвимости:

1. поиск всех инициализированных, целочисленных переменных;
2. поиск всех арифметических выражений с найденными раннее переменными;
3. проверка результата арифметического выражения на возможность переполнения.

Между объявлением инициализированных переменных и арифметическим выражением, которое их использует может быть любое количество произвольного кода или других уязвимостей.

1. Проблемы с динамическим выделением памяти

Данная уязвимость возникает при работе с динамическим выделением памяти. Например, программа завершается без освобождения памяти.

Обработчик MemoryLeakHandler руководствуется следующим шаблоном при поиске уязвимости:

1. поиск всех переменных, для которых была выделена память;
2. поиск операторов высвобождения памяти, по всем найденным переменным;
3. если высвобождение памяти не было обнаружена, появляется предупреждение о возможной ошибке.

Между объявлением переменных, для которых была выделена память и потенциальным оператором высвобождения памяти может быть любое количество произвольного кода или других уязвимостей.

1. Внедрение SQL команд

Эта уязвимость предполагает, что любая строка, значение которой не может быть определена во время компиляции, может содержать данные, вводимые пользователем. Созданная из введенных пользователем данных командная строка SQL уязвима перед атаками путем внедрения кода SQL. При атаке путем внедрения кода SQL нарушитель предоставляет входные данные, которые изменяют структуру запроса с целью повредить или получить несанкционированный доступ к основной базе данных. Типичные методы включают в себя введение одинарной кавычки или апострофа, который является разделителем строкового литерала SQL; два тире, обозначающие комментарий SQL; и точка с запятой, которая указывает, что следует новая команда.

Обработчик SqlInjectionHandler руководствуется следующим шаблоном при поиске уязвимости:

1. поиск функций в которых выполняется sql – запрос;
2. проверка тела sql – запроса, переданного функции на наличие неэкранированной одинарной кавычки.

Данная уязвимость не является составной, поэтому ее не может пересекать произвольный код или другие уязвимости.

1. Внедрение команд

Если в отношении пользовательского ввода не осуществляется должная проверка, то может быть осуществлена атака путем внедрения команды. Используя эту уязвимость, атакующий может формировать ввод таким образом, что он будет содержать команды операционной системы, которые будут выполняться с привилегиями уязвимого приложения.

Обработчик CommandsIntroductionHandler руководствуется следующим шаблоном при поиске уязвимости:

1. поиск функций, позволяющие выполнять исполняемые файлы и системные команды, а также создавать дочерние процессы, которым можно передать команду на выполнение.

Данная уязвимость не является составной, поэтому ее не может пересекать произвольный код или другие уязвимости.

1. Состояние гонки

Уязвимость, при которой работа системы или приложения зависит от того, в каком порядке выполняются части кода. Состояние гонки возникает тогда, когда несколько потоков многопоточного приложения пытаются одновременно получить доступ к одним и тем же ресурсам, причем хотя бы один поток выполняет запись. Кроме того, ошибки такого рода невероятно сложно воспроизвести повторно.

Обработчик RaceConditionHandler работает с любым количеством потоков и руководствуется следующим шаблоном при поиске уязвимости:

1. получение всех объявленных потоков;
2. анализ найденных потоков на наличие одинаковых исполняемых функций.

Между объявленными потоками и исполняемыми функциями может быть любое количество произвольного кода или других уязвимостей.

1. Ошибка форматной строки

Данная уязвимость заключается в использовании одной из функций форматного вывода (printf, fprintf, sprintf). Эти функции имеют неопределенное количество параметров, число и интерпретация которых определяется форматной строкой. Это позволяет злоумышленнику задать свою собственную форматную строку, что ведет к возможности чтения значений из стека, а при указании специального формата, и к записи в память.

Обработчик FormatStringHandler руководствуется следующим шаблоном при поиске уязвимости:

1. поиск функций, использующие форматную строку в качестве параметра.

Данная уязвимость не является составной, поэтому ее не может пересекать произвольный код или другие уязвимости.

1. Некорректный доступ к файлам

Можно выделить типичные проблемы безопасности, связанные с файловыми операциями. Первый тип - является ли файл файлом вообще и стоит ли с ним работать. Второй вид уязвимости состоит в вопросе существования файла, с которым будет вестись работа.

Обработчик IncorrectFileAccess руководствуется следующим шаблоном при поиске уязвимости:

1. поиск объявленных файловых переменных;
2. поиск функций, отвечающий за создание директорий, файлов.

Данная уязвимость не является составной, поэтому ее не может пересекать произвольный код или другие уязвимости.

1. Задача читатели писатели

Данная уязвимость возникает при одновременном доступе нескольких потоков к одной и той же области памяти, допускающей чтение и запись. При этом одновременно могут читать сколько угодно потоков, но писать — только один. При одновременном чтении проблем не возникает, но изменение памяти по прерыванию во время ее чтения некоторым потоком может привести к серьезным ошибкам.

Обработчик ReadersWritersHandler работает с любым количеством потоков и руководствуется следующим шаблоном при поиске уязвимости:

1. получение всех объявленных потоков;
2. анализ потоков на использование (чтение или запись) одних и тех же переменных;
3. анализ конструкций синхронизаций в исполняемых функциях потоков, а именно проверка наличия мьютексов, а также проверка освобождения мьютексов.

Между объявленными потоками, переменными, мьютексами и исполняемыми функциями может быть любое количество произвольного кода или других уязвимостей.

1. Случайные числа криптографического характера

Зачастую при решении криптографических задач используются случайные числа, которые получают с помощью генераторов случайных чисел. Алгоритмы их получения должны быть криптостойкими, иначе это может привести к возникновению угроз безопасности. Частой ошибкой является инициализация генератора кода константным значением, так как в этом случае криптостойкость алгоритма значительно уменьшается, и значения последовательности можно вычислить. Рекомендуется использовать в качестве значения инициализации уникальное значение, например, текущее время.

Обработчик RandomGeneratorHandler руководствуется следующим шаблоном при поиске уязвимости:

1. поиск функций генерации случайных чисел;
2. проверка на то, что найденная функция генерации инициализирована константным значением.

Данная уязвимость не является составной, поэтому ее не может пересекать произвольный код или другие уязвимости.

1. Пренебрежение безопасным хранением данных

Уязвимость возникает при встраивании конфиденциальной информации в код программы и некорректном использовании элементов управления доступом к файлам, дескрипторам и другим данным.

Обработчик DataStorageManagmentHandler руководствуется следующим шаблоном при поиске уязвимости:

1. поиск функций, которые изменяют уровень доступа к данным.

Данная уязвимость не является составной, поэтому ее не может пересекать произвольный код или другие уязвимости.

1. Утечка информации

Злоумышленник получает данные, которые могут привести к нарушению политики безопасности. Утечка информации может быть случайной и намеренной. При случайной утечке, информация может просочиться по неочевидному каналу, либо из-за алгоритмической ошибки в программы. Намеренная же возникает, когда команда программистов и конечных пользователей неодинаково понимают, что нужно защищать. Яркий пример утечки – слишком подробное логирование действий, происходящих в программе или чрезмерное количество информации об ошибках.

Обработчик DataLeakHandler руководствуется следующим шаблоном при поиске уязвимости:

1. поиск функций, которые могут приводить к утечке информации.

Данная уязвимость не является составной, поэтому ее не может пересекать произвольный код или другие уязвимости.

### Модуль пользовательского интерфейса

Модуль отвечает за реализацию графического интерфейса, с помощью которого пользователь использует функционал анализатора.

**1. Главное окно.**

Главное окно приложения (рисунок 3) можно поделить на три основополагающие части:

1) Блок доступных к анализу уязвимостей, расположенный в левой части главного окна (рисунок 3). Данный блок предназначен для выбора тех уязвимостей, которые будут исследоваться при анализе выбранной программы. Можно выбрать как одну отдельную уязвимость, так и несколько сразу.

2) Блок, в котором располагается список всех тестов, написанных участниками бригады, согласно составленной таблице ответственности по написанию тестов. Блок расположен в центре главного окна (рисунок 2). Здесь же располагаются две активные кнопки: «Проверить все» - запускает анализ всех тестов в данном блоке по выбранным уязвимостям, и «Проверить» - запускает анализ только выбранных тестов в данном блоке по выбранным уязвимостям.

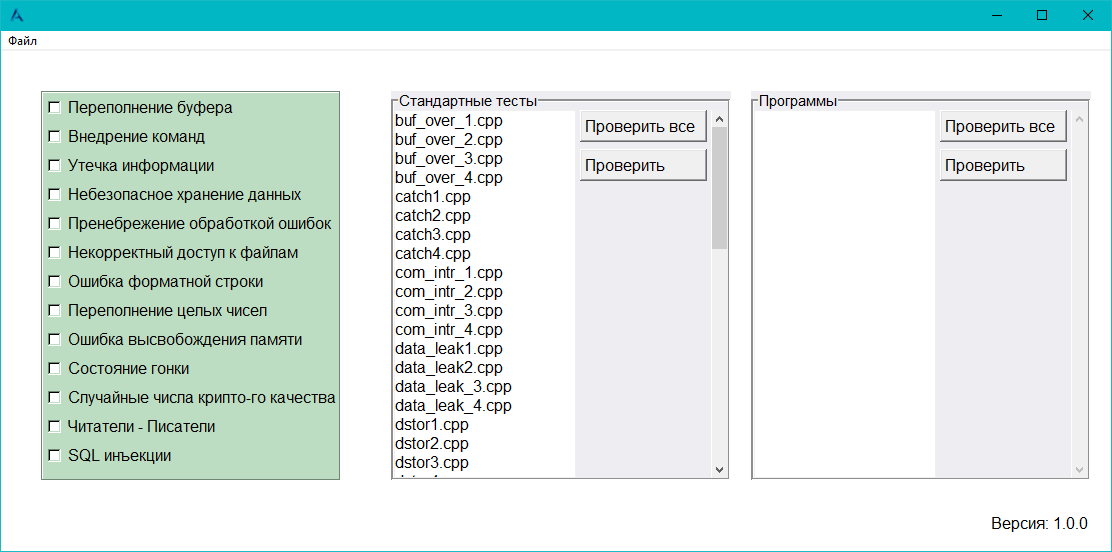
****3) Блок, в котором располагается список программ, которые пользователь самостоятельно загружает в анализатор. Расположен в правой части главного окна (рисунок 2). Функциональность кнопок «Проверить все» и «Проверить» аналогична центральному блоку.

Рисунок 3 – Главное окно

Загрузка собственных программ в анализатор происходит путем нажатия кнопки «Файл», расположенного в левом верхнем углу главного окна (рисунок 4), и далее кнопки «Открыть». После этого появится окно выбора программ (рисунок 5).

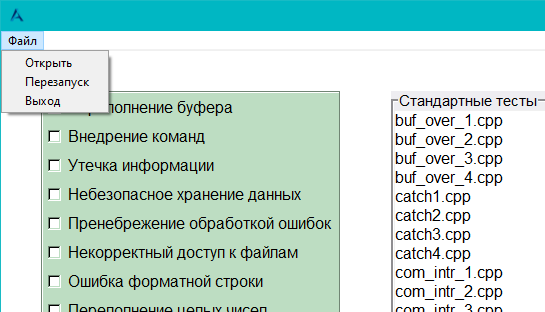


Рисунок 4 – Кнопка «Файл»

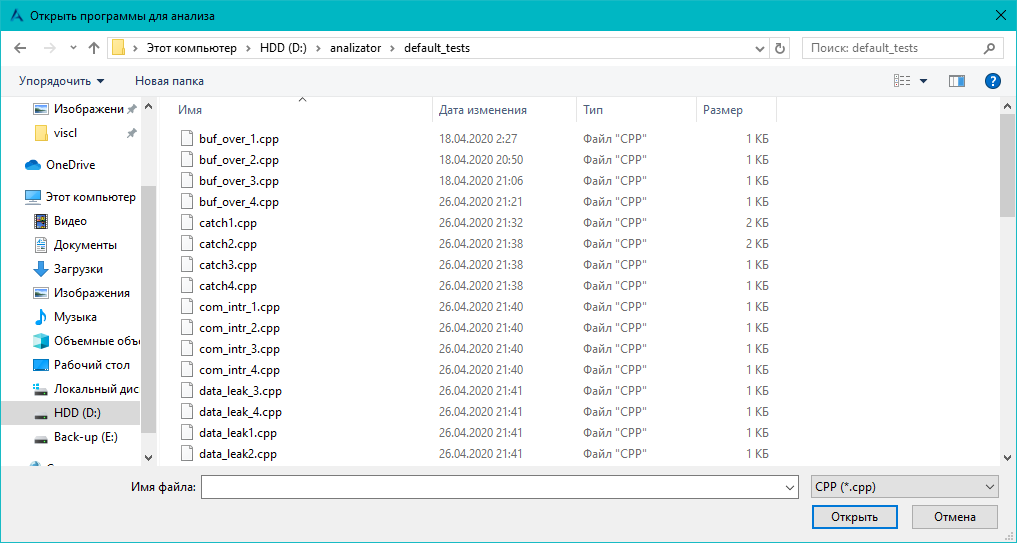


Рисунок 5 – открытие файлов для анализа

Также в меню «Файл» доступны пункты: «Перезапуск» - перезапуск программы и «Выход» - выход из программы.

**2. Окно анализа программы.**

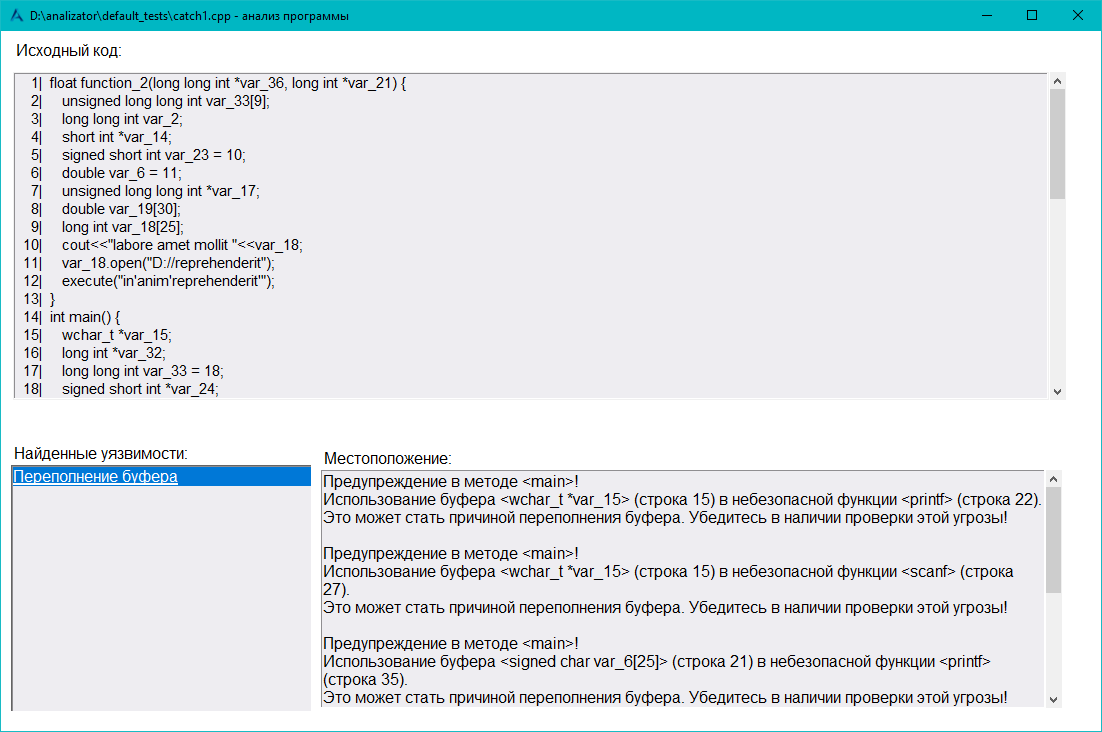
При нажатии на одну из кнопок «Проверить все» или «Проверить» откроется окно анализа выбранной программы или сразу несколько окон в случае выбора нескольких программ, по одному окну на каждую программу (рисунок 6).

Рисунок 6 – окно анализа программы.

В верхней части данного окна для более удобного просмотра результатов анализа располагается анализируемый код. В нижней части – список найденных уязвимостей и их подробные описания, а именно информация по каждой найденной уязвимости каждого типа и ее местоположение (номер строки в коде).

В случае, если в программе не будут найдены выбранные уязвимости появится информационное окно (рисунок 7).

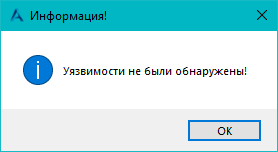


Рисунок 7 – информационное окно

Был рассмотрен и реализован ряд действий при возникновении внештатных ситуаций при работе программы.

1. Если пользователь не выберет ни одного теста и начнет анализ, то появится информационное окно (рисунок 8), при нажатии на которое пользователь сможет продолжить работу.

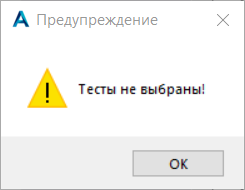


Рисунок 8 – информационное окно

1. В случае с отсутствием выбора программ ситуация – такая же (рисунок 9).

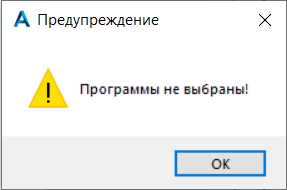


Рисунок 9 – информационное окно

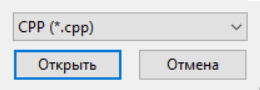
1. Во избежание нештатной ситуации, связанной с открытием файла неизвестного формата, вместо программы предусмотрен контроль открытия файлов (рисунок 10). Разрешенный формат – «.cpp»

Рисунок 10 – разрешенный формат для анализа

1. В случае с отсутствием выбора уязвимости аналогична ситуация с 1 и 2 (рисунок 11)

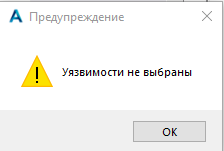


Рисунок 11 – информационное окно

### Агрегирующий модуль

Данный модуль отвечает за координацию и объединение всех предыдущих модулей в одно целое.

### Модуль тестирования и генерации тестового кода С++

Разработанный статический анализатор прошел отладку и ручные тесты, написанные участниками бригады (подробнее об этом в 3 главе), но важным показателем правильности его работы является тестирование с помощью автоматического генератора кода, так как он позволяет провести статистический анализ количества сгенерированных ошибок к количеству найденных ошибок.

Модуль генератор кода разработан как консольное приложение, обособленное от остальных частей проекта. Оно позволяет генерировать произвольный код с заданными уязвимостями, которые будут пересекаться и чередоваться друг с другом, произвольным кодом и другими уязвимостями. При генерации осуществляется подсчет каждой уязвимости. Для проведения статистического анализа указывается количество тестов к генерации, а также уязвимости, которые должны быть сгенерированы.

Так как программа для генерации кода предназначена лишь для отладки основной программы, для ее управления вместо графического интерфейса был использован консольный интерфейс для минимизации затрат времени на разработку данного утилитного приложения.

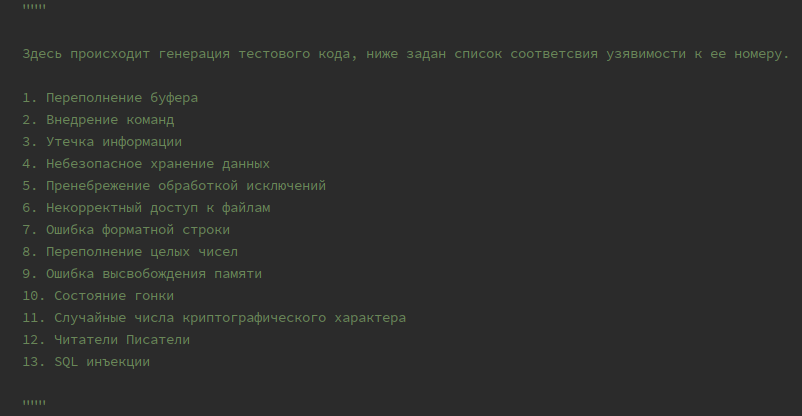
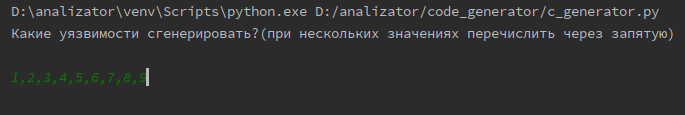
В коде самого генератора, в виде закомментированной документации указан пронумерованный список уязвимостей (рисунок 12).

Рисунок 12 – соответствие уязвимости к ее номеру

При запуске модуля генерации будет предложено ввести номера выбранных уязвимостей, которые точно должны появиться в сгенерированном коде (рисунок 13). При необходимости генерации всех уязвимостей сразу, достаточно просто ввести «-1».

Рисунок 13 – ввод номеров уязвимостей в консоли

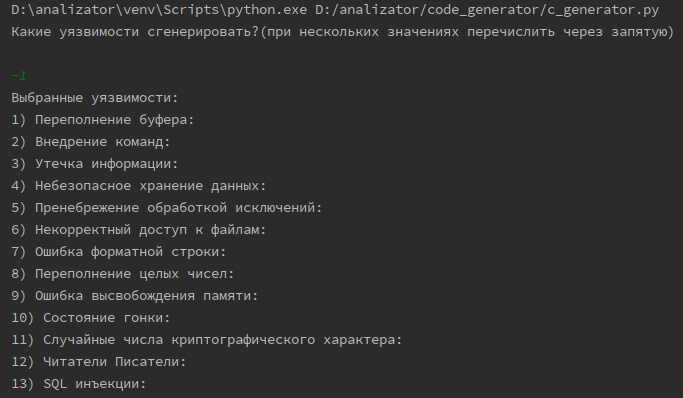
Далее будет отображен список выбранных для генерации уязвимостей (рисунок 14). Последний шаг – выбор количества тестов (рисунок 15).

Рисунок 14 – отображение выбранных уязвимостей

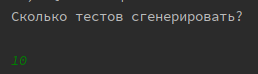


Рисунок 15 – выбор количества тестов

Алгоритм генерации построен на использовании библиотеки Lib/random.py. Сам модуль генерации состоит из функций, каждая из которых генерирует свою часть кода.

Все функции можно разделить на два основных типа. Первый тип функций отвечает за генерацию уязвимостей. Второй тип отвечает за генерацию обычного произвольного кода, например, объявление переменных, построение условных конструкций «if», объявление сигнатур функций и т.д.

В коде программы всегда генерируется несколько вспомогательных функций. Они нужны, например, для воссоздания уязвимостей многопоточности, так как используются потоками в качестве исполняемых функций. Помимо этих функций, всегда создается основная функция «main», в которой и будет размещен уязвимый код. При генерации каждой уязвимости, если она является составной, т.е. может пересекаться другим кодом, перед уязвимостью, между частями уязвимости и после нее генерируется произвольный код, максимальное количество которого контролируется заданным в параметрах генератора интервалом. Общий размер теста не является детерминированной величиной ввиду использования псевдослучайного генератора из библиотеки Lib/random.py для определения количества произвольного кода, генерируемых переменных, функций и т.д. Но задавая параметры генерации внутри класса генератора, можно оценивать примерный размер генерируемого кода. Так при стандартных параметрах, указанных ниже, а также при всех выбранных для генерации уязвимостей размер кода варьируется от 130 до 300 строк.

1. Максимальное количество вспомогательных функций – 4;
2. Максимальное количество переменных в функциях – 7;
3. Максимальное количество блоков произвольного кода перед уязвимостью, между ее частями и после уязвимости - 3.

Создание данного модуля – общая задача бригады, ответственность за выполнение своей части работы, а также ее отладке была распределена согласно 3 таблице. Подробное описание каждой части генератора, а также информацию об ее отладке можно найти в индивидуальном отчете участника бригады, наряду с другими разработанными модулями программы. Распределение по функциям генерации уязвимостей соответствует распределению по обработчикам уязвимостей. Так, если, например, Терентьев реализовывал обработчик внедрения команд, он также отвечает за создание функции генерации этой уязвимости.

Таблица 3 – распределение частей генератора по исполнителям

|  |  |
| --- | --- |
| Ковалев К.А. | 1) Генератор уязвимости переполнения буфера;  2) Генератор уязвимости пренебрежения обработок ошибок;  3) Генератор уязвимости задачи Читатели – Писатели;  4) Генератор уязвимости состояния гонки;  5) Функция, отвечающая за агрегирование всех остальных генераторов. |
| Терентьев Е.А. | 1) Генератор уязвимости внедрения SQL команд;  2) Генератор уязвимости внедрения команд;  3) Генератор проблем, связанных с динамическим выделением памяти. |
| Щекочихин Д.А. | 1) Генератор уязвимости ошибки форматной строки;  2) Генератор уязвимости некорректного доступа к файлам. |
| Оганисян Т.С. | 1) Генератор уязвимости случайных чисел криптографического характера;  2) Генератор уязвимости переполнения целых чисел;  3) Генератор условного оператора «if». |
| Шабакаев А.Р. | 1) Генератор уязвимости пренебрежения безопасным хранением данных;  2) Генератор уязвимости утечки информации;  3) Генераторы переменных. |

Алгоритм статистического анализа:

1. Генерация кода;
2. Анализ сгенерированного кода;
3. Вывод отчета о количестве сгенерированных и найденных ошибок.

С помощью полученного отчета можно вести отладку приложения и делать вывод о правильности его работы.

## Вывод и результаты

В результате был разработан статический анализатор кода, способный обнаруживать все 13 заявленных уязвимостей. Также для проверки работоспособности программы был разработан автоматический генератора кода С++.

# ТЕСТИРОВАНИЕ

Для тестирования статического анализатора использовались два подхода:

1. Тесты, написанные членами бригады. По два теста на разрабатываемую уязвимость и по два теста на уязвимость «соседа». Итого по 4 теста на все 13 уязвимостей, то есть 52 теста. Все тесты указаны в индивидуальных частях отчета каждого из участников.

Таблица 4 - распределение тестов по исполнителям

|  |  |
| --- | --- |
| Ковалев К.А. | 1) Тесты для обработчика переполнения буфера;  2) Тесты для обработчика пренебрежения обработок ошибок;  3) Тесты для обработчика задачи Читатели – Писатели;  4) Тесты для обработчика состояния гонки; |
| Терентьев Е.А. | 5) Тесты для обработчика внедрения SQL команд;  6) Тесты для обработчика внедрения команд.  7) Тесты для обработчика проблем, связанных с динамическим выделением памяти; |
| Щекочихин Д.А. | 8) Тесты для обработчика ошибки форматной строки;  9) Тесты для обработчика некорректного доступа к файлам; |
| Оганисян Т.С. | 10) Тесты для обработчика случайных чисел криптографического характера;  11) Тесты для обработчика переполнения целых чисел. |
| Шабакаев А.Р. | 12) Тесты для обработчика пренебрежения безопасным хранением данных;  13) Тесты для обработчика утечки информации. |

Каждый из членов команды пишет тесты на свою уязвимость и на «уязвимость соседа», т.е. Шабакаев пишет тесты для уязвимостей Оганисяна, Оганисян для Щекочихина, Щекочихин для Терентьева, Терентьев для Ковалева, Ковалев для Шабакаева.

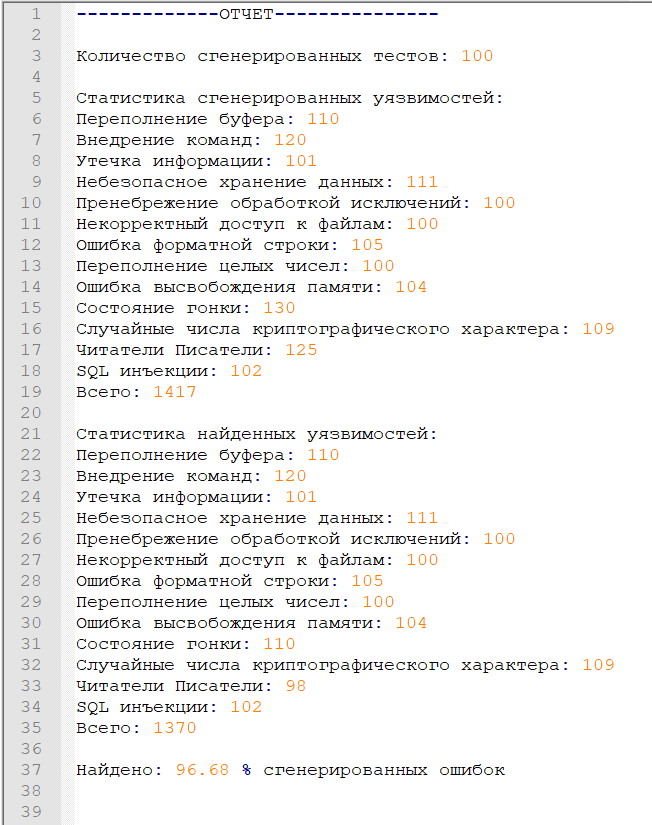
1. Автоматический генератор тестового кода C++. С помощью формируемого им отчета (рисунок 16) производилась отладка и делался вывод о правильности работы программы.

Рисунок 16 – статистический отчет о работе анализатора

Помимо того, что каждый генератор уязвимости проходил тестирование и отладку ответственным за его создание участником бригады, проводилась общая проверка правильности сгенерированных тестов. На рисунке 17 изображен код, одного из сгенерированных тестов.

mutex testMutex4;

singed long int function\_5(signed short int var\_402, signed long long int \*var\_1925, byte var\_1320, float \*var\_798) {

testMutex4.lock();

long double \*var\_1017;

unsigned short int var\_935[45];

unsigned long long int var\_144;

double var\_103[38];

unsigned char var\_1028 = 7;

unsigned short int var\_1116;

long int var\_1939;

strerror(var\_1925);

}

int main(long long int \*var\_165) {

unsigned long int var\_1313 = 24;

byte var\_1319 = 19;

long double var\_1903 = 8;

double \*var\_665;

double \*var\_139;

unsigned int var\_268;

int var\_1407;

22. executeQuery(var\_139);

memcpy(var\_1903, var\_165, var\_139);

24. utime(var\_1903, var\_1407);

25. ShellExecute(var\_1407, var\_139, var\_268, var\_1319, var\_165, var\_1903);

26. printf(Format %s and %s, var\_139, var\_665);

27. perror(var\_1407);

perror(var\_1319);

GetEnvironmentStrings();

utime(var\_268, var\_1319);

SHGetFolderPath(var\_1319, var\_165, var\_665, var\_1407, var\_268);

32. SetKernelObjectSecurity(var\_165, var\_1319, var\_139);

try {

printf(Format %s and %s, var\_1903, var\_665);

executeQuery(var\_665);

}

catch (Exception\_11 err) {

ShellExecute(var\_1313, var\_665, var\_165, var\_139, var\_1903, var\_1407);

chmod(var\_1313, var\_268);

}

printf(Format %s and %s, var\_665, var\_1407);

chmod(var\_1313, var\_1319);

execvp(var\_139, var\_165);

printf(Format %s and %s, var\_1313, var\_665);

mktemp(var\_1903);

cout<<errno;

utime(var\_165, var\_665);

system(var\_165);

printf(Format %s and %s, var\_139, var\_1319);

executeQuery(var\_1903);

printf(Format %s and %s, var\_139, var\_1313);

popen(var\_1313, var\_665);

short var\_1518 = 25223;

long var\_27 = 6790476784283491845;

short var\_1716 = 28946;

56. scanf(%s, var\_139);

57. var\_1518 = var\_27 + var\_1716;

unsigned long long int \*var\_1147 = new unsigned long long int;

short int \*var\_69 = new short int;

execute(var\_139);

utime(var\_268, var\_1903);

printf(Format %s and %s, var\_139, var\_27);

63. printf(Format %s and %s, var\_165, var\_268);

vsprintf(var\_1903, var\_268, var\_165);

65. std::thread t0(function\_5, var\_1313, var\_1903, var\_1147);

GetEnvironmentVariable(var\_1313);

printf(Format %s and %s, var\_1147, var\_268);

mkdir(var\_1313);

69. std::thread t1(function\_5, var\_1147, var\_1518, var\_1716);

printf(Format %s and %s, var\_268, var\_1147);

std::uniform\_real\_distribution<> dis(2, 3);

72. std::uniform\_real\_distribution<> dis(1, 4);

rand();

short var\_399 = 27648;

int var\_1654 = 1122740508;

short var\_776 = 19983;

if(var\_1319 != 191) {

printf(Format %s and %s, var\_69, var\_1903);

printf(Format %s and %s, var\_1147, var\_1654);

execute("ea'tempor'dolor'");

std::uniform\_real\_distribution<> dis(2, 1);

82. strcpy(var\_1313, var\_165);

}

var\_399 = var\_1654 + var\_776;

popen(var\_1654, var\_69);

86. std::thread t0(function\_5, var\_1903, var\_1313, var\_1407, var\_1654);

execlp(var\_1407, var\_1654, var\_1319);

printf(Format %s and %s, var\_165, var\_776);

ShellExecute(var\_139, var\_1518, var\_27, var\_268, var\_165, var\_1407);

90. std::thread t1(function\_5, var\_399, var\_27, var\_665, var\_1518);

91. signed int \*var\_1486 = new signed int;

execvp(var\_1716, var\_27);

93. std::thread t2(function\_5, var\_1716, var\_665, var\_1407, var\_1654);

execute(var\_1716);

std::uniform\_real\_distribution<> dis(3, 2);

chmod(var\_268, var\_1407);

97. std::thread t3(function\_5, var\_1147, var\_1319, var\_1903, var\_165);

execlp(var\_139, var\_1147, var\_665);

strncpy(var\_1313, var\_69, var\_1654);

execute("consectetur'velit'nostrud'");

try {

scanf(%s, var\_13);

102. catch (Exception\_3 err) {

}

}}

Рисунок 17 – сгенерированный тест.

В этом тесте для генерации были указаны все 13 уязвимостей. Помимо функции «main», в которой содержаться все уязвимости, сгенерирована вспомогательная функция для уязвимостей многопоточности – «function\_5». Убедимся в том, что все указанные уязвимости сгенерированы в функции «main»:

1. В строке 56 сгенерирована уязвимость переполнения буфера;
2. В строке 25 сгенерирована уязвимость внедрения команд;
3. В строке 27 сгенерирована уязвимость утечки информации;
4. В строке 32 сгенерирована уязвимость небезопасного хранения данных;
5. В строке 102 сгенерирована уязвимость пренебрежения обработкой исключений;
6. В строке 24 сгенерирована уязвимость некорректного доступа к файлам;
7. В строке 63 сгенерирована уязвимость форматной строки;
8. В строке 57 сгенерирована уязвимость переполнения целых чисел;
9. В строке 91 сгенерирована уязвимость высвобождения памяти;
10. В строках 65, 69, 86, 90, 93, 97 сгенерированы потоки, использующие одни туже исполняемую функцию «function\_5», т.е. сгенерирована уязвимость состояния гонки;
11. В строке 72 сгенерирована уязвимость случайных чисел криптографического характера;
12. В строке 65, 86 сгенерированы потоки, использующие одну и ту же переменную «var\_1313», а также в их исполняемой функции «function\_5» блокируется, но не высвобождается мьютекс «testMutex4», т.е. сгенерирована уязвимость задачи Читатели-Писатели.
13. В строке 22 сгенерирована уязвимость SQL инъекции.

Следует отметить, что количество найденных ошибок многопоточности всегда меньше количества сгенерированных, это обусловлено случайностью генерации потоков. Невозможно точно подсчитать сколько сгенерированных потоков будут использовать одну и туже исполняемую функцию или одни и те же параметры, то есть будут действительно уязвимы.

В результате использования двух вышеприведенных подходов программа была полностью отлажена и протестирована на соответствие поставленным целям и задачам.

Результаты проверки анализатором сгенерированного теста, приведенного на рисунке 17.

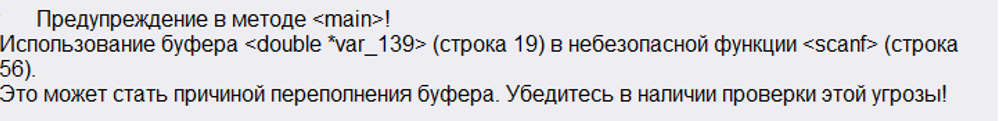
1)

Рисунок 18 – найденная уязвимость переполнения буфера.

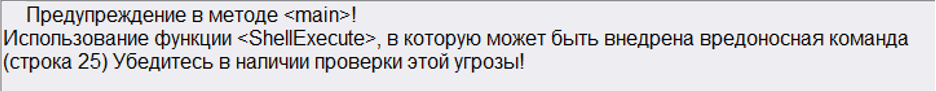
2) 

Рисунок 19 – найденная уязвимость внедрения команд.

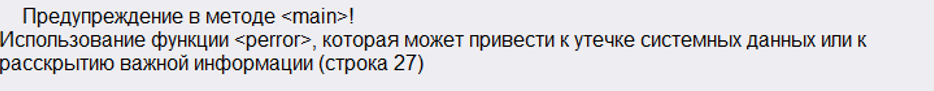
3) 

Рисунок 20 – найденная уязвимость утечки информации.

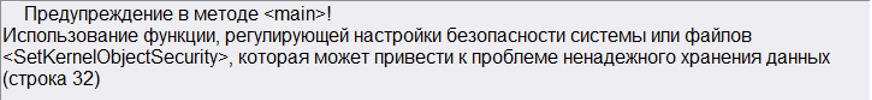
4) 

Рисунок 21 – найденная уязвимость небезопасного хранения данных.

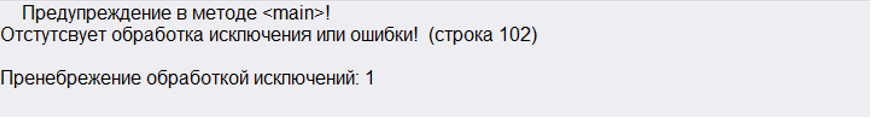
5)

Рисунок 22 – найденная уязвимость пренебрежения обработкой ошибок.

6)

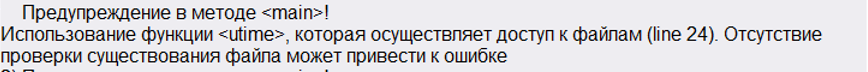


Рисунок 23 – найденная уязвимость некорректного доступа к файлам.

7)

Рисунок 24 – найденная уязвимость форматной строки.

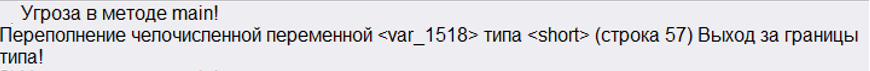
8) 

Рисунок 25 – найденная уязвимость переполнения целых чисел.

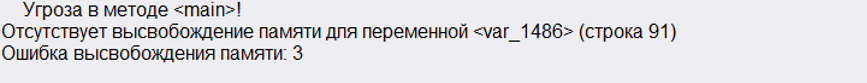
9) 

Рисунок 26 – найденная уязвимость высвобождения памяти.

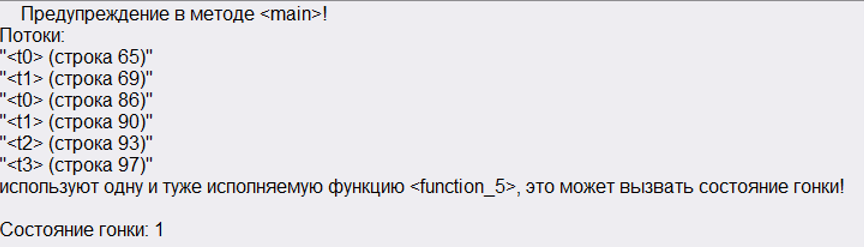
10) 

Рисунок 27 – найденная уязвимость состояния гонки.

11)

Рисунок 28 – найденная уязвимость случайных чисел криптографического характера.

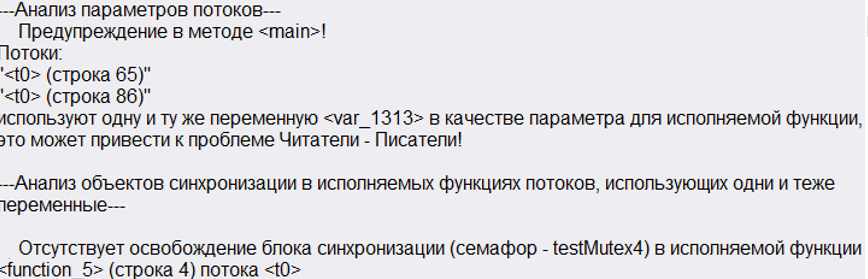
12)

Рисунок 29 – найденная уязвимость задачи Читатели-Писатели.

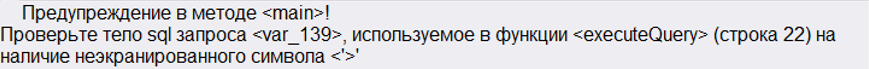
13) 

Рисунок 30 – найденная уязвимость SQL инъекции.

# ИНДИВИДУАЛЬНАЯ ЧАСТЬ

В ходе работы над курсовым проектом мною были решены следующие задачи:

1. Разработка общего алгоритма работы программы;
2. Разработка программы:
3. Обработчик проблем, связанных с динамическим выделением памяти;
4. Обработчик внедрения SQL команд;
5. Обработчик внедрения команд;
6. Модуль тестирования и генерации тестового кода;
7. Разработка тестов (указаны ниже в отдельных частях пункта «разработка программы»);
8. Разработка общего алгоритма работы программы

В ходе анализа предметной области было решено использовать гибридный подход, то есть использование комплексного анализа с разбиением программы на лексемы и анализ их с помощью регулярных выражений. Данный подход позволил осуществить разработку анализатора, способного находить распределенные по программному коду уязвимости.

Анализатор принимает на вход код программы на языке C++. Для удобства анализа принятая программа преобразуется в набор строк, а также производится очистка кода от лишних пробелов и комментариев. Далее код попадает в модуль первичного лексического анализа, где происходит построение объекта, характеризующего структуру анализируемой программы. В нем хранится информация о всех объявленных переменных программы, функциях и потоках.

Далее сформированный объект поступает в модуль обработки уязвимостей, где и происходит анализ на наличие уязвимостей, выбранных пользователем.

После этого формируется отчет работы обработчиков уязвимостей, который выдается пользователю.

1. Разработка программы
   1. Обработчик проблем, связанных с динамическим выделением памяти

Исходный код:

1. class MemoryLeakHandler(BaseHandler):  
    vulnerability\_name = 'Ошибка высвобождения памяти'  
     
    def \_\_init\_\_(self):  
    self.pattern = r'^\s\*delete(\(.\*\))'  
    self.output = []  
     
    def parse(self, contexts: List[FunctionContext]):  
    for context in contexts:  
    m\_vars = []  
    for var in context.variables:  
    if var.value is not None:  
    if "new" in var.value:  
    m\_vars.append(var)  
    free\_matches = []  
    for \_, line in context.source\_code.items():  
    matches = re.finditer(self.pattern, line, re.IGNORECASE)  
    free\_matches += matches  
    for var in m\_vars:  
    if var.var\_name not in list(map(lambda x: x.group(1)[1:-1].strip(), free\_matches)):  
    self.output.append(f"Угроза в методе <{context.name}>!\n"  
    f"Отсутствует высвобождение памяти для переменной <{var.var\_name}> (строка {var.line\_appeared})")  
     
    return self.output

Данный модуль производит поиск в коде программы функций выделения памяти, таких как функция new, а также запоминает переменную, для которой была выделена память. Далее производится поиск вызовов функции delete – функции высвобождения памяти, а также параметров, с которыми был произведен вызов данной функции. После этого производится поиск переменных, под которые была выделена память, среди параметров, с которыми вызывалась функция высвобождения памяти. Если совпадений не найдено, то пользователю выдается отчет об ошибке, указывающий переменную и номер строки, в которой была обнаружена уязвимость.

Поиском и анализом данной уязвимости занимается класс «MemoryLeakHandler».

* 1. Обработчик внедрения SQL команд

Исходный код:

1. class SQLInjectionHandler(BaseHandler):  
    vulnerability\_name = 'SQL инъекции'  
     
    def \_\_init\_\_(self):  
    self.pattern = r'(executeQuery|execute)(\(.\*\))'  
    self.output = []  
     
    def parse(self, contexts: List[FunctionContext]):  
    for context in contexts:  
    for line\_number, line in context.source\_code.items():  
    matches = re.finditer(self.pattern, line)  
    for match in matches:  
    tmp = match.group(2)[1:-1]  
    if tmp[0] == '\"': # check if parameter is literally string like "test string"  
    if re.search(r'[^\\]\'', tmp) is not None:  
    self.output.append(f"Угроза в методе <{context.name}>!\n"  
    f"Тело sql запроса <{tmp[1:-1]}>, используемое в функции <{match.group(1)}> (строка {line\_number}) "  
    f"имеет неэкранированный символ(ы) <'>\nЭто может привести к sql инъекции")  
    else:  
    self.output.append(f"Предупреждение в методе <{context.name}>!\n"  
    f"Проверьте тело sql запроса <{tmp}>, используемое в функции <{match.group(1)}> (строка {line\_number}) "  
    f"на наличие неэкранированного символа <'>'\nЭто может привести к sql инъекции")  
    return self.output

Данный модуль производит поиск в коде программы функций, исполняющих SQL-запросы, таких как exequteQuery и execute. Далее происходит проверка тела SQL-запроса, на наличие неэкранированной одинарной кавычки, что может означать наличие в теле запроса исполняемого кода. При обнаружении данной уязвимости пользователю выдается отчет об ошибке, содержащий название метода, в котором был осуществлен запрос, используемая функция, а также номер строки в коде.

Поиском и анализом данной уязвимости занимается класс «SqlInjectionHandler».

* 1. Обработчик внедрения команд

Исходный код:

lass CommandsIntroductionHandler(BaseHandler):  
  
 vulnerability\_name = 'Внедрение команд'  
  
 """  
 - int system (const char\* command); - выполняет системную команду.  
   
 - FILE \*popen(const char \*command, const char \*type); - создает дочерний процесс, которому можно передать команду.  
   
 - int execlp(const char \*file, const char \*arg, ...); - выполнение исполняемого файла по пути <file> с передачей   
 параметра в <arg>.  
   
 - int execvp(const char \*file, char \*const argv[]); - тоже что и execlp, но параметры передаются в <argv>.  
   
 - HINSTANCE ShellExecute(HWND hwnd, LPCSTR lpOperation, LPCSTR lpFile, LPCSTR lpParameters, LPCSTR lpDirectory, INT nShowCmd); -   
 выполняет команду <lpOperation> над указанным файлом <lpFile>.  
 """  
  
 def \_\_init\_\_(self):  
 self.pattern = r'(system|popen|execlp|execvp|ShellExecute)'  
 self.output = []  
  
 def parse(self, contexts: List[FunctionContext]):  
 for context in contexts:  
 for line\_number, line in context.source\_code.items():  
 matches = re.finditer(self.pattern, line)  
 for match in matches:  
 self.output.append(f"Предупреждение в методе <{context.name}>!\n"  
 f"Использование функции <{match.group(1)}>, в которую может быть внедрена"  
 f" вредоносная команда (строка {line\_number}) "  
 f"Убедитесь в наличии проверки этой угрозы!\n")  
  
 return self.output

Данный модуль производит поиск в коде программы вызовов опасных функций, позволяющих выполнять исполняемые файлы и системные команды, а также создавать дочерние процессы, которым можно передать команду на выполнение. Ведется поиск следующих функций:

1. int system (const char\* command); - выполняет системную команду;
2. FILE \*popen(const char \*command, const char \*type); - создает дочерний процесс, которому можно передать команду;
3. int execlp(const char \*file, const char \*arg, ...); - выполнение исполняемого файла по пути <file> с передачей параметра в <arg>;
4. int execvp(const char \*file, char \*const argv[]); - тоже что и execlp, но параметры передаются в <argv>;
5. HINSTANCE ShellExecute(HWND hwnd, LPCSTR lpOperation, LPCSTR lpFile, LPCSTR lpParameters, LPCSTR lpDirectory, INT nShowCmd); - выполняет команду <lpOperation> над указанным файлом <lpFile>;

При обнаружении вызова одной из данных функций пользователю выдается предупреждение о возможном наличии уязвимости, название метода, в котором произведен вызов функции, сама функция, а также номер строки в коде.

Поиском и анализом данной уязвимости занимается класс «CommandsIntroductionHandler».

* 1. Модуль тестирования и генерации тестового кода

1. Генератор уязвимости, связанной с динамическим выделением памяти

Исходный код:

def free\_error(indent, params, a\_f):  
 var\_type = r\_v(constants.types)  
 var\_name = "var\_" + str(rng(1, max\_val))  
 while var\_name in params:  
 var\_name = "var\_" + str(rng(1, max\_val))  
 params.append(var\_name)  
 code = indent + var\_type + " \*" + var\_name + " = new " + var\_type + ";\n"  
 return code

Функция отвечает за генерацию уязвимости, связанной с динамическим выделением памяти.

В начале функция с помощью псевдослучайного генератора библиотеки Lib/random генерирует имя и тип переменной, для которой будет выделена память. Затем формируется строка кода, содержащая сгенерированную переменную, а также вызов функции «new», обеспечивающую динамическое выделение памяти для этой переменной. На выходе функция возвращает строку, содержащую уязвимость.

В ходе разработки генератора производилась отладки каждой функции, а также контроль правильности генерации кода. Это возможно благодаря тому, что функции можно запускать отдельно, не запуская весь код генератора. Пример вывода функции, с помощью которого производилась отладка приведен на рисунке 3.

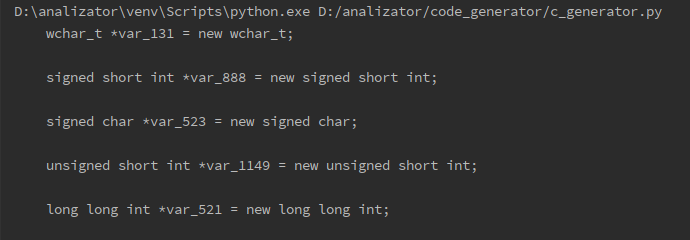


Рисунок 31 – пример отладки генератора

1. Генератор уязвимости внедрения SQL команд

Исходный код:

def sql\_error(indent, params, a\_f):  
 funcs = ["execute(", "executeQuery("]  
 cur\_func = r\_v(funcs)  
 sample\_text = ""  
 for \_ in range(3):  
 sample\_text += r\_v(sample\_words) + "'"  
 random = rng(0, 1)  
 if random == 0:  
 code = indent + cur\_func + "\"" + sample\_text + "\");\n"  
 else:  
 code = indent + cur\_func + r\_v(params) + ");\n"  
 return code

Функция отвечает за генерацию уязвимости внедрения SQL команд.

В начале функция с помощью псевдослучайного генератора библиотеки Lib/random выбирает случайную небезопасную функцию. Затем генерируется строка кода, в которую входит вызов небезопасной функции с параметрами. В качестве параметров могут выступать либо случайно сгенерированная строка текста, содержащая уязвимость, либо уже объявленная ранее переменная. На выходе функция возвращает строку, содержащую уязвимость.

В ходе разработки генератора производилась отладки каждой функции, а также контроль правильности генерации кода. Это возможно благодаря тому, что функции можно запускать отдельно, не запуская весь код генератора. Пример вывода функции, с помощью которого производилась отладка приведен на рисунке 4.

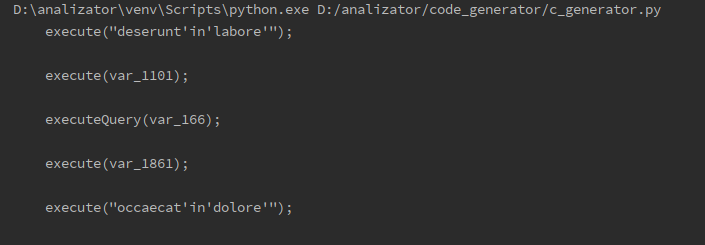


Рисунок 32 – пример отладки генератора

1. Генератор уязвимости внедрения команд

Исходный код:

def c\_intr\_error(indent, params, a\_f):  
 funcs = [  
 {"system(": 1},  
 {"popen(": 2},  
 {"execlp(": 3},  
 {"execvp(": 2},  
 {"ShellExecute(": 6},  
 ]  
 cur\_func\_dict = r\_v(funcs)  
 cur\_func = list(cur\_func\_dict.keys())[0]  
 n = list(cur\_func\_dict.values())[0]  
 chosen\_param\_list = [params[param] for param in gen\_n\_rands(n, 0, len(params) - 1)]  
 code = indent + cur\_func + ", ".join(chosen\_param\_list) + ");\n"  
 return code

Функция отвечает за генерацию уязвимости внедрения команд.

В начале функция с помощью псевдослучайного генератора библиотеки Lib/random выбирает случайную небезопасную функцию. Затем генерируется строка кода, в которую входит вызов небезопасной функции с параметрами. Число параметров зависит от числа принимаемых небезопасной функцией параметров. Эти параметры функции выбираются из числа уже объявленных ранее переменных. На выходе функция возвращает строку, содержащую уязвимость.

В ходе разработки генератора производилась отладки каждой функции, а также контроль правильности генерации кода. Это возможно благодаря тому, что функции можно запускать отдельно, не запуская весь код генератора. Пример вывода функции, с помощью которого производилась отладка приведен на рисунке 5.

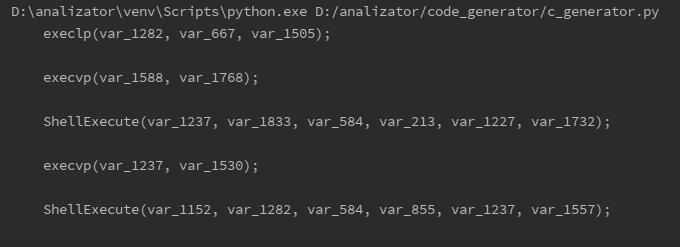


Рисунок 33 – пример отладки генератора

1. Разработка тестов

Для проверки правильности работы данных модулей были разработаны тестовые программы, в соответствии с распределением ролей: по 2 теста на разрабатываемую уязвимость и по 2 теста на уязвимости «соседа».

1. **Обработчик проблем, связанных с динамическим выделением памяти**

Тест № 1. Автор: Терентьев Е.А.

Исходный код:

float function\_5(unsigned int var\_1) {  
 unsigned int var\_23[35];  
 ifstream var\_36.open("D://eu");  
 unsigned char var\_31;  
 ofstream var\_6.open("D://dolore");  
 double \*var\_9;  
 unsigned char var\_38 = 24;  
 ofstream var\_19.open("D://velit");  
 cout<<"eiusmod dolore et "<<var\_1;  
 execute(var\_6);  
 printf(Format %s and %s, var\_19, var\_31);  
 try {  
 srand(14);  
 try {  
 cout<<"nostrud dolore occaecat "<<var\_19;  
 rmdir(var\_23);  
 GetLastError();  
 }  
 catch (Exception\_7 err) {  
 if(var\_38 != 37) {  
 execlp(var\_1, var\_9, var\_6);  
 printf(Format %s and %s, var\_23, var\_1);  
 }  
 executeQuery("voluptate'nulla'ea'");  
 }  
 if(var\_6 < 173) {  
 std::uniform\_real\_distribution<> dis(1, 5);  
 chmod(var\_6, var\_1);  
 }  
 }  
 catch (Exception\_8 err) {  
 std::uniform\_real\_distribution<> dis(1, 7);  
 GetLastError();  
 }  
 short var\_4 = 31140;  
 char var\_40 = 120;  
 byte var\_13 = 31701;  
 system(var\_36);  
 var\_4 = var\_40 + var\_13;  
 printf(Overflow %s and %s, var\_9, var\_23);  
}  
int main() {  
 ofstream var\_7.open("D://consectetur");  
 unsigned int var\_8 = 6;  
 ofstream var\_18.open("D://in");  
 byte var\_36 = 25;  
 short int \*var\_38;  
 signed short int var\_11 = 15;  
 long int var\_15[23];  
 execute(var\_15);  
 rmdir(var\_38);  
 unsigned int \*var\_27 = new unsigned int;  
}

Описание:

В данном тесте присутствует уязвимость в методе main. Отсутствует высвобождение памяти, выделенной под переменную var\_27.

Результат работы обработчика уязвимости:

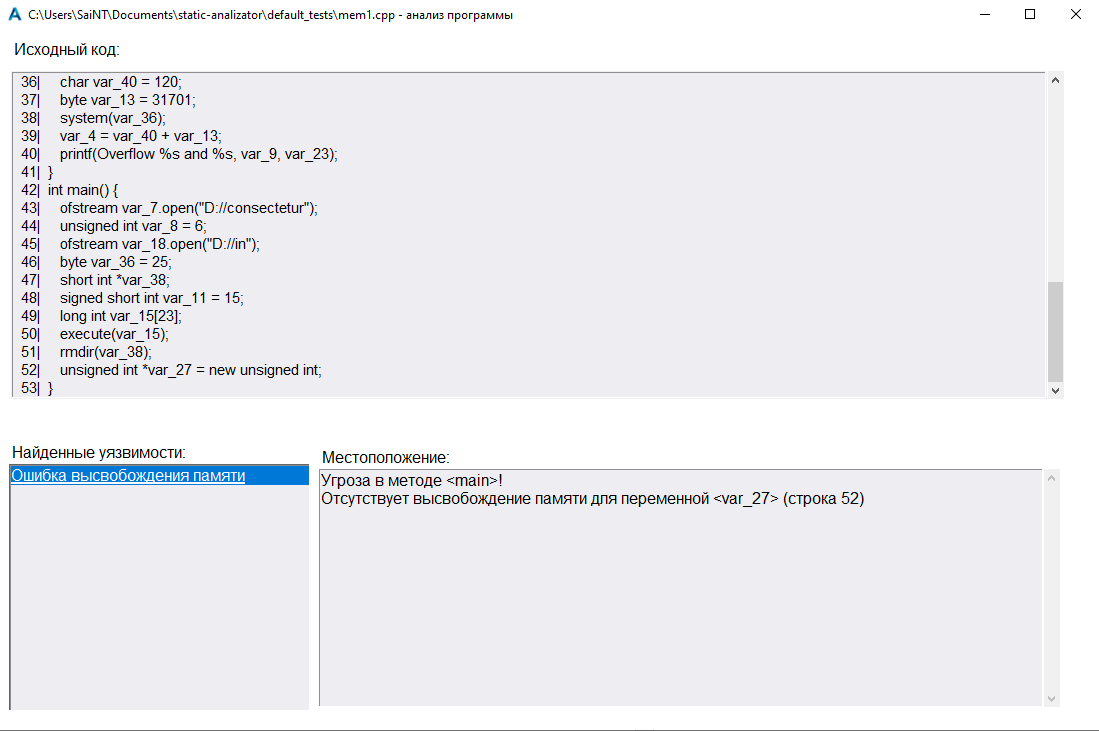


Рисунок 34 – Результат работы обработчика уязвимости над тестом №1

Обработчик успешно обнаружил указанную уязвимость.

Тест № 2. Автор: Терентьев Е.А.

Исходный код:

unsigned int function\_11(char var\_11, singed long int \*var\_2, signed long long int var\_15, unsigned long int var\_28) {  
 wchar\_t var\_21[15];  
 singed long int var\_13[47];  
 unsigned char var\_39 = 15;  
 ofstream var\_31.open("D://dolor");  
 unsigned char \*var\_20;  
 double var\_4;  
 char \*var\_7;  
 long int var\_27;  
 cout<<"non labore dolore "<<var\_20;  
 cout<<"Excepteur non deserunt "<<var\_27;  
 system(var\_20);  
 std::uniform\_real\_distribution<> dis(1, 9);  
 long long int \*var\_3 = new long long int;  
}  
int main() {  
 long double var\_3 = 25;  
 int \*var\_35;  
 signed short int var\_38 = 3;  
 char var\_19[7];  
 long int var\_8;  
 singed long int \*var\_6;  
 signed char var\_25 = 26;  
 srand(55);  
 SetServiceObjectSecurity(var\_40, var\_6, var\_38);  
 cout<<"cupidatat officia Duis "<<var\_8;  
 byte \*var\_17 = new byte;  
}

Описание:

В данном тесте присутствуют две уязвимости: в методе function\_11 и методе main. В первом случае отсутствует высвобождение памяти, выделенной под переменную var\_3. Во втором случае отсутствует высвобождение памяти, выделенной под переменную var\_17.

Результат работы обработчика уязвимости:

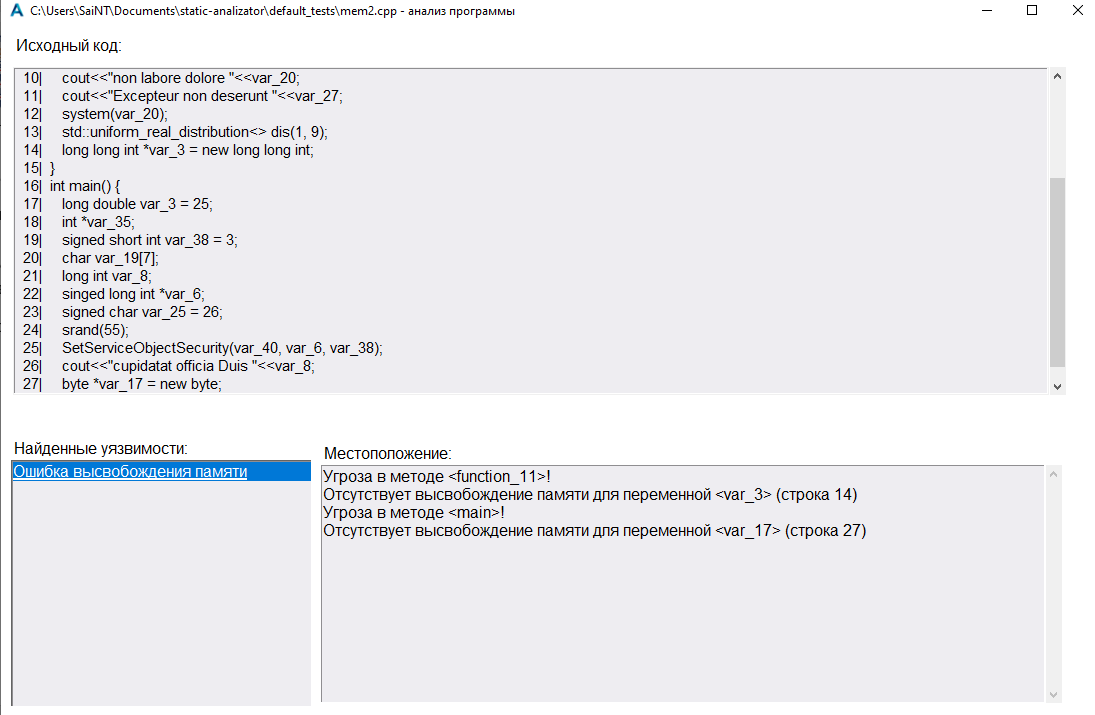


Рисунок 35 – Результат работы обработчика уязвимости над тестом №2

Обработчик успешно обнаружил указанные уязвимости.

Тест №3. Автор: Щекочихин Д.А.

Исходный код:

long long int function\_14(long long int var\_40, wchar\_t var\_36) {  
 short int var\_2[24];  
 unsigned long int var\_19;  
 int var\_33;  
 ofstream var\_29.open("D://in");  
 singed long int var\_28 = 30;  
 float var\_24;  
 wchar\_t var\_34 = 10;  
 ifstream var\_6.open("D://fugiat");  
 cout<<"aliquip irure esse "<<var\_34;  
 signed char \*var\_18 = new signed char;  
 if(var\_6 > 439) {  
 GetEnvironmentVariable(var\_34);  
 long var\_7 = 4732786967433569850;  
 byte var\_10 = 21594;  
 char var\_20 = 101;  
 popen(var\_34, var\_28);  
 var\_7 = var\_10 + var\_20;  
 }  
 long double \*var\_13 = new long double;  
}  
int main() {  
 unsigned char var\_14 = 29;  
 short int var\_8 = 4;  
 unsigned short int \*var\_36;  
 int var\_22 = 11;  
 ifstream var\_9.open("D://cupidatat");  
 long int var\_2;  
 char var\_35 = 117;  
 char var\_1 = 102;  
 short var\_37 = 23755;  
 executeQuery(var\_14);  
 var\_35 = var\_1 + var\_37;  
 executeQuery(var\_8);  
 long double \*var\_18 = new long double;  
}

Описание:

В данном тесте присутствуют три уязвимости: две в методе function\_14 и одна в методе main. В первом случае отсутствует высвобождение памяти, выделенной под переменные var\_18 и var\_13. Во втором случае отсутствует высвобождение памяти, выделенной под переменную var\_18. Переменные var\_18 в первом и втором случае различные объекты, так как находятся в разных областях видимости.

Результат работы обработчика уязвимости:

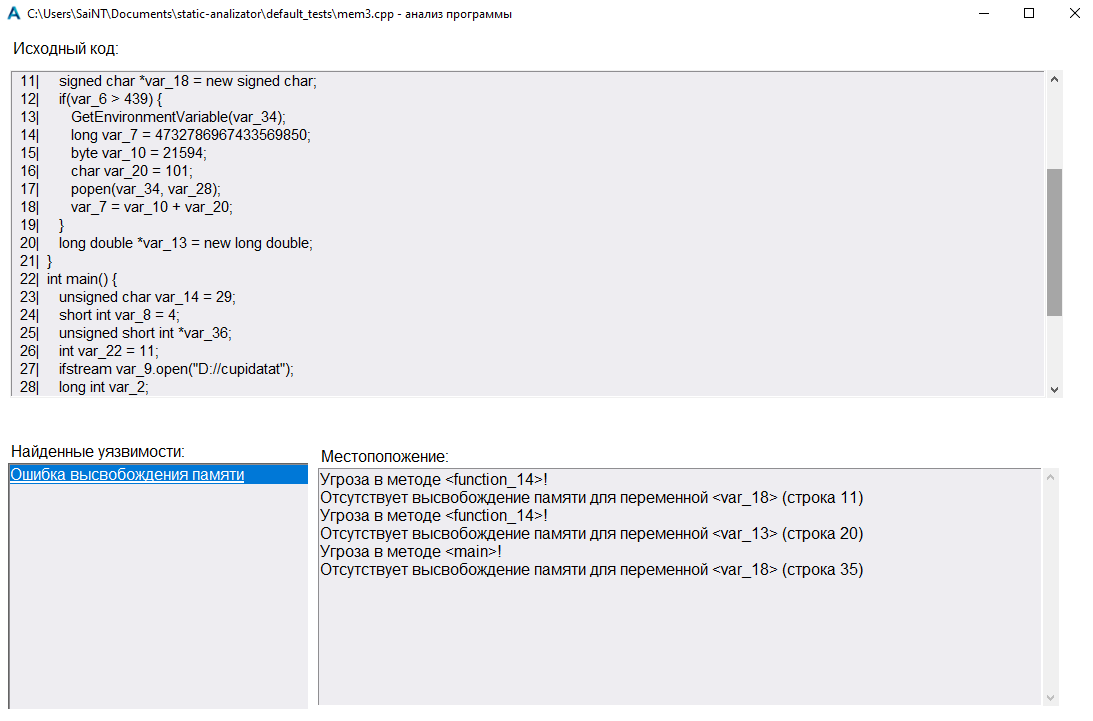


Рисунок 36 – Результат работы обработчика уязвимости над тестом №3

Обработчик успешно обнаружил указанные уязвимости.

Тест №4. Автор: Щекочихин Е.А.

Исходный код:

int function\_11(long long int \*var\_4) {  
 signed long long int var\_23[40];  
 unsigned short int \*var\_24;  
 long double var\_22;  
 unsigned char var\_34;  
 int var\_16;  
 long double \*var\_10;  
 unsigned long int var\_33 = 15;  
 unsigned short int var\_38;  
 cout<<"anim incididunt laboris "<<var\_24;  
 char var\_7 = 124;  
 char var\_13 = 109;  
 long var\_21 = 7821609182552675775;  
 cout<<"adipiscing ad anim "<<var\_34;  
 var\_7 = var\_13 + var\_21;  
 printf(Format %s and %s, var\_21, var\_22);  
}  
int main() {  
 singed long int \*var\_36;  
 signed char var\_20[10];  
 long int var\_1[28];  
 signed int var\_40 = 3;  
 short int \*var\_15;  
 signed char var\_11[2];  
 getenv(var\_19);  
 printf(Format %s and %s, var\_22, var\_9);  
 signed char \*var\_33 = new signed char;  
 wchar\_t \*var\_13 = new wchar\_t;  
}

Описание:

В данном тесте присутствуют две уязвимости: в методе main. В первом случае отсутствует высвобождение памяти, выделенной под переменную var\_33. Во втором случае отсутствует высвобождение памяти, выделенной под переменную var\_13.

Результат работы обработчика уязвимости:

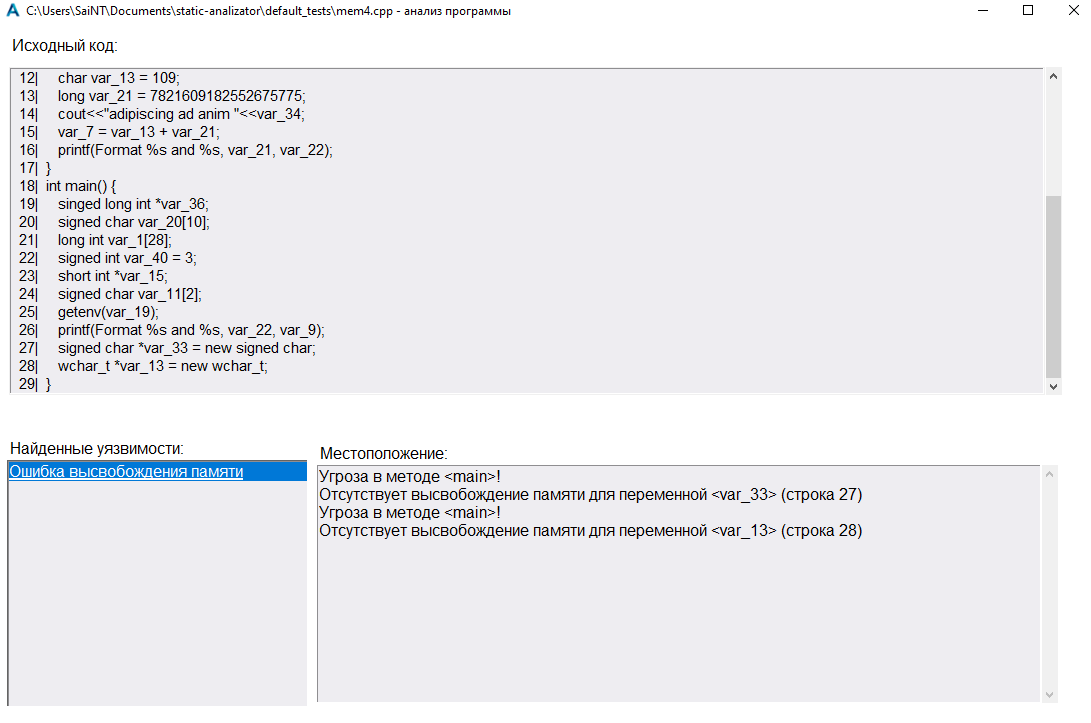


Рисунок 37 – Результат работы обработчика уязвимости над тестом №4

Обработчик успешно обнаружил указанные уязвимости.

1. **Обработчик внедрения команд**

Тест №5. Автор: Терентьев Е.А.

Автор модуля обнаружения уязвимости: Терентьев Е.А.

Исходный код:

#include <stdio.h>  
#include <stdlib.h>  
  
int main(){  
 system("cd c:\\Users\\USER\\Desktop");  
 system("mkdir test");  
 return 0;  
}

Описание:

В данном тесте присутствуют две потенциальные уязвимости, о которых должен предупредить анализатор. Дважды производится вызов опасной функции system.

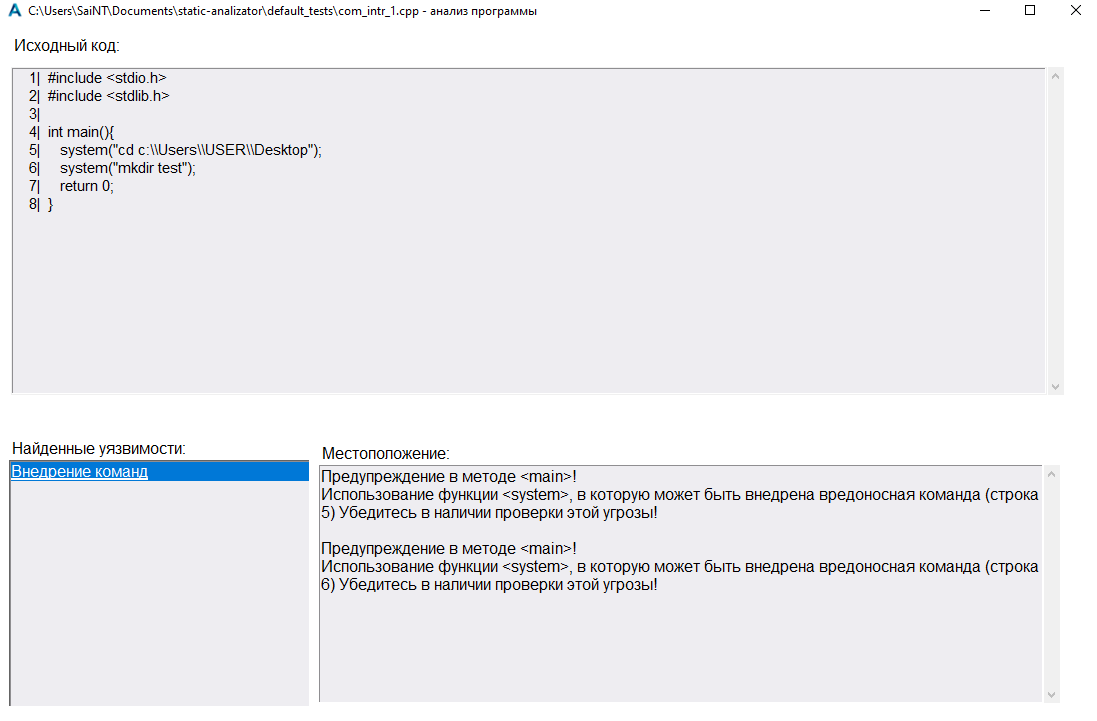
Результат работы обработчика уязвимости:

Рисунок 38 – Результат работы обработчика уязвимости над тестом №5

Обработчик успешно обнаружил указанные уязвимости.

Тест № 6. Автор: Терентьев Е.А.

Исходный код:

#include <stdio.h>  
#include <stdlib.h>  
  
int main(){  
 char\* home=getenv("APPHOME");  
 char\* cmd=(char\*)malloc(strlen(home)+strlen(INITCMD));  
 if (cmd) {  
 strcpy(cmd,home);  
 strcat(cmd,INITCMD);  
 execlp(cmd, NULL);  
 }  
 return 0;  
}

Описание:

В данном тесте присутствует потенциальная уязвимость, о которой должен предупредить анализатор. Производится вызов опасной функции execlp.

Результат работы обработчика уязвимости:

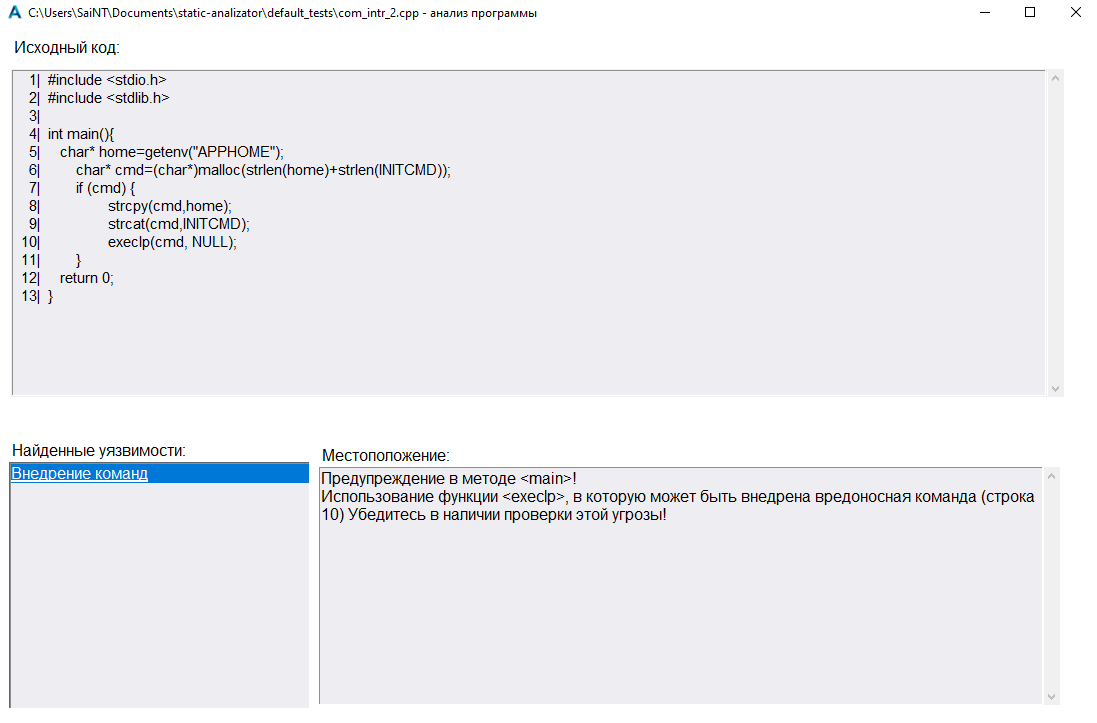


Рисунок 39 – Результат работы обработчика уязвимости над тестом №6

Обработчик успешно обнаружил указанные уязвимости.

Тест №7. Автор: Щекочихин Д.А.

Исходный код:

#include<iostream>  
#include<cv.h>  
  
int main(){  
 system("color 2");  
 system("dir");  
 execvp('tmp/dir', "arg")  
}

Описание:

В данном тесте присутствуют три потенциальные уязвимости, о которых должен предупредить анализатор. Дважды производится вызов опасной функции system а также вызов функции execvp.

Результат работы обработчика уязвимости:

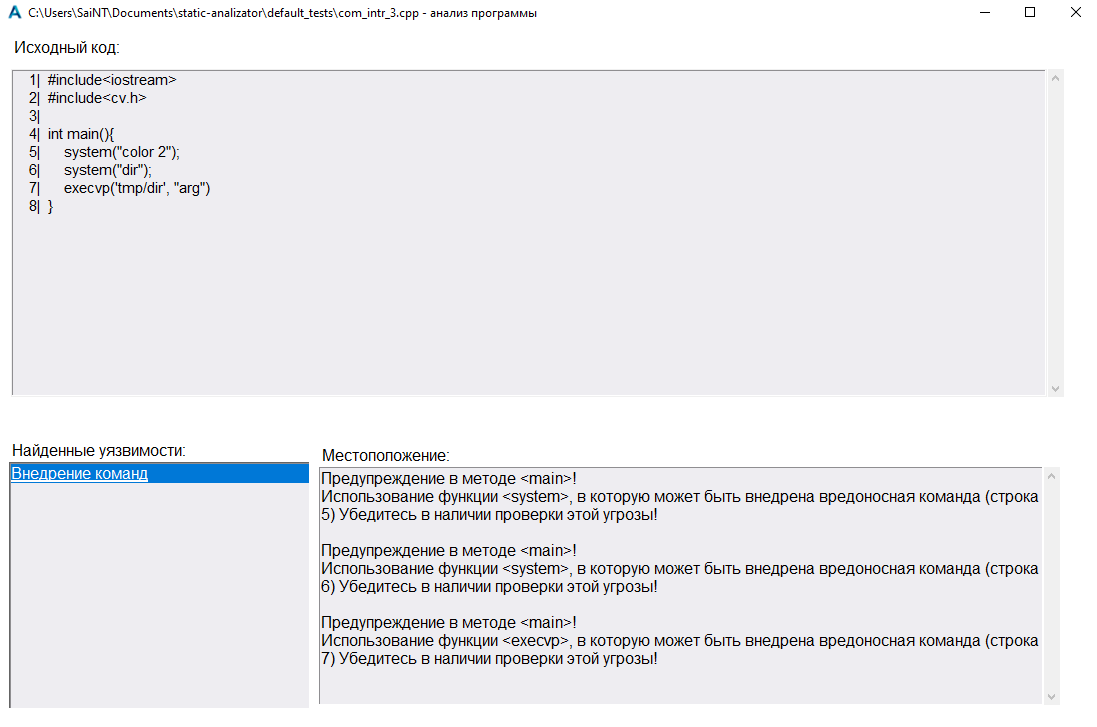


Рисунок 40 – Результат работы обработчика уязвимости над тестом №7

Обработчик успешно обнаружил указанные уязвимости.

Тест №8. Автор: Щекочихин Д.А.

Исходный код:

#include <stdio.h>  
  
int main(){  
 char\* home=getenv("APPHOME");  
 ShellExecute(NULL, \_T("open"), \_T("C:\\Windows\\Sys32\\sigverif.exe"), NULL, NULL, SW\_RESTORE);  
 return 0;  
}

Описание:

В данном тесте присутствует потенциальная уязвимость, о которой должен предупредить анализатор. Производится вызов опасной функции ShellExecute.

Результат работы обработчика уязвимости:

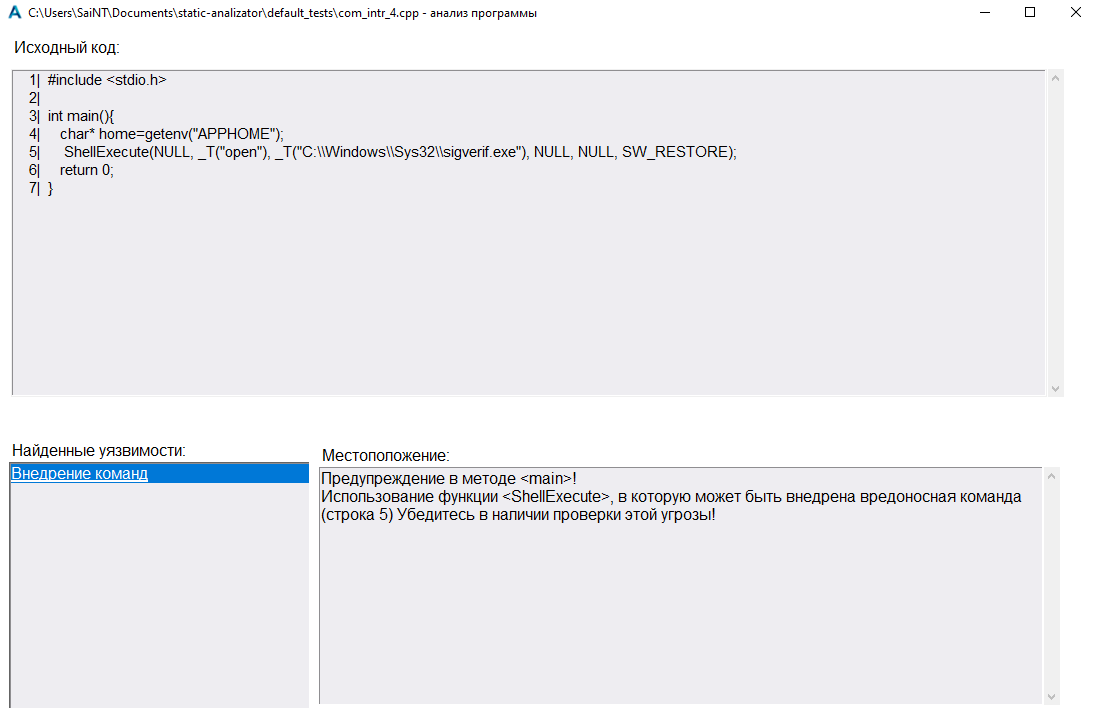


Рисунок 41 – Результат работы обработчика уязвимости над тестом №8

Обработчик успешно обнаружил указанные уязвимости.

1. **Обработчик внедрения SQL команд**

Тест №9. Автор: Терентьев Е.А.

Исходный код:

#include <mysql/mysql.h>   
#include <stdio.h>   
  
int main(){   
  
 MYSQL mysql;   
 MYSQL\_ROW row;   
 MYSQL\_RES \*result;   
   
 unsigned int num\_fields;   
 unsigned int i;   
  
 mysql\_init(&mysql);   
  
 execute("SELECT \* FROM my\_table where id = 'test'");  
 executeQuery("SELECT \* FROM my\_table where id = \'test\'");  
 return 0;  
}

Описание:

В данном тесте существует уязвимость в теле запроса функции exequte. Присутствуют неэкранированные символы одинарных кавычек. В данном коде также присутствует функция exequteQuery, но в теле ее запроса кавычки экранированы, что означает отсутствие уязвимости.

Результат работы обработчика уязвимости:

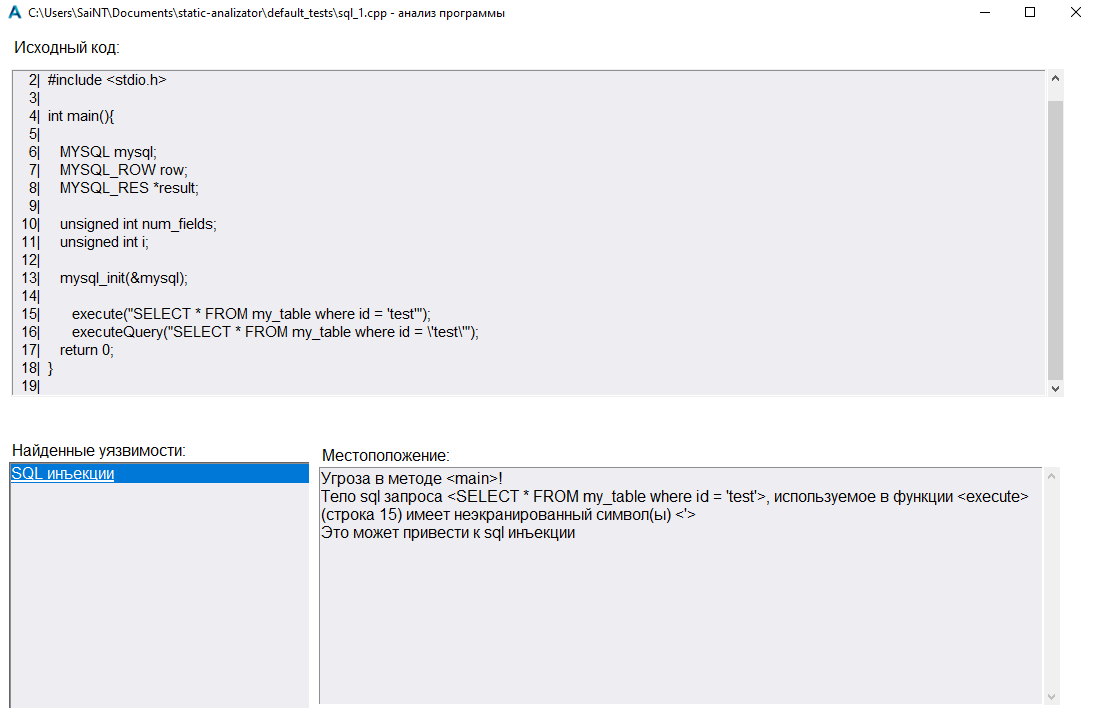


Рисунок 42 – Результат работы обработчика уязвимости над тестом №9

Обработчик успешно обнаружил указанные уязвимости.

Тест №10. Автор: Терентьев Е.А.

Исходный код:

#include <SQL>  
  
int main() {  
 std::string sql;  
 std::cin « sql;  
 executeQuery("select \* from db where id = 'dasd'")  
 auto connect = SQL::connect(“127.0.0.1”. “123”, “123”);  
 SQL::work W(connect);  
 W.execute(sql);  
}

Описание:

В данном тесте присутствуют две уязвимости. В теле запроса функции exequteQuery присутствуют неэкранированные символы одинарных кавычек. Телом запроса функции exequte является строковая переменная sql, значение которой заполняется пользователем. Обработчик должен вывести предупреждение, о возможном наличии уязвимости в данной строке.

Результат работы обработчика уязвимости:

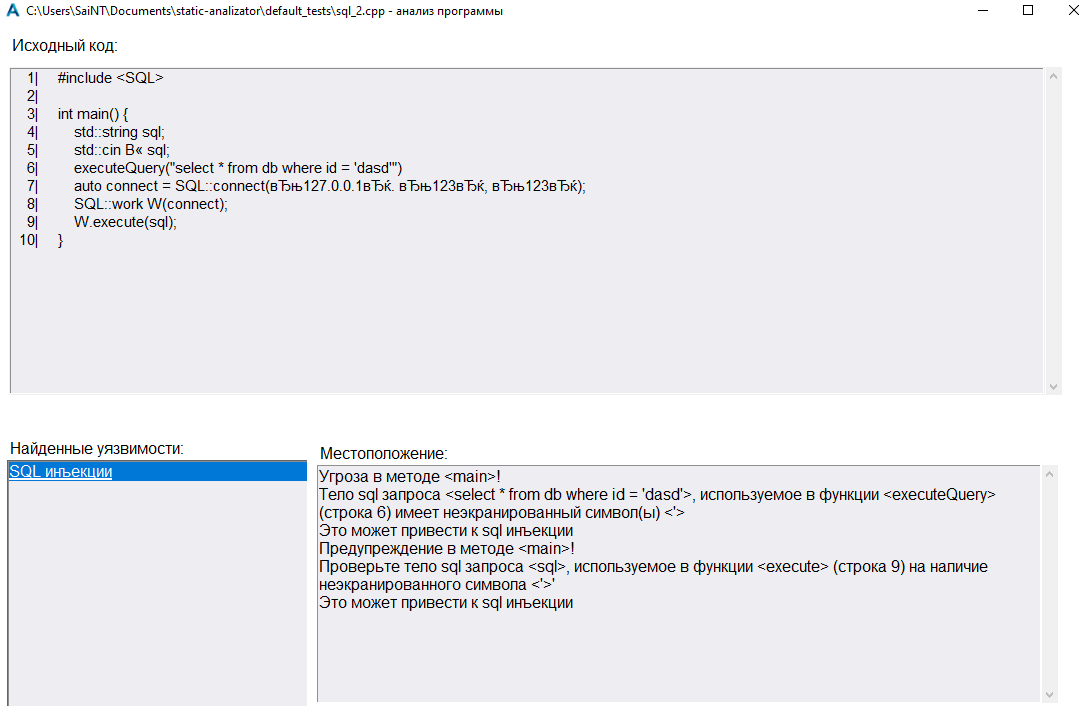


Рисунок 43 – Результат работы обработчика уязвимости над тестом №10

Обработчик успешно обнаружил указанные уязвимости.

Тест №11. Автор: Щекочихин Д.А.

Исходный код:

#include <mysql/mysql.h>  
#include <stdio.h>  
  
 int main(){  
 mysql\_init(&mysql);  
 mysql.execute(smth);  
 if (!mysql\_real\_connect(&mysql,"localhost","root","","MyDatabase",0,NULL,0))  
 {  
 fprintf(stderr, "Failed to connect to database: Error: %s\n",  
 mysql\_error(&mysql));  
 }  
  
 return 0;  
  
 }

Описание:

В данном тесте существует уязвимость в теле запроса функции exequte. Телом запроса функции exequte является строковая переменная smth. Обработчик должен вывести предупреждение, о возможном наличии уязвимости в данной строке.

Результат работы обработчика уязвимости:

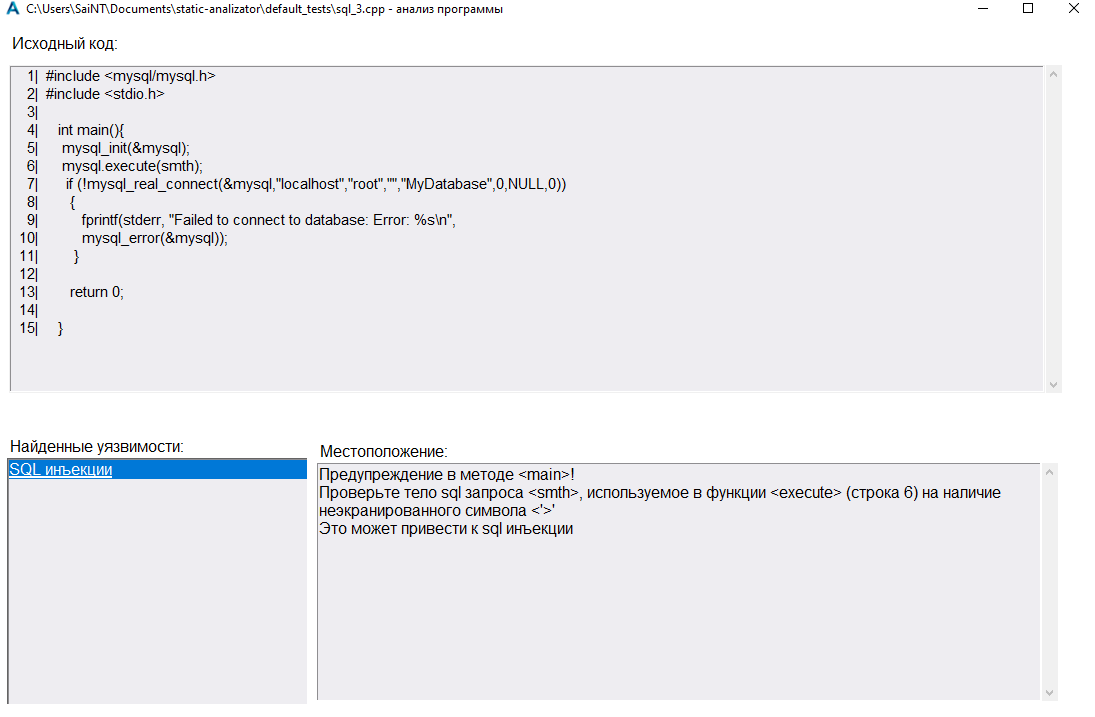


Рисунок 44 – Результат работы обработчика уязвимости над тестом №11

Обработчик успешно обнаружил указанные уязвимости.

Тест №12. Автор: Щекочихин Д.А.

Исходный код:

#include <mysql/mysql.h>  
#include <stdio.h>  
  
 int main(){  
 mysql\_init(&mysql);  
 mysql.executeQuery("smth'");  
 if (!mysql\_real\_connect(&mysql,"localhost","root","","MyDatabase",0,NULL,0))  
 {  
 fprintf(stderr, "Failed to connect to database: Error: %s\n",  
 mysql\_error(&mysql));  
 }  
  
 return 0;  
  
 }

Описание:

В данном тесте существует уязвимость в теле запроса функции exequteQuery. Присутствуют неэкранированные символы одинарных кавычек, что может привести к SQL-инъекции.

Результат работы обработчика уязвимости:

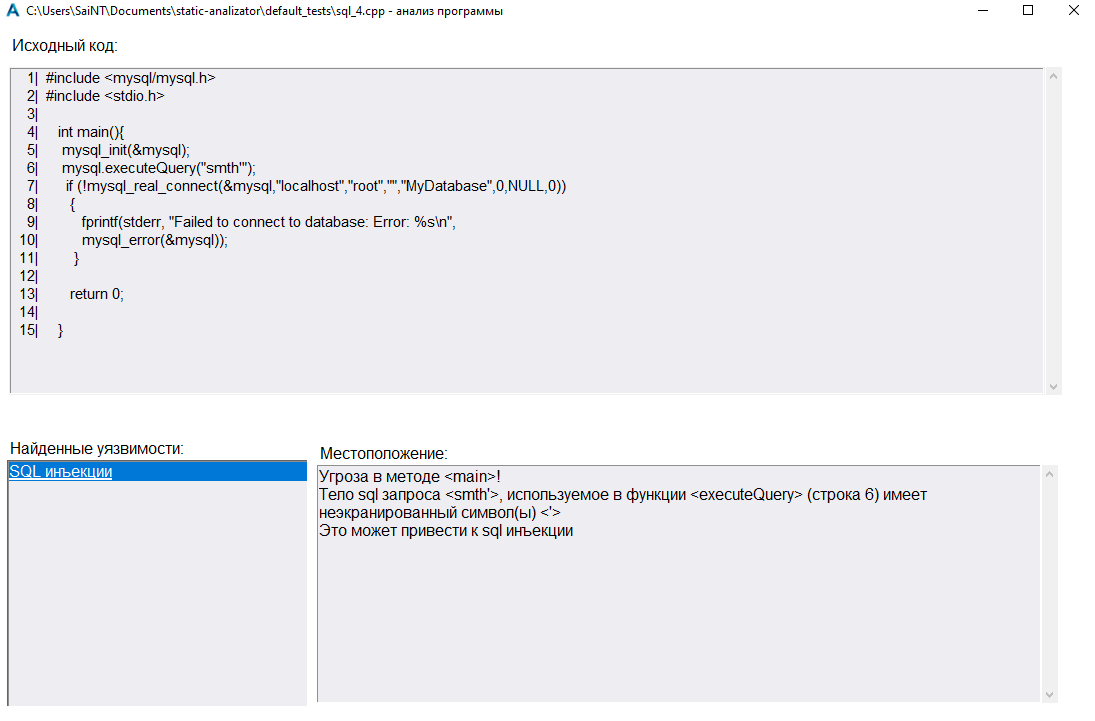


Рисунок 45 – Результат работы обработчика уязвимости над тестом №12

Обработчик успешно обнаружил указанные уязвимости.

ЗАКЛЮЧЕНИЕ

В процессе выполнения курсового проекта мы проанализировали раннее разработанные анализаторы кода и их методы реализации, на основе полученных знаний сами реализовали статический анализатор кода, написанного на языке «С++», написали тесты, а также генератор случайного кода С++ для проверки работоспособности нашей программы.

Статический анализатор кода проверяет код на наличие уязвимостей информационной безопасности, согласованных на этапе проектирования.

В течение выполнения проекта мы запрограммировали разработанные алгоритмы по поиску уязвимостей и провели тестирование на тестах, написанных членами бригады, и тестах, сгенерированных автоматическим генератором.

Также на протяжении разработки статического анализатора, мы проводили отладку, устраняли ошибки и недоработки.

В результате, мы изучили все согласованные на этапе проектирования уязвимости информационной безопасности, которые могут привести к непоправимым ошибкам при разработке приложения на языке С++, а также методы их определения, реализовали статический анализатор кода С++ и получили бесценный опыт работы в команде.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1 Регулярные выражения / Джеффри Фридл, под ред. А. Галунова, Н. Макаровой, Б. Попова, Ю. Бочиной. - 3-е изд. - СПб.: Символ-Плюс, 2008. - 608 с.

2 Ховард М., Лебланк Д., Виега Д. Как написать безопасный код на С++, Java, Perl, PHP, ASP.NET. - Москва: ДМК Пресс, 2014. - 288 с.

3 Архипова М.В. Генерация тестов для семантических анализаторов // Вычислительные методы и программирование. - 2006. - №3. - С. 55-70.

4 Статический анализ кода C++ // habr.com URL: https://habr.com/ru/post/75123/ (дата обращения: 06.04.2020).

5 Энтони Уильямс Параллельное программирование на С++ в действии. Практика разработки многопоточных программ. - Москва: ДМК Пресс, 2012. - 672 с.

6 Brian Chess, Jacob West Secure Programming with Static Analysis. Addison-Wesley Professional, 2007. - 624 с.

7 Регулярные выражения в Python от простого к сложному. Подробности, примеры, картинки, упражнения // habr.com URL: https://habr.com/ru/post/349860/ (дата обращения: 15.04.2020).

8 М. И. Глухих, В. М. Ицыксон Программная инженерия. Обеспечение качества программных средств методами статистического анализа: учебное пособие. - СПб: Изд-во Политехнического ун-та, 2011. - 149 с.

9 Ф.М. Пучков Методы и средства автоматизированного обнаружения уязвимостей в программах на языке C на основе статического анализа их исходных текстов: автореф. дис. ... канд. физ.-мат. наук: 05.13.19. - Москва, 2010. - 24 с.

10 Златопольский Д. М. Основы программирования на языке Python. - 1 изд. - Москва: ДМК Пресс, 2017. - 284 с.

ПРИЛОЖЕНИЕ А

Статический анализатор кода С++ (рис. 2) состоит из 5 основных модулей.

1. Модуль первичного лексического анализа кода предназначен для разбиения анализируемого кода на составляющие части и построения структуры анализируемого кода. Это позволяет передавать важную информацию об анализируемом коде обработчикам уязвимостей.
2. Модуль обработчиков уязвимостей содержит в себе 13 python-модулей, каждый из которых отвечает за обработку отдельной уязвимости.

Список модулей:

* Обработчик переполнения буфера;
* Обработчик пренебрежения обработок ошибок;
* Обработчик проблем, связанных с динамическим выделением памяти;
* Обработчик внедрения SQL команд;
* Обработчик внедрения команд;
* Обработчик состояния гонки;
* Обработчик ошибки форматной строки;
* Обработчик некорректного доступа к файлам;
* Обработчик задачи Читатели – Писатели;
* Обработчик случайных чисел криптографического характера;
* Обработчик переполнения целых чисел;
* Обработчик пренебрежения безопасным хранением данных;
* Обработчик утечки информации.

1. Агрегирующий модуль отвечает за агрегацию и управление всеми используемыемыми модулями программы.

4) Пользовательский интерфейс отвечает за графическую оболочку нашей программы, принимает и отображает в удобном, читаемом виде информацию, полученную в ходе анализа.

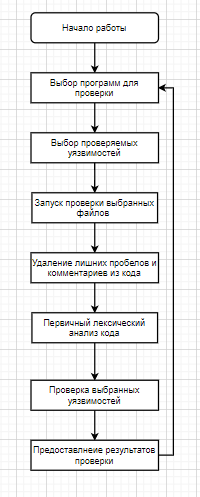
5) Модуль тестирования и генерации тестового кода С++ предназначен для тестирования и отладки разработанного анализатора.

Рисунок 1 – Временная диаграмма статического анализатора

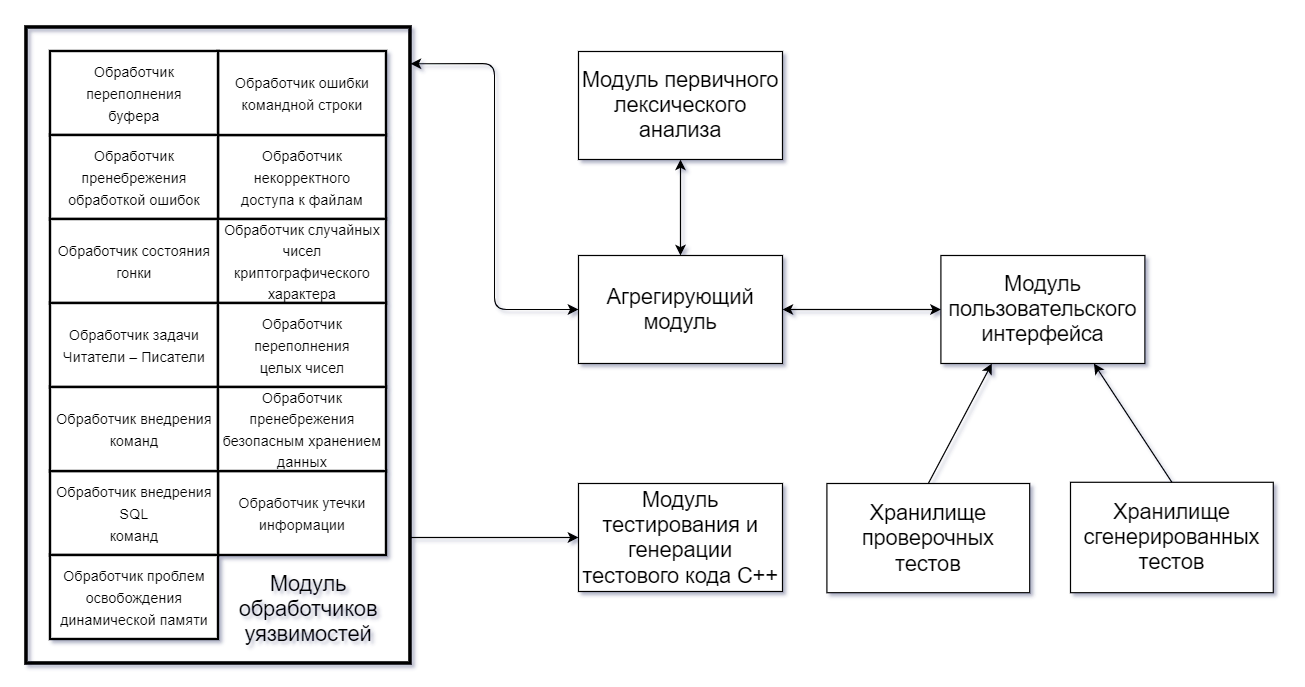
Библиотекой уязвимостей является модуль обработчиков уязвимостей. Модуль представляет собой набор классов обработчиков уязвимостей, описанных в таблице 2, пункта 2.4.2. При необходимости ее можно дополнить новыми обработчиками, добавив собственный класс обработчика, который будет подчиняться общим правилам создания обработчиков. На структурной схеме (рисунок 2) также можно увидеть два объекта: «Хранилище проверочных тестов» и «Хранилище сгенерированных тестов», это директории, в которых хранятся соответственно проверочные и сгенерированные тесты.

Рисунок 2 - Структурная схема статического анализатора

Таблица 1 - распределения частей проекта среди участников бригады.

|  |  |  |
| --- | --- | --- |
| Часть проекта и должность | Задачи и наименования. | Ответственный |
| Часть №1 - Бригадир | 1) Разработка структуры и общего алгоритма.  2) Разработка обработчика переполнения буфера.  3) Разработка обработчика пренебрежения обработкой ошибок.  4) Разработка обработчика состояния гонки.  5) Разработка обработчика задачи Читатели – Писатели.  6) Разработка модуля первичного лексического анализа кода.  7) Разработка агрегирующего модуля.  8) Разработка модуля тестирования и генерации кода.  9) Распределение работ. | Ковалев К.А. |
| Часть №2 – Зам бригадира | 1) Разработка общего алгоритма.  2) Разработка обработчика внедрения команд.  3) Разработка обработчика внедрения SQL команд.  4) Разработка обработчика проблем, связанных с динамическим выделением памяти.  5) Разработка модуля тестирования и генерации кода. | Терентьев Е.А. |
| Часть №3 – Разработчик программы | 1) Разработка обработчика ошибки командной строки.  2) Разработка обработчика некорректного доступа к файлам.  3) Разработка графического интерфейса.  4) Разработка модуля тестирования и генерации кода. | Щекочихин Д.А. |
| Часть №4 – Разработчик программы | 1) Разработка обработчика случайных чисел криптографического характера.  2) Разработка обработчика переполнения целых чисел.  3) Разработка модуля тестирования и генерации кода.  4) Разработка инструкций пользователя и другой ЭТД. | Оганисян Т.С. |
| Часть №5 – Разработчик программы | 1) Разработка обработчика пренебрежения безопасным хранением данных программы.  2) Разработка обработчика утечки информации.  3) Разработка модуля тестирования и генерации кода.  4) Разработка перечня тестов, перечня внештатных ситуаций. | Шабакаев А.Р. |

ПРИЛОЖЕНИЕ Б

**Обработчик проблем, связанных с динамическим выделением памяти.**

Автор: Терентьев Е.А.

from typing import List

from core.base\_handler import BaseHandler

from core.function\_context import FunctionContext

from core.variable import \*

class MemoryLeakHandler(BaseHandler):

vulnerability\_name = 'Ошибка высвобождения памяти'

def \_\_init\_\_(self):

self.pattern = r'^\s\*delete(\(.\*\))'

self.output = []

def parse(self, contexts: List[FunctionContext]):

total\_errors = 0

for context in contexts:

m\_vars = []

for var in context.variables:

if var.value is not None:

if "new" in var.full\_declaration:

m\_vars.append(var)

free\_matches = []

for \_, line in context.source\_code.items():

matches = re.finditer(self.pattern, line, re.IGNORECASE)

free\_matches += matches

for var in m\_vars:

if var.var\_name not in list(map(lambda x: x.group(1)[1:-1].strip(), free\_matches)):

total\_errors += 1

self.output.append(f"{total\_errors}) Угроза в методе <{context.name}>!\n"

f"Отсутствует высвобождение памяти для переменной <{var.var\_name}> (строка {var.line\_appeared})")

self.output.append(self.vulnerability\_name + ": " + str(total\_errors))

return self.output

**Обработчик внедрения SQL команд.**

Автор: Терентьев Е.А.

import re

from typing import List

from core.base\_handler import BaseHandler

from core.function\_context import FunctionContext

class SQLInjectionHandler(BaseHandler):

vulnerability\_name = 'SQL инъекции'

def \_\_init\_\_(self):

self.pattern = r'(executeQuery|execute)(\(.\*\))'

self.output = []

def parse(self, contexts: List[FunctionContext]):

total\_errors = 0

for context in contexts:

for line\_number, line in context.source\_code.items():

matches = re.finditer(self.pattern, line)

for match in matches:

tmp = match.group(2)[1:-1]

if tmp[0] == '\"': # check if parameter is literally string like "test string"

if re.search(r'[^\\]\'', tmp) is not None:

total\_errors += 1

self.output.append(f"{total\_errors}) Угроза в методе <{context.name}>!\n"

f"Тело sql запроса <{tmp[1:-1]}>, используемое в функции <{match.group(1)}> (строка {line\_number}) "

f"имеет неэкранированный символ(ы) <'>\nЭто может привести к sql инъекции")

else:

total\_errors += 1

self.output.append(f"{total\_errors}) Предупреждение в методе <{context.name}>!\n"

f"Проверьте тело sql запроса <{tmp}>, используемое в функции <{match.group(1)}> (строка {line\_number}) "

f"на наличие неэкранированного символа <'>'\nЭто может привести к sql инъекции")

self.output.append(self.vulnerability\_name + ": " + str(total\_errors))

return self.output

**Обработчик внедрения команд.**

Автор: Терентьев Е.А.

import re

from typing import List

from core.base\_handler import BaseHandler

from core.function\_context import FunctionContext

class CommandsIntroductionHandler(BaseHandler):

vulnerability\_name = 'Внедрение команд'

"""

- int system (const char\* command); - выполняет системную команду.

- FILE \*popen(const char \*command, const char \*type); - создает дочерний процесс, которому можно передать команду.

- int execlp(const char \*file, const char \*arg, ...); - выполнение исполняемого файла по пути <file> с передачей

параметра в <arg>.

- int execvp(const char \*file, char \*const argv[]); - тоже что и execlp, но параметры передаются в <argv>.

- HINSTANCE ShellExecute(HWND hwnd, LPCSTR lpOperation, LPCSTR lpFile, LPCSTR lpParameters, LPCSTR lpDirectory, INT nShowCmd); -

выполняет команду <lpOperation> над указанным файлом <lpFile>.

"""

def \_\_init\_\_(self):

self.pattern = r'(system|popen|execlp|execvp|ShellExecute)'

self.output = []

def parse(self, contexts: List[FunctionContext]):

total\_errors = 0

for context in contexts:

for line\_number, line in context.source\_code.items():

matches = re.finditer(self.pattern, line)

for match in matches:

total\_errors += 1

self.output.append(f"{total\_errors}) Предупреждение в методе <{context.name}>!\n"

f"Использование функции <{match.group(1)}>, в которую может быть внедрена"

f" вредоносная команда (строка {line\_number}) "

f"Убедитесь в наличии проверки этой угрозы!\n")

self.output.append(self.vulnerability\_name + ": " + str(total\_errors))

return self.output

**Функции генерации тестового кода С++.**

Автор: Терентьев Е.А.

# генерирует ошибку встраивания команд  
 def c\_intr\_error(self, indent, params):  
 funcs = [  
 {"system(": 1},  
 {"popen(": 2},  
 {"execlp(": 3},  
 {"execvp(": 2},  
 {"ShellExecute(": 6},  
 ]  
 cur\_func\_dict = rand\_value(funcs)  
 cur\_func = list(cur\_func\_dict.keys())[0]  
 n = list(cur\_func\_dict.values())[0]  
 chosen\_param\_list = [params[param][0] for param in gen\_n\_rands(n, 0, len(params) - 1)]  
 code = indent + cur\_func + ", ".join(chosen\_param\_list) + ");\n"  
 for i in self.gen\_report\_map:  
 if "Внедрение команд" in i[0]:  
 i[0] += f"\nСтрока {self.line\_number}\n"  
 i[1] += 1  
 self.line\_number += 1  
 return code  
  
 # генерирует ошибку высвобождения памяти  
 def free\_error(self, indent, params):  
 var\_type = rand\_value(constants.types)  
 var\_name = "var\_" + str(rng(1, 2000))  
 while var\_name in params:  
 var\_name = "var\_" + str(rng(1, max\_val))  
 code = indent + var\_type + " \*" + var\_name + " = new " + var\_type + ";\n"  
 params.append((var\_name, code))  
  
 for i in self.gen\_report\_map:  
 if "Ошибка высвобождения памяти" in i[0]:  
 i[0] += f"\nСтрока {self.line\_number}\n"  
 i[1] += 1  
 self.line\_number += 1  
 return code  
  
 # генерирует ошибку внедрения SQL-инъекций  
 def sql\_error(self, indent, params):  
 funcs = ["execute(", "executeQuery("]  
 cur\_func = rand\_value(funcs)  
 sample\_text = ""  
 for \_ in range(3):  
 sample\_text += rand\_value(sample\_words) + "'"  
 random = rng(0, 1)  
 if random == 0:  
 code = indent + cur\_func + "\"" + sample\_text + "\");\n"  
 else:  
 code = indent + cur\_func + rand\_value(params)[0] + ");\n"  
 for i in self.gen\_report\_map:  
 if "SQL инъекции" in i[0]:  
 i[0] += f"\nСтрока {self.line\_number}\n"  
 i[1] += 1  
 self.line\_number += 1  
 return code