МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Факультет прикладной математики и информатики

Кафедра технологий программирования

ПАШКО
Александра Алексеевна

СТАТИЧЕСКИЙ АНАЛИЗ ПРОГРАММНОГО КОДА ДЛЯ ПРЕДОТВРАЩЕНИЯ ЕГО НЕПРЕДВИДЕННОГО ПОВЕДЕНИЯ

Дипломная работа

	Научный руководитель: доцент кафедры технологий программирования И.С. Войтешенко

«__» _____ 2019 г.

Зав. кафедрой технологий программирования
доктор технических наук, профессор А.Н. Курбацкий

Минск, 2019

Аннотация

Пашко А.А. Статический анализ программного кода для предотвращения его непредвиденного поведения / Минск: БГУ, 2019.

В работе изучаются основные понятия, связанные со статическим анализом программного кода, задачи статического анализа, демонстрируются возможности статического анализатора PVS-Studio. Были исследованы проблемы безопасности, которые можно решить с помощью статического анализа и создано приложение, которое выявляет несколько типов уязвимостей по принципу статического анализатора.

Анатацыя

Пашко А.А. Статычны аналіз праграмнага кода для прадухілення яго неспадзяваных паводзін / Мінск: БДУ, 2019.

У працы вывучаюцца асноўныя паняцці, звязаныя са статычным аналізам праграмнага кода, задачы статычнага аналізу, дэманструюцца магчымасці статычнага аналізатара PVS-Studio. Былі даследаваны праблемы бяспекі, якія можна вырашыць з дапамогай статычнага аналізу і створан дадатак, які выяўляе некалькі тыпаў уразлівасцяў па прынцыпе статычнага аналізатара.

Annotation

Pashko A.A. Static analysis of software code to prevent its unexpected behavior / Minsk: BSU, 2019.

The work deals with the main points of the basic concepts associated with static code analysis, goals of static analysis, also capabilities of the PVS-Studio static analyzer are demonstrated. Was made an investigation, which security problems can be solved with static analysis and created an application that detects several types of vulnerabilities, using techniques of the static analyzer.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ 7
ГЛАВА 1 ВВЕДЕНИЕ В СТАТИЧЕСКИЙ АНАЛИЗ 8
1.1 Понятие и принципы работы статического анализа 8
1.2 Преимущества и ограничения статического анализа 11
1.3 Основные задачи статического анализа 12
1.4 Выводы 13
ГЛАВА 2 МЕТОДЫ СТАТИЧЕСКОГО АНАЛИЗА 14
2.1 Анализ потока данных (Data-Flow Analysis) 14
2.2 Анализ потока управления (Control-Flow Analysis) 15
2.3 Символьное выполнение (Symbolic Execution) 15
2.4 Аннотирование методов (Method Annotations) 17
2.5 Сопоставление с шаблоном (Pattern-based analysis) 18
2.6 Выводы 19
ГЛАВА 3 КЛАССИФИКАЦИЯ УЯЗВИМОСТЕЙ ЗАЩИТЫ 20
3.1 Основные виды уязвимостей 20
3.2 Ошибки обработки ввода данных 21
3.3 Переполнение буфера 23
3.4 Ошибки форматирования строк 24
3.5 Выводы 25
ГЛАВА 4 ИСПОЛЬЗОВАНИЕ СТАТИЧЕСКОГО АНАЛИЗА В РАЗРАБОТКЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ДЛЯ ВСТРОЕННЫХ СИСТЕМ 27
4.1. Роль статического анализа при разработке для встроенных систем 27
4.2. Основные виды анализа программного кода для встроенных систем 28
4.3. Сравнение существующих статических анализаторов для встроенных систем 30
4.4 Выводы 32
ГЛАВА 5 РЕАЛИЗАЦИЯ ИНСТРУМЕНТА СТАТИЧЕСКОГО АНАЛИЗА 33
5.1 Постановка задачи и принцип работы приложения 33
5.2 Алгоритмы поиска статическим анализатором уязвимостей в исходном коде 34
5.3 Выводы 39
ЗАКЛЮЧЕНИЕ 40
СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ 41
ПРИЛОЖЕНИЕ А 43
ПРИЛОЖЕНИЕ Б 47

РЕФЕРАТ

Дипломная работа, 49 страниц, 22 рисунка, 16 источников, 2 приложения.

Статический анализ программного кода для предотвращения его непредвиденного поведения

Ключевые слова: СТАТИЧЕСКИЙ АНАЛИЗ, ЛЕКСИЧЕСКИЙ РАЗБОР, СИНТАКСИЧЕСКИЙ РАЗБОР, УЯЗВИМОСТЬ, ОШИБКИ ВВОДА, ОШИБКИ ФОРМАТИРОВАНИЯ СТРОК, ПЕРЕПОЛНЕНИЕ БУФЕРА, ВСТРОЕННЫЕ СИСТЕМЫ

Объект исследования — статический анализатор PVS-Studio, сфера встроенных систем, статический анализ как средство повышения качества кода и поиска уязвимостей в программном коде.

Цель работы — изучить понятие статического анализа и уязвимости, которые можно обнаружить при его использовании, изучить возможности статического анализатора на примере PVS-Studio и создать собственный инструмент поиска уязвимостей в программном коде.

Результаты работы — реализовано приложение поиска различных видов уязвимостей программного кода на языках Си и C++, изучены методы статического анализа исходного кода программ.

Область применения результатов — проект, написанный на языках C и C++ для оптимизации и поиска ошибок, которые могут повлиять на корректную и безопасную работу приложения.

РЭФЕРАТ

Дыпломная праца, 49 старонак, 22 малюнка, 16 крыніц, 2 дадатка.

Статычны аналіз праграмнага кода для прадухілення яго неспадзяваных паводзін

Ключавыя словы: СТАТЫЧНЫ АНАЛІЗ, ЛЕКСІЧНЫ РАЗБОР, СІНТАКСІЧНЫ РАЗБОР, ЎРАЗЛІВАСЦЬ, ПАМЫЛКІ ЎВОДУ, ПАМЫЛКІ ФАРМАТАВАННЯ РАДКОЎ, ПЕРАПАЎНЕННЕ БУФЕРА, УБУДАВАНЫЯ СІСТЭМЫ

Аб'ект даследвання — астатический аналізатар PVS-Studio, сфера ўбудаваных сістэм, статычны аналіз як сродак павышэння якасці кода і пошуку ўразлівасцяў у праграмным кодзе.

Мэты працы — вывучыць паняцце статычнага аналізу і ўразлівасці, якія можна выявіць пры яго выкарыстанні, вывучыць магчымасці статычнага аналізатара на прыкладзе PVS-Studio і стварыць уласны інструмент пошуку ўразлівасцяў у праграмным кодзе.

Вынік працы — рэалізаваны дадатак пошуку розных відаў уразлівасцяў праграмнага кода на мовах Си і С ++, вывучаны метады статычнага аналізу зыходнага кода праграм.

Вобласць выкарыстоўвання — праект, напісаны на мовах С і С ++ для аптымізацыі і пошуку памылак, якія могуць паўплываць на карэктную і бяспечную работу дадатку.

ABSTRACT

Diploma thesis, 49 pages, 22 figures, 16 sources, 2 appendices.

Static analysis of software code to prevent its unexpected behavior

Keywords: STATIC ANALYSIS, LEXICAL PARSING, PARSING, VULNERABILITY, INPUT ERRORS, STRING FORMATTING ERRORS, BUFFER OVERFLOW, EMBEDDED SYSTEMS.

Object of research — static analyzer PVS-Studio, the scope of embedded systems, static analysis as a means of improving the quality of the code and searching for vulnerabilities in the program code.

Purpose — study the concept of static analysis and vulnerabilities that can be detected when using it, examine the capabilities of the static analyzer using the example of PVS-Studio and create your own tool for program code vulnerability scanning.

Result — implemented an application to search for various types of software code vulnerabilities in C and C ++ languages; methods of static analysis of programs source code were studied.

Application area — a project written in C and C ++ languages for optimizing and searching for errors that may affect the correct and safe operation of the application.

ВВЕДЕНИЕ

Программное обеспечение используется повсюду: для автоматизации процессов на предприятиях, оптимизации торговли и передачи информации между людьми. Однако без соответствующего обеспечения безопасности корректное функционирование данных систем невозможно. Стоит обратить внимание на то, что большая часть деятельности, которая происходит под видом компьютерной безопасности, вовсе не связана с решением проблем безопасности. Скорее с тем, как навести порядок там, где эти проблемы уже существуют. Антивирусные программы, системы исправления ошибок и системы обнаружения вторжений - все это средства, с помощью которых восполняются недостатки в безопасности программного обеспечения. Соответственно, больше усилий прилагается для компенсации плохой безопасности, вместо того, чтобы изначально создавать безопасное программное обеспечение.

Хотя наиболее популярные и опасные для безопасности программы уязвимости давно известны, их количество с каждым годом не уменьшается, а программисты, как правило, повторяют одни и те же ошибки. Удобным средством для их предотвращения и устранения и является статический анализ как метод поиска общих ошибок безопасности в исходном коде. Термин статический анализ относится к любому процессу оценки кода без его выполнения, то есть он позволяет рассмотреть большое количество различных сценариев работы программы, в том числе в труднодоступных состояниях, которые могут быть сложными для осуществления и тестирования.

Статический анализ необходимо использовать так же, когда речь идет о разработке встроенных систем для использования в аэрокосмической, автомобильной, медицинской, ядерной, железнодорожной и других отраслях. Цена ошибки в программном обеспечении в этих областях может быть очень высока, поэтому разработчики, которые пишут код для встроенных систем должны придерживаться стандартов производительности и безопасности, которые превосходят стандарты большинства других отраслей.

Ни один инструмент или техника никогда не обеспечит решение всех проблем безопасности, но с помощью грамотного использования средств статического анализа можно исправлять ошибки безопасности на раннем этапе, минимизировать количество ошибок в программе, повысить ее отказоустойчивость.

ГЛАВА 1
ВВЕДЕНИЕ В СТАТИЧЕСКИЙ АНАЛИЗ

1.1 Понятие и принципы работы статического анализа

Статический анализ кода — это процесс выявления ошибок в исходном коде программы до ее выполнения.

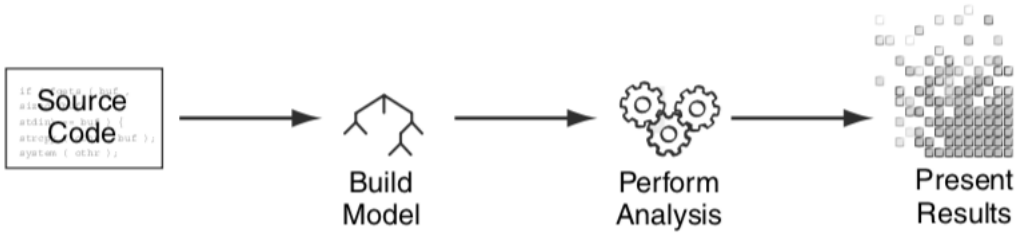


Рисунок 1.1 - Общая структура обработки данных при статическом анализе

Общую структуру обработки данных при статическом анализе можно увидеть на рисунке 1.1. Часто встречается рассуждение о том, что динамический анализ кода или тесты на проникновение могут заменить статический анализ, поскольку эти методы проверки выявят реальные проблемы и ложных срабатываний не будет. Однако стоит обратить внимание на то, что динамический анализ, в отличие от статического, не проверяет весь код, а проверяет только устойчивость программного обеспечения к набору атак, которые имитируют действия злоумышленника. Злоумышленник же может оказаться изобретательнее проверяющего вне зависимости от того, кто выполняет проверку: человек или машина.

Динамический анализ будет полным только в том случае, если выполняется на полном тестовом покрытии, что в применении к реальным приложениям — трудновыполнимая задача. Доказательство полноты тестового покрытия — алгоритмически неразрешимая задача [7].

Почти все статические анализаторы так или иначе построены по принципу компиляторов, то есть в их работе присутствуют этапы преобразования исходного кода — такие же, какие выполняет компилятор.

Можно выделить следующие шаги в работе статического анализатора:

1. Разбор текста

1.1. Лексический анализ (текст программы считывается строка за строкой, разбивается на лексемы: зарезервированные слова, идентификаторы и константы)

- выделение лексем;
- определение, какой группе лексем принадлежит данная лексема;
- передача результатов дальше, чтобы понять, правильно ли составлены предложения;

1.2. Синтаксический анализ (определяется, правильно ли составлены предложения из слов)

- проверка на наличие синхронизирующих лексем: текст до синхронизирующей лексемы считается правильным;
- локальная коррекция (происходит автоматическое добавление синхронизирующих лексем);
- расширение грамматики ошибками (исходная грамматика дополняется неправильными выражениями и затем в нужный момент предлагается вариант исправления ошибки);

1.3. Контекстный анализ (проверяется, допустимо ли использование слов в данном контексте)

2. Применение некоторых правил

Изложенный далее текст следует лекции [9].

Как правило, лексический и синтаксический анализаторы строятся на основе формализованного описания лексики и синтаксиса языка. Лексика языка (правила выделения лексем) может быть описана в виде регулярных выражений или грамматик, синтаксис всегда описывается с помощью формальных грамматик (или их графического изображения в виде схем).

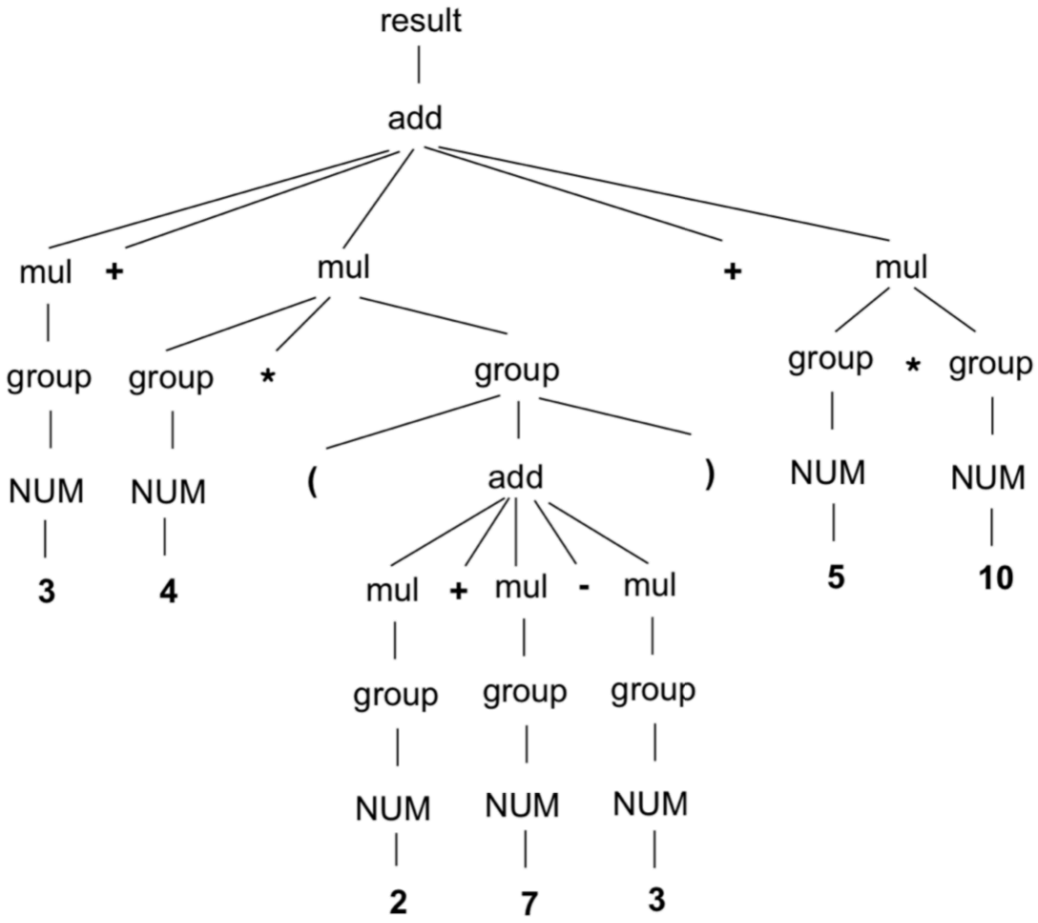
В общем случае грамматики состоят из правил преобразования последовательностей символов в другие последовательности.

Грамматики одинаково помогают решать задачи как программистов, использующих язык, так и создателей компиляторов/анализаторов для данного языка:

- грамматика предоставляет точную и достаточно легкую для понимания синтаксическую спецификацию языка программирования;
- для некоторых классов грамматик можно автоматически сконструировать эффективный анализатор, который определяет, является ли исходная программа синтаксически правильной;

• компиляторы, разработанные на базе грамматик, могут быть достаточно легко расширены (это особенно полезно для добавления новых конструкций, появившихся в результате развития языка);

Собственно, порядок применения правил грамматики для разбора входных данных будет называться деревом разбора (рисунок 1.2).



Дерево разбора, как результат синтаксического анализа, удобно представлять в виде абстрактного синтаксического дерева – AST (рисунок 1.3).

Рисунок 1.2 - Пример дерева разбора арифметического выражения «3+4*(2+7-3)+5*10»

Это позволяет оптимизировать некоторые участки на более позднем этапе генерации объектного кода, так же язык может иметь очень много ключевых слов, однако AST показывает только значимые компоненты, отражающие смысл программы, поэтому оно лишено семантических подробностей, мешающих отображению сути программы.

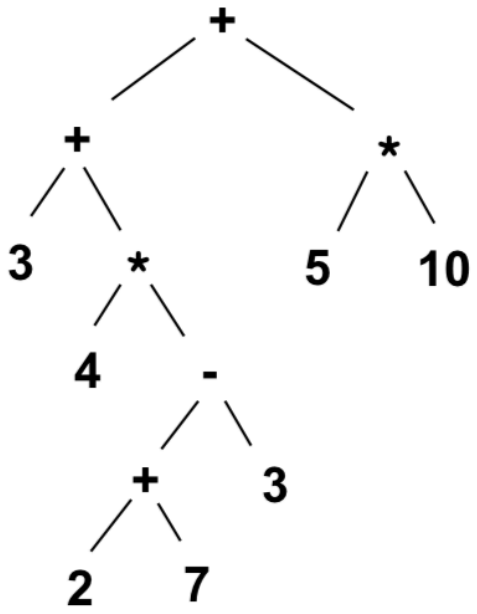


Рисунок 1.3 - Пример AST-дерева арифметического выражения «3+4*(2+7-3)+5*10»

В качестве узлов в AST-дереве выступают операторы, к которым присоединяются их аргументы, которые в свою очередь также могут быть составными узлами.

1.2 Преимущества и ограничения статического анализа

Преимущества статического анализа:

- Раннее обнаружение ошибок (во время разработки);
- Код при этом может не собираться и не запускаться;
- Полное покрытие кода;
- Не требуется задавать тестовые входные данные;
- Лучше проверяет участки кода, сложные для тестирования (обработка граничных ситуаций);

Однако статический анализ не является заменой тестирования, поскольку не проверяет логику программ.

Для статического анализа есть ряд ограничений:

- Отсутствие знаний о входных значениях (неопределённые значения переменных).
- Экспоненциальный рост числа путей в программе.
- Большая длина отдельных путей в программе (циклы, рекурсивные вызовы).
- Сложность анализа циклов и рекурсивных вызовов.

- Большой объём данных описывающих состояния программы (возможные значения).

1.3 Основные задачи статического анализа

1. Задача доказательства корректности с пользовательскими аннотациями.

Для каждой функции известны пред-, пост-условия и, возможно, инварианты циклов. Задача анализатора заключается в доказательстве соблюдения всех пред-,пост-условий и отсутствия ошибок времени выполнения. Допускается наличие ложных срабатываний, гарантируется отсутствие пропуска ошибок. Основная сложность разработки заключается в поиске оптимального соотношения между временем работы и минимизацией количества программ, которые являются корректными, но при этом не прошли проверки анализатора (т.е. ложных срабатываний).

2. Задача доказательства корректности для подмножества языка.

Анализу подвергаются специальные программы, которые не используют некоторые возможности языка. Задача анализатора заключается в доказательстве отсутствия ошибок времени выполнения. Возможно наличие ложных срабатываний, гарантируется отсутствие пропуска ошибок. Основная сложность разработки также заключается в поиске оптимального соотношения между временем работы и минимизацией количества программ, которые являются корректными, но при этом не прошли проверки анализатора.

3. Задача доказательства корректности без ограничений.

Никаких дополнительных ограничений на анализируемые программы не накладывается. Задачей статического анализа является доказательство отсутствия некоторых классов ошибок времени выполнения. Гарантируется отсутствие как ложных срабатываний, так и пропусков ошибок, однако анализ подвержен зацикливанию. Основная сложность разработки заключается в минимизации классов программ, на которых анализ зацикливается.

4. Задача поиска дефектов.

Никаких дополнительных ограничений на анализируемые программы не накладывается. Задача анализа — поиск потенциальных ошибок времени

выполнения. Допускается присутствие ложных срабатываний и пропусков ошибок. Сложность разработки заключается в обнаружении практически значимого класса ошибок при сохранении низкого процента ложных срабатываний.[12]

1.4 Выводы

В главе были рассмотрены принципы работы статического анализа: каким образом происходит лексический и синтаксический анализ, как во время синтаксического анализа исходный код преобразовывается в структуру данных «дерево разбора» и затем представляется в виде AST-дерева. Также были перечислены задачи, которые решает статический анализ, его преимущества и недостатки, которые накладывают на анализ исходного кода некоторые ограничения.

ГЛАВА 2
МЕТОДЫ СТАТИЧЕСКОГО АНАЛИЗА

Существует множество различных методов статического анализа, в частности, анализ с обходом дерева кода, анализ потока данных, анализ потока данных с выбором пути и т. д. Конкретные реализации этих методов различны в различных анализаторах. Однако независимо от языков программирования анализаторы могут использовать один и тот же базовый код (инфраструктуру). Этот базовый код содержит набор основных алгоритмов, которые могут использоваться в разных анализаторах кода вне зависимости от предоставляемых задач и анализируемого языка. Набор поддерживаемых методов и конкретная реализация этих методов, опять же, будет зависеть от конкретной инфраструктуры.

Рассмотрим основные методы анализа данных.

2.1 Анализ потока данных (Data-Flow Analysis)

Под анализом потока данных понимают совокупность задач, нацеленных на выяснение некоторых глобальных свойств программы, то есть извлечение информации о поведении тех или иных конструкций в некотором контексте [1]. Соотвественно, такой тип анализа кода проверяет наличие проблемных конструкций на основе наборов правил, причем инструменты анализа потока также моделируют пути принятия решений, что позволяет углубиться в приложение и сильно расширить возможности поиска дефектов, таких как нулевые указатели, переполнение буфера и другие дефекты безопасности.

Существует несколько классических задач анализа потока данных.

Задача 1: Достижимые определения

Задачу можно сформулировать следующим образом: для каждого вхождения переменной требуется определить множество присваиваний, такое, что для каждого из них существует путь, в котором между ним и данным вхождением отсутствуют другие присваивания той же переменной. Другими словами задача достижимых определений заключается в выяснении, где именно устанавливаются значения того или иного вхождения данной переменной.

Задача 2: Живые переменные

В задаче требуется для каждой вершины графа потока управления построить множество переменных, обладающих следующим свойством:

существует путь через данную вершину, начинающийся присваиванием данной переменной и кончающийся ее использованием, не содержащий иных присваиваний той же переменной.

Решения данных задач применяются при оптимизации кода программы.

2.2 Анализ потока управления (Control-Flow Analysis)

С помощью данного метода удобно исследовать различные пути выполнения программы, которые могут иметь место, когда присутствуют вызовы функций, не возвращающих управление, исключения, а также используются ключевые слова break, continue (если говорить о языке C/C++) и т.д.

Чтобы сделать эти алгоритмы эффективными, большинство инструментов строят график потока управления поверх AST. Узлы в графе потока управления являются основными блоками: последовательностями команд, которые всегда будут выполняться, начиная с первой инструкции и заканчивая последней командой, без возможности пропуска каких-либо инструкций. Ребра в графе потока управления имеют направление и представляют собой потенциальные пути управления потоком между базовыми блоками.

2.3 Символьное выполнение (Symbolic Execution)

Символьное выполнение позволяет вычислять значения переменных, которые могут приводить к ошибкам, производить проверку диапазонов значений.

Зная предполагаемые значения переменных, можно выявлять такие ошибки как:

- утечки памяти;
- переполнения;
- выход за границу массива;
- разыменование нулевых указателей в C++ / доступ по нулевой ссылке в C#;
- бессмысленные условия;

- деление на 0;

Рассмотрим следующий пример на рисунке 2.1, используя в качестве статического анализатора PVS-Studio:

```
int Foo(int A, int B)
{
    if (A == B)
        return 10 / (A - B);
    return 1;
}
```

Рисунок 2.1 - Пример кода, в котором содержится ошибка типа “деление на 0”

На данный код анализатор выведет предупреждение:
error: V609 Divide by zero. Denominator 'A - B' == 0.
Значение переменных *A* и *B* неизвестны анализатору. Зато анализатор знает, что в момент вычисления выражения *10 / (A - B)* переменные *A* и *B* равны. Следовательно, произойдёт деление на ноль. Если же значения *A* и *B* будут известны анализатору, то он подставит их и проанализирует ситуацию. Рассмотрим следующий пример кода на рисунке 2.2:

```
double Division(int x)
{
    return 5.0 / x;
}

void Foo()
{
    double result = 0.0;
    for (int i = 0; i < 4; ++i)
        result += Division(i);
}
```

Рисунок 2.2 - Пример кода, в котором содержится ошибка “потенциальное деление на 0”

В данном случае анализатор тоже предупредил о делении на ноль, т.к. на одной из итераций цикла такая ситуация возможна.

- *error: V609 Divide by zero. Denominator 'x' == 0. The 'Division' function processes value ‘0'.*

- *error: V609 Divide by zero. Denominator 'x' == 0. The 'Division' function processes value '[0..3]'.*

В данном примере работают сразу несколько технологий: анализ потока данных, символьное выполнение и автоматическое аннотирование методов (будет рассмотрено далее). Анализатор видит, что переменная x используется в функции Division как делитель. На основании этого для функции Division автоматически строится специальная аннотация. Далее учитывается, что в функцию в качестве аргумента x передаётся диапазон значений [0..3]. Анализатор приходит к выводу, что возникнет деление на ноль.

2.4 Аннотирование методов (Method Annotations)

Вместо того, чтобы рекурсивно проверять корректность работы каждого метода, можно задать аннотации для методов языка, то есть какие-то характеристики и ограничения на входные параметры, которые помогут в поиске ошибок. Можно составлять аннотации, самостоятельно анализируя тело функции. Так приходится делать, когда нельзя получить доступ к телу функции или тело функции описано в модуле, который собирается динамически и т.д. Для популярных библиотек создатели статических анализаторов составляют ручные аннотации методов. Рассмотрим следующий пример на рисунке 2.3:

```
void Foo(FILE *file)
{
    char buf[100];
    size_t i = fread(buf, sizeof(char), 1000, file);
    buf[i] = 0;
}
```

Рисунок 2.3 - Пример кода, в котором содержится ошибка “потенциальное переполнение буфера”

Согласно аннотации функции fread в PVS-Studio размер буфера, переданный в функцию, должен быть не меньше, чем количество байт, которое планируется прочитать из файла.

Анализатор выдал следующее предупреждение:

error: V512 A call of the 'fread' function will lead to overflow of the buffer ‘buf’.

Анализатор нашел произведение второго и третьего фактических аргументов и вычислил, что функция может прочитать до 1000 байт данных.
При этом, размер буфера составляет только 100 байт, и может произойти его переполнение.
warning: V557 Array overrun is possible. The value of 'i' index could reach 1000.
Поскольку функция может прочитать до 1000 байт, то диапазон возможных значений переменной i равен [0..1000]. Соответственно, может произойти доступ к массиву по некорректному индексу.

2.5 Сопоставление с шаблоном (Pattern-based analysis)

Этот метод достаточно прост в реализации. На основе дерева разбора (или AST-дерева) применяется определенный паттерн для определенных типов узлов. С помощью этого метода удобно искать ошибки Copy Paste. Рассмотрим следующий пример на рисунке 2.4:

```
int func(bool condition)
{
    int left_value = 0;
    int right_value = 1;
    return condition ? left_value : left_value;
}
```

Рисунок 2.4 - Пример кода, в котором содержится ошибка “тернарный оператор всегда возвращает одно и то же значение”

Анализатор выдал следующие предупреждение:
error: V583 The '?:' operator, regardless of its conditional expression, always returns one and the same value: left_value.
Анализатор рассмотрел тернарный условный оператор и выражения в then- и else- ветках. Эти значения совпадают, значит, скорее всего, что-то не так(должно было быть left_value : right_value).

2.6 Выводы

В главе были рассмотрены основные методы статического анализа исходного кода, а именно: анализ потока данных, анализ потока управления, символьное выполнение, аннотирование методов, сопоставление с шаблоном. Также были продемонстрированы примеры анализа различными методами с помощью статического анализатора PVS-Studio.

ГЛАВА 3
КЛАССИФИКАЦИЯ УЯЗВИМОСТЕЙ ЗАЩИТЫ

3.1 Основные виды уязвимостей

Когда требование корректной работы программы на всех возможных входных данных нарушается, становится возможным появление уязвимостей защиты. Их можно классифицировать в зависимости от программных ошибок.

Vulnerabilities By Type

Year	# of Vulnerabilities	DoS	Code Execution	Overflow	Memory Corruption	Sql Injection	XSS	Directory Traversal	Http Response Splitting	Bypass something	Gain Information	Gain Privileges	CSRF	File Inclusion	# of exploits
1999	894	177	112	172			2	7		25	16	103			2
2000	1020	257	208	206		2	4	20		48	19	139			
2001	1677	403	403	297		7	34	123		83	36	220		2	2
2002	2156	498	553	435	2	41	200	103		127	74	199	2	14	1
2003	1527	381	477	371	2	49	129	60	1	62	69	144		16	5
2004	2451	580	614	410	3	148	291	111	12	145	96	134	5	38	5
2005	4935	838	1627	657	21	604	786	202	15	289	261	221	11	100	14
2006	6610	893	2719	663	91	967	1302	322	8	267	271	184	18	849	30
2007	6520	1101	2601	954	95	706	884	339	14	267	324	242	69	700	44
2008	5632	894	2310	699	128	1101	807	363	7	288	270	188	83	170	74
2009	5736	1035	2185	700	188	963	851	322	9	337	302	223	115	138	738
2010	4652	1102	1714	680	342	520	605	275	8	234	282	238	86	73	1493
2011	4155	1221	1334	770	351	294	467	108	7	197	409	206	58	17	557
2012	5297	1425	1459	843	423	243	758	122	13	343	389	250	166	14	624
2013	5191	1455	1186	859	366	156	650	110	7	352	511	274	123	1	205
2014	7946	1598	1574	849	420	305	1105	204	12	457	2106	239	264	2	401
2015	6484	1791	1826	1082	749	218	778	150	12	577	748	367	248	5	127
2016	6447	2028	1494	1325	717	94	497	99	15	444	843	600	87	7	1
2017	14714	3154	3004	2797	745	503	1516	274	11	629	1705	459	327	18	6
2018	16555	1852	3036	2487	400	516	2004	516	11	709	1430	247	461	31	4
2019	3993	355	729	461	163	95	382	82	1	163	287	44	98	11	
Total	114592	23038	31165	17717	5206	7532	14052	3912	163	6043	10448	4921	2221	2206	4333
% Of All		20.1	27.2	15.5	4.5	6.6	12.3	3.4	0.1	5.3	9.1	4.3	1.9	1.9	

Рисунок 3.1 - Данные о количестве уязвимостей за период с 1999 по 2019 год в зависимости от вида уязвимости

Согласно CVE (Common Vulnerabilities and Exposures) - базе данных общеизвестных уязвимостей информационной безопасности[15] - по рисунку 3.1 можно сделать вывод, что уязвимости переполнения буфера, а также некорректной обработки ввода данных и форматирования строк стабильно являются наиболее популярными среди всех обнаруженных уязвимостей, поэтому далее рассмотрим их подробнее.

1. **Переполнение буфера.** Эта уязвимость возникает из-за отсутствия контроля за выходом за пределы массива в памяти во время выполнения программы. Когда слишком большой пакет данных переполняет буфер ограниченного размера, содержимое посторонних ячеек памяти перезаписывается, происходит сбой и аварийный выход из программы.

2. **Уязвимость некорректной обработки ввода данных.** Уязвимости могут возникать в случаях, когда вводимые пользователем данные без достаточного контроля передаются интерпретатору некоторого внешнего языка. В этом случае пользователь может таким образом задать входные данные, что запущенный интерпретатор выполнит совсем не ту команду, которая предполагалась авторами программы.

3. **Уязвимость некорректного форматирования строк.** Данный тип уязвимости возникает из-за недостаточного контроля параметров при использовании функций форматного ввода-вывода (например некоторых функций стандартной библиотеки языка C). Эти функции принимают в качестве одного из параметров символьную строку, задающую формат ввода или вывода последующих аргументов функции, но при этом размер и характер переданных данных не анализируется.

3.2 Ошибки обработки ввода данных

Одна из самых распространенных уязвимостей — некорректный ввод исходных данных. С одной стороны может показаться, что в таких ошибках нет ничего критичного, но на самом деле если программа не гарантирует, что введенные данные всегда будут правильно отформатированы, следовать стандартам, иметь смысл и соответствовать правилам кодирования, то программист должен обратить на это внимание. Не стоит априори доверять источнику входных данных можно доверять, т.к. программа может получить входные данные из менее заслуживающего доверия источника или сам доверенный источник может быть скомпрометирован.

Что необходимо проверять :

- Все входные данные

Проверьте все места ввода данных в программе.

- Входные данные из всех источников

Проверьте входные данные из всех источников, включая параметры командной строки, файлы конфигурации, запросы к базе данных, переменные среды, сетевые службы, системные свойства, временные файлы и любые другие внешние источники.

- Установить доверительные границы

Храните надежные и ненадежные данные отдельно, чтобы гарантировать, что проверка входных данных всегда выполняется.

Как необходимо проверять:

- Используйте строгую проверку ввода

Используйте самую сильную форму проверки входных данных, применимую в данном контексту.

- Не путайте удобство с безопасностью

Различайте проверку, которую приложение выполняет для удобства использования, с проверкой ввода для безопасности.

- Отклонить неверные данные

Отклоняйте данные, которые не проходят проверки. Не стоит корректировать данные и делать их более приемлемыми для дальнейшего использования.

- Сделать хорошую проверку ввода по умолчанию

Используйте слой абстракции вокруг важных или опасных операций, чтобы гарантировать, что проверки безопасности всегда выполняются и что опасные условия не могут возникнуть.

- Всегда проверяйте длину ввода

Проверьте входные данные по минимальной ожидаемой длине и максимальной ожидаемой длине.

- Ограничьте числовой ввод

Проверьте числовой ввод относительно максимального и минимального значения. Обратите внимание на операции и методы, которые работают с введенным числом, нарушится ли их поведение при выходящих за границы значениях.

Решение: простой способ использовать статический анализ для помощи в проверке входных данных — это указать инструменту все места, где программа принимает ввод. Совокупность таких мест можно условно назвать поверхностью атаки приложения. В инструменте статического анализа поверхность атаки состоит из всех точек входа в программу и вызовов функций-источников, то есть набора функций, которые вызываются извне или осуществляют пользовательский ввод в программу. Как правило, чем больше поверхность атаки, тем больше внимания нужно уделить проверке правильности ввода.

3.3 Переполнение буфера

Переполнение буфера происходит, когда программа записывает данные за пределы выделенной памяти. Ошибки переполнения буфера дают злоумышленнику контроль над уязвимым кодом, а именно возможность перезаписи значений в памяти в интересах атакующего.

Лучший способ предотвратить уязвимости переполнения буфера — это использовать язык программирования, который обеспечивает безопасность памяти и безопасность типов. При написании программного кода на небезопасных языках, из которых C и C ++ являются наиболее широко используемыми, программист берет на себя ответственность за предотвращение нежелательных изменений операций в памяти. Любая операция, которая манипулирует памятью, может привести к переполнению буфера, но на практике ошибки, которые чаще всего приводят к переполнению буфера, сгруппированы вокруг ограниченного набора операций.

В случае классической атаки с разбиванием стека злоумышленник отправляет данные, содержащие сегмент вредоносного кода, в программу, уязвимую к переполнению буфера в стеке. Помимо вредоносного кода, злоумышленник включает в себя адрес памяти начала кода. Когда происходит переполнение буфера, программа записывает данные злоумышленника в буфер и продолжает работу за пределами буфера, пока в конечном итоге не перезаписывает адрес возврата функции адресом начала вредоносного кода. Когда функция возвращается, она переходит к значению, сохраненному в ее адресе возврата. Поскольку адрес возврата был перезаписан, элемент управления переходит к буферу и начинает выполнять вредоносный код злоумышленника.

Решение: выбранный программистом подход к распределению памяти будет влиять на то, как вы используете статический анализ для выявления потенциальных переполнений буфера и других нарушений памяти в коде. Механизмы статического распределения буфера обычно легче проследить и проверить. Поскольку буферы назначаются с фиксированным размером во время компиляции, анализатор может анализировать размеры буферов, используемых в различных операциях и идентифицировать ошибки. Кроме выхода за границы буфера, так же опасными ситуациями являются использование ресурса после его освобождения и двойное освобождение

ресурса, т.к. они могут вызвать ошибки сегментации и потенциально могут привести к переполнению буфера.

Ошибки использования ресурса после освобождения возникают, когда программа продолжает использовать указатель после освобождения. Если освобожденная память уже перераспределена, злоумышленник может использовать указатель на память для запуска атаки переполнения буфера.

В качестве решения можно установить следующее правило для анализатора (рисунок 3.2), по которому после освобождения указателя, ему присваивалось значение NULL. Другое правило может искать вызовы функций освобождения памяти (например, функция delete в C++ или в общем случае на рисунке 3.2 обозначена как free()), а затем сообщает об ошибке, если следующая операция не устанавливает свободную переменную в NULL.

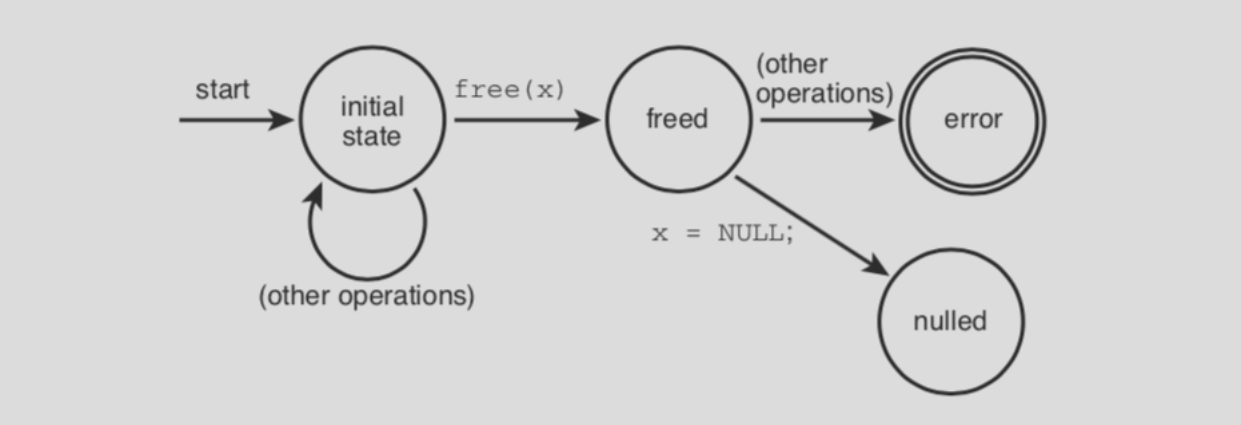


Рисунок 3.2 - Схематичное изображение правила работы с освобожденным ресурсом

3.4 Ошибки форматирования строк

Базовая структура данных строки в языке C (массив символов с нулевым символом в конце) подвержен ошибкам, а встроенные функции стандартной библиотеки для работы со строками только усугубляют ситуацию.

При использовании многих функций манипуляций со строками C легко допустить ошибку. Вместо того чтобы стараться быть с ними особенно осторожными, лучше всего вообще избегать их. В частности, стоит избегать использования таких функций манипуляции со строками как get(), scanf(), strcpy() или sprintf ().

Поведение метода get () довольно простое: функция читает из потока, на который указывает stdin, и копирует данные в буфер, пока не достигнет символа новой строки (\n). Соответственно, функция переполняет свой целевой буфер

всякий раз, когда количество символов, считываемых из входного источника, превышает буфер, переданный в get (). Хотя функция scanf() немного сложнее, чем get(), она тоже уязвима, поскольку предназначена для чтения произвольного количества форматированных данных в один или несколько буферов фиксированного размера. Когда scanf() встречает спецификатор “%s” в строке формата, он считывает символы в соответствующий буфер до тех пор, пока не встретится значение, отличное от ASCII, что может привести к переполнению буфера, если в функцию передается количество данных больше, чем размер буфера. Если указан спецификатор ширины, например, “%255s”, scanf() будет считывать до указанного количества символов в буфер. Из-за возможности ограничить количество читаемых входных данных функция может быть безопасно использована, если спецификатор формата правильно ограничивает объем считываемых данных. Такой метод использования scanf() более безопасный, однако данный метод подвержен другим ошибкам и корректность его работы сильно зависит от входных данных.

Решение: хорошей практикой является составление списка небезопасных функций, использование которых анализатор должен запрещать. Решение о том, какие именно функции должны быть запрещены, зависит от программы и зависит от таких факторов, как история базы кода, специфические для контекста риски, связанные с окружением, в котором используется программа и т.д.

3.5 Выводы

В главе были рассмотрены основные классификации уязвимостей защиты, которые может предотвратить статический анализ, а именно: переполнение буфера, ошибки обработки ввода данных, ошибки форматирования строк. Хотя ошибкам обработки ввода данных и ошибкам форматирования строк зачастую не уделяется должного внимания, и данные ошибки считаются не критичными, они могут существенно повлиять на работоспособность вашей программы. В главе описано, какие части вашей программы могут содержать различные виды уязвимостей и как от них можно себя предостеречь.

Переполнение буфера до сих пор является актуальной и очень распространенной уязвимостью таких языков как C и C++, без которых сложно представить разработку программного обеспечения в том числе и для

встроенных систем, про которые пойдет речь в следующей главе. Поэтому пока сами разработчики данных языков не повысят его безопасность, наиболее эффективным решением является использование методов статического анализа.

ГЛАВА 4
ИСПОЛЬЗОВАНИЕ СТАТИЧЕСКОГО АНАЛИЗА В РАЗРАБОТКЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ДЛЯ ВСТРОЕННЫХ СИСТЕМ

4.1. Роль статического анализа при разработке для встроенных систем

Статический анализ позволяет обнаружить ошибки еще на этапе написания кода, что экономит время на отладку в будущем и удешевляет процесс разработки. Особенно полезным статический анализ может быть при разработке встроенных систем. Поскольку в таких системах ошибки часто может содержать не только написанный программный код, а также и само устройство (например, из-за плохо изготовленного макета отсутствует контакт на микроконтроллере), то использование статического анализа экономит время на поиск ошибок хотя бы на стороне разработчика программного обеспечения, и можно будет больше времени уделить проверке самого устройства.

Программы для встроенных систем подвергаются сильной оптимизации по памяти и времени выполнения, поэтому для их разработки в основном используется языки C и C++, которые напрямую работают с памятью и подвержены многим уязвимостям безопасности.

Цена ошибок в программах для встроенных систем очень высока, ведь их невозможно или очень сложно и дорого исправлять, когда уже началось серийное производство. Иногда это бывает особенно опасно даже для жизни людей (разработка программного обеспечения для машин, ракет). Поэтому код встраиваемых устройств должен тестироваться как можно тщательнее, особенно, если ошибки могут привести к жертвам или материальным потерям.

Статические анализаторы имеют базу знаний о различных шаблонах кода, которые в определённых условиях приводят к ошибке. Статический анализатор обращает внимание на ненадежный код и помогает сделать программу менее зависимой от версии и настроек компилятора. Также анализатор удобно использовать для контроля соответствия кода определённому стандарту разработки программного обеспечения, такому как MISRA.

Встраиваемые системы все чаще становятся включенными в сеть, что подвергает их атакам по сети. Будь то код для сетевых маршрутизаторов, медицинских устройств или систем домашней безопасности, любое устройство с уязвимостью в сети становится открытым для кибератак.

В качестве доказательства можно рассмотреть исследование, проведенное Институтом системных наук IBM[14], результаты которого приведены на

рисунке 4.1. Из диаграммы можно сделать вывод, что на более ранних этапах разработки внесение изменений и исправление ошибок занимает гораздо меньше времени, а соответственно, и денег, чем на более поздних этапах.

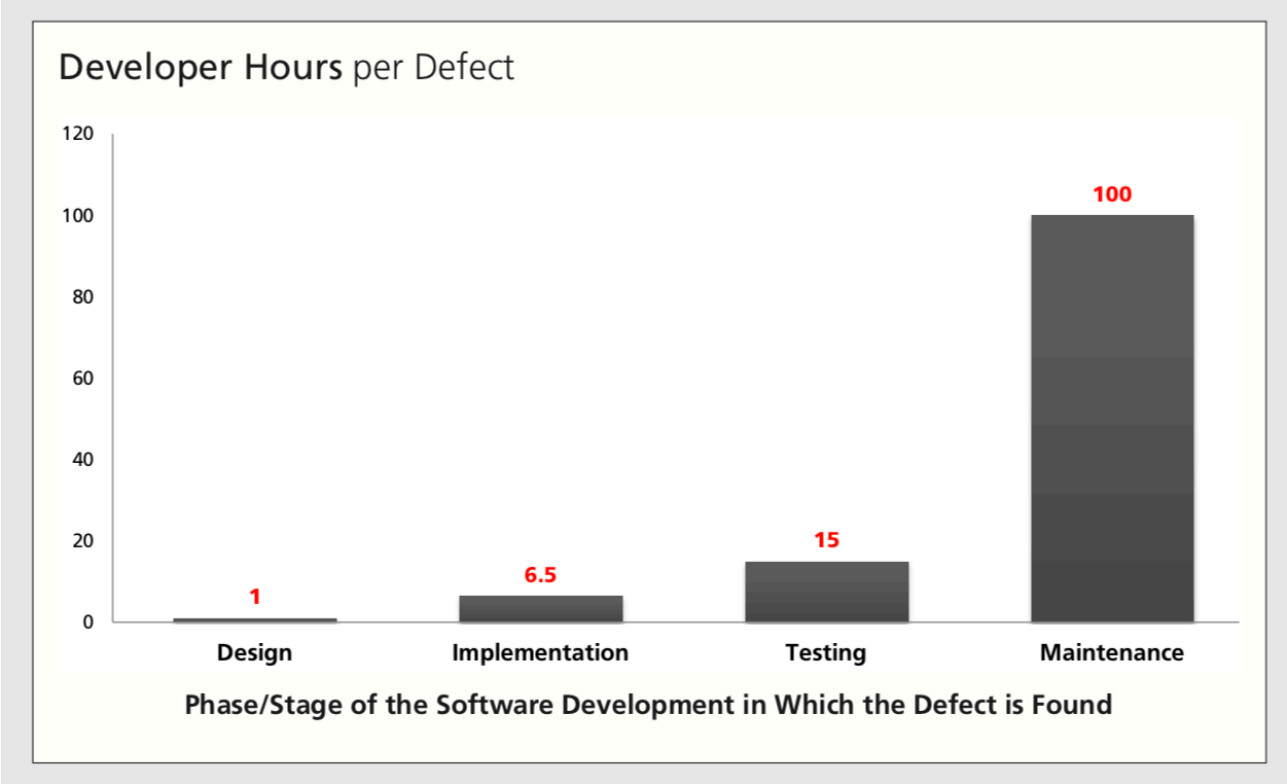


Рисунок 4.1 - Диаграмма времени, которое необходимо разработчику для устранения ошибки на различных этапах написания кода

4.2. Основные виды анализа программного кода для встроенных систем

- **Бинарный анализ**

Чаще всего статический анализ применяется к исходному коду программы и промежуточному представлению. Однако бинарный статический анализ становится все более популярным из-за нескольких тенденций. Одна из них — это расширение функциональности программного обеспечения путем использования внешних библиотек. Например, в популярную для разработки для встроенных систем среду разработки Eclipse, интегрируется такой инструмент как STMCubeMX для более быстрого и удобного написания кода для микроконтроллеров. В других ситуациях в проект необходимо интегрировать несколько готовых модулей заказчика. Для того, чтобы быть уверенным в надежности собственной системы, необходимо проверить модули на наличие неисправностей.

- **Комплексная безопасность**

Тенденция к созданию сетей во встраиваемых системах создала более широкие потенциальные возможности для атак злоумышленников. Эксплойты[8] обычно запускаются, когда злоумышленник отправляет данные по входному каналу, такому как сетевой порт. Программист может защититься от этих уязвимостей, рассматривая входные данные как потенциально опасные и проверяя их перед тем, как использовать их в приложении.

Обнаружение таких потенциальных эксплойтов является нетривиальной задачей, потому что для этого необходимо вручную отслеживать поток данных через все приложение. Использование автоматических инструментов, таких как статические анализаторы, для проверки данных на наличие потенциально вредоносных входов значительно сокращает время поиска ошибок и повышает эффективность кода.

- **Проверка параллелизма**

Программирование многопоточных приложений содержит трудно обнаруживаемые ошибки и не всегда удобно в отладке. Поскольку все большее внимание уделяется разработке многоядерных чипов, программное обеспечение должно быть многопоточным, чтобы использовать преимущества современного оборудования. Решения для расширенного статического анализа были доступны для решения проблем параллелизма для программ на C и C ++, но теперь Java — это третий по популярности язык для встраиваемых систем, а средств для анализа кода на нем гораздо меньше.

- **Поддержка стандартов**

Такие стандарты как MISRA или ISO 26262 часто используются в комбинации в автомобильной, аэрокосмической, медицинской промышленности и др. Организации в этих отраслях должны быть оснащены соответствующими инструментами для выявления не только нарушений поверхностных синтаксических ошибок, но и более серьезных ошибок, возникающих, например, из-за неопределенного поведения, как это предусмотрено стандартом MISRA C: 2012[16].

4.3. Сравнение существующих статических анализаторов для встроенных систем

1. Polyspace Bug Finder

В данном анализаторе при анализе безопасности встроенного программного обеспечения на языке C используются рекомендации по безопасному кодированию, такие как CERT C Coding Standard, MISRA C, ISO / IEC TS 17961, а также уязвимости в системе безопасности, которые описаны в Common Weakness Enumeration (CWE). Статический анализатор Polyspace исследует программный код на нарушение рекомендаций в данных стандартах. По результатам верификации Polyspace отмечает каждую операцию цветом, указывающим, содержит ли эта операция ошибки при выполнении, является недостижимой или недоказанной, такая визуализация ошибок помогает удобнее и быстрее анализировать исходный код программы.

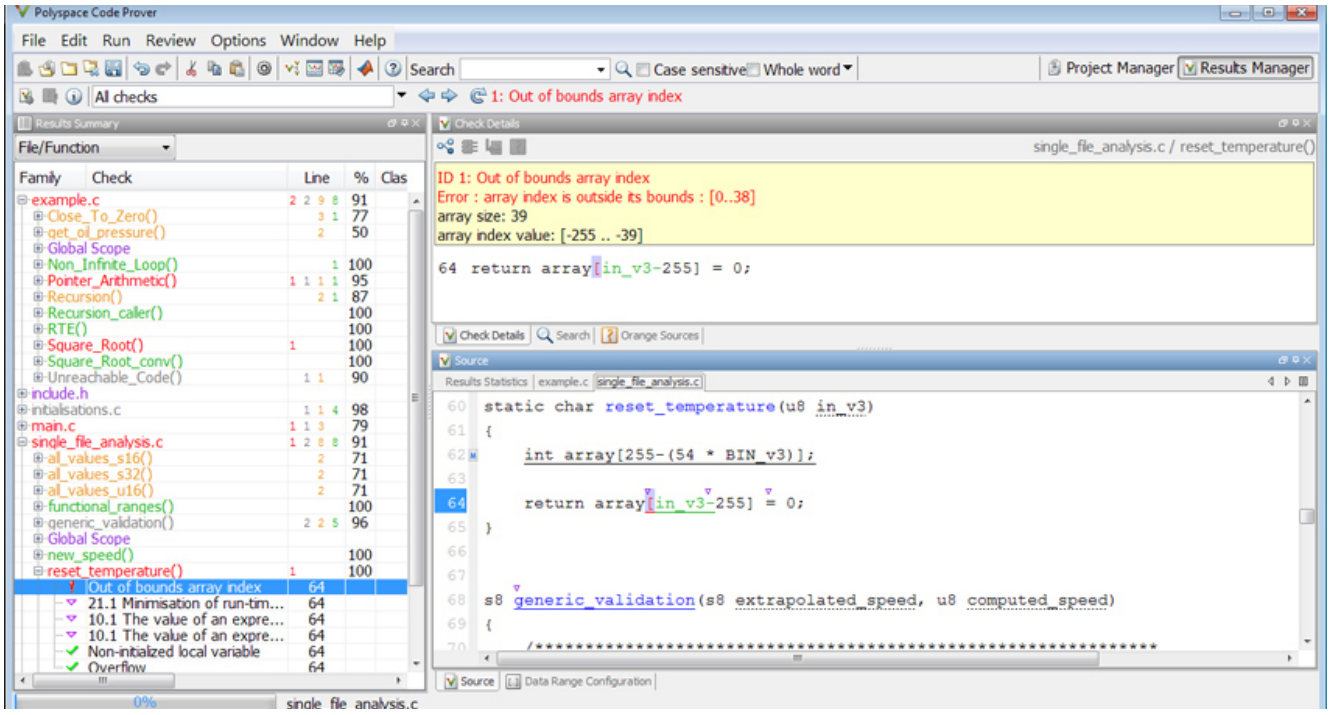


Рисунок 4.2 - Пример анализа кода анализатором Polyspace Bug Finder

2. HelixQAC

Данный статический анализатор поддерживает стандарты MISRA C, CWE, а также стандарт кодирования на языке C++14 для автомобильной промышленности AUTOSAR и стандарт CERT Secure Coding Standards,

следование которому помогает обнаружить уязвимости в безопасности и применить лучшие методы для устранения неопределенного поведения программы.

3. Goanna Studio

Статический анализатор языков C/C++ , который интегрируется в среду разработки Eclipse, одну из самых распространенных для программирования встроенных систем. Также поддерживает стандарты MISRA, CWE, CERT. Goanna обнаруживает такие программные ошибки как переполнение буфера, утечки памяти, арифметические ошибки, ошибки переносимости, недостатки безопасности, использование небезопасных библиотек и деление на ноль.

4. CodeSonar

Данный статический анализатор позволяет одновременно анализировать сторонние библиотеки и собственный программный код. Анализатор находит дефекты в коде, вызванные неправильным использованием библиотек. Они могут возникать из-за того, что документация не всегда является явной, и возможны случаи, когда сторонняя библиотека работает иначе, чем ожидал разработчик. При этом анализатор выполняет аудит сторонних библиотек, чтобы убедиться, что в них нет важных дефектов, и их использование не вызывает утечек памяти.

5. Imagix 4D

Статический анализатор C/C++, Java обеспечивает проверку безопасности потоков и управления параллелизмом, обнаруживает потенциальные конфликты, ведущие к тупикам и условиям гонки. В данном анализаторе существует возможность добавления собственных исключений из общепринятых стандартов проектирования и кодирования специально под нормы, принятые в вашей организации. Также поддерживается стандарт CWE, и существует функция визуализации исходного кода и всех связанных с ним зависимостей.

4.4 Выводы

В главе описано, почему статический анализ необходим при разработке программного обеспечения для встроенных систем, какие инструменты уже существуют, коротко описаны их преимущества и недостатки. Также были рассмотрены основные виды анализа исходного кода для встроенных систем (бинарный анализ, комплексная безопасность, ошибки параллелизма, поддержка стандартов).

ГЛАВА 5
РЕАЛИЗАЦИЯ ИНСТРУМЕНТА СТАТИЧЕСКОГО АНАЛИЗА

5.1 Постановка задачи и принцип работы приложения

Все большее внимание уделяется написанию безопасного кода в том числе и самими создателями языка и его стандартов, поэтому методы, которые часто приводят к появлению уязвимостей в программе, заменяют на более безопасные. Список методов, которые подвержены уязвимостям, можно найти в регулярно обновляющейся документации по языку и учесть в написании своего программного кода. Процесс поиска методов, которые помечены разработчиками языка как “не рекомендованные” может быть автоматизирован, поэтому было бы удобно создать приложение, которое ищет небезопасные функции языка, а также другие уязвимости, которые могут привести к непредвиденному завершению программы и могут использоваться злоумышленником для запуска атаки переполнения буфера.

Перед приложением ставятся следующие задачи:

1. Анализ исходного программного кода на языке C/C++ на наличие в нем уязвимостей, а именно:
    - обнаружение небезопасных стандартных функций и предложение альтернативы;
    - обнаружение мест, где ресурсы использовались после освобождения, т.е. небезопасное обращение с памятью;
    - обнаружение мест, где происходило переполнение буфера, т.е. небезопасное обращение с памятью;
2. Реализация пользовательской части приложения, где пользователь имеет следующие возможности:
    - добавление исходного кода программы для анализа самостоятельно;
    - добавление файла с исходных кодом программы для анализа через файловую систему;
    - сохранение результатов статического анализа в файл;

В качестве языка написания приложения был выбран Python, поскольку он позволяет использовать существующие решения наиболее эффективно и обладает большим набором дополнительных библиотек. Пользовательский интерфейс реализован с помощью библиотеки Tkinter, логика приложения

построена на анализе исходного кода с помощью регулярных выражений, т.е. места, подозрительные на существование уязвимостей, находятся по определенным заранее заданным правилам.

5.2 Алгоритмы поиска статическим анализатором уязвимостей в исходном коде

1. Подготовка к анализу исходного кода

Исходный код, написанный на языке C/C++, который необходимо проанализировать можно добавить в приложение двумя способами: вручную, т.е. Самостоятельно вставить его в левое окно приложения или через файловую систему (рисунок 5.1).

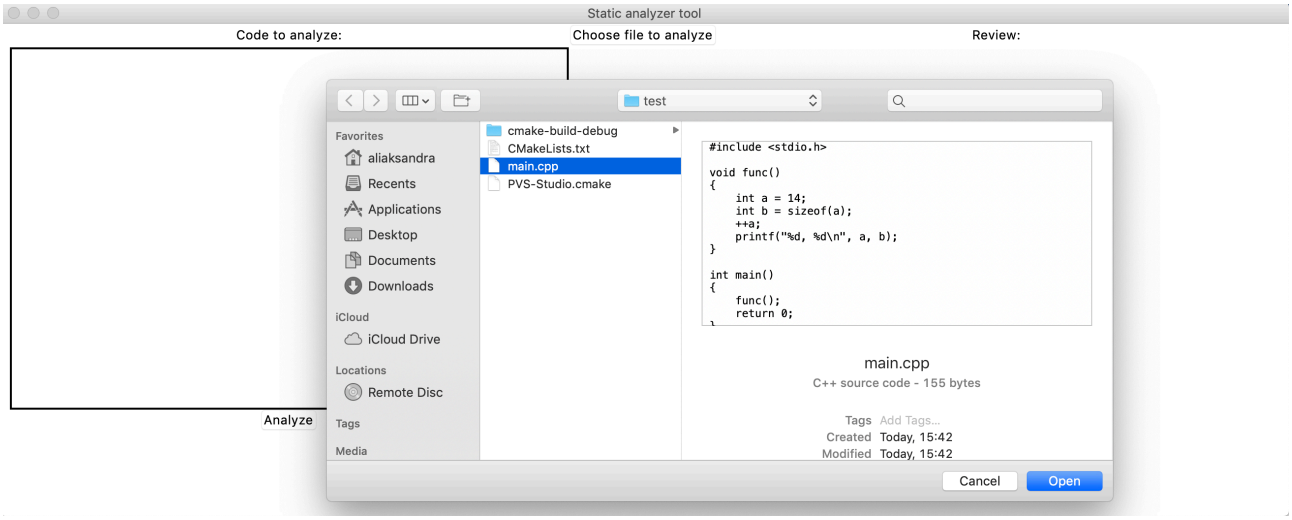


Рисунок 5.1 - Добавление исходного кода, который нужно проанализировать, через файловую систему

2. Поиск небезопасных функций

Для хранения данных о функциях была использован ассоциативный контейнер dictionary (рисунок 5.2). Ключом является небезопасная функция, а значению по данному ключу соответствует альтернативная ей безопасная функция.

```
methods_replacement_rule = {
    'strcpy': 'strcpy_s',
    'strcat': 'strcat_s',
    'sprintf': 'sprintf_s',
    'strlen': 'strlen_s',
    'printf': 'printf_s',
    'ctime': 'ctime_s',
    'rewind': 'fseek'
}
```

Рисунок 5.2 - Структура данных, которая хранит соответствие между небезопасными и более безопасными функциями

Поиск функций из данного контейнера осуществляется с помощью инструмента “регулярные выражения”, который ищет по всему исходному коду программы небезопасные функции, а также запоминает их месторасположение, чтобы более наглядно вывести результаты анализа (рисунок 5.3).

```
def unsafe_functions_warning(unsafe_method, file, positions):
    if len(positions) == 1:
        pos_str = "position "
    else:
        pos_str = "positions "
    warning = "Method " + unsafe_method + " in " + pos_str + ",".join(map(str, positions)) + \
              " is unsafe. You should better use " + methods_replacement_rule[unsafe_method] + "\n"
    file.write(warning)


def find_unsafe_functions(data, result_file):
    for method in methods_replacement_rule:
        matches = [i.start() for i in re.finditer(method, data)]
        if len(matches):
            unsafe_functions_warning(method, result_file, matches)
```

Рисунок 5.3 - Функции, обеспечивающие поиск небезопасных функций языка C++ и вывод предупреждающего сообщения пользователю

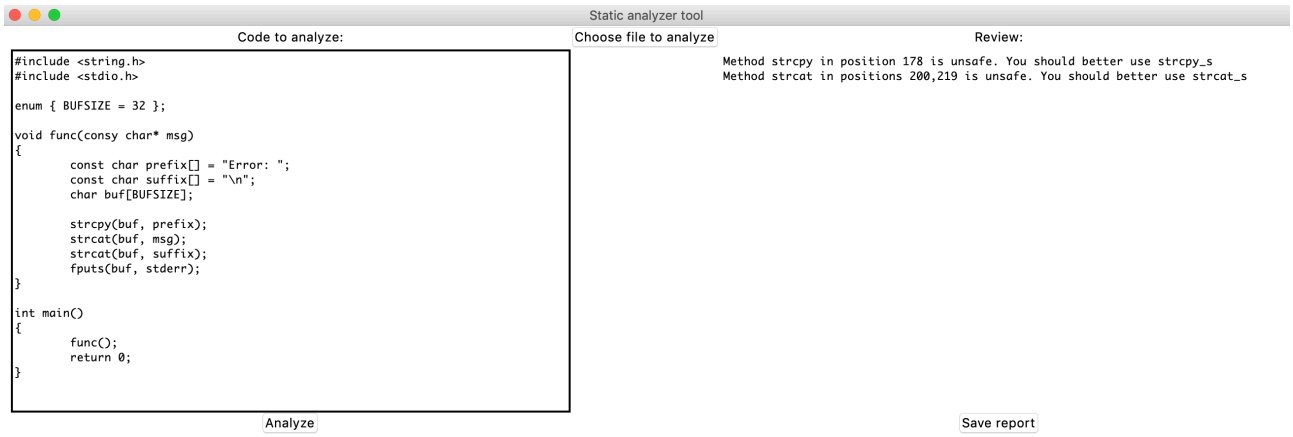


Рисунок 5.4 - Результат работы анализатора

На рисунке 5.4 изображен результат работы анализатора на примере. В данном случае были обнаружены небезопасные функции printf, strcpy, strcat.

3. Поиск мест, где происходит обращение к ресурсам после их освобождения

Для поиска мест, где происходит обращение к ресурсам после их освобождения было создано регулярное выражение следующего вида (рисунок 5.5), которое сначала осуществляет поиск объявления указателей определенных типов данных, а именно тех, что хранятся в переменной var_types. Например, данное регулярное выражение обнаружит инициализацию вида “char* buffer;”, но не обнаружит “char buf[12];”. После того, как была получена новая переменная, начинается поиск вызова функции delete для нее. После ее обнаружения использование переменной больше не безопасно.

```
var_types = ['char', 'int', 'uint8_t', 'uint16_t', 'uint32_t']

def find_using_variables_after_deleting(data, result_file):
    r = re.search("((" + '|'.join(var_types) + ")\*\s[a-zA-Z_][a-zA-Z0-9_]*)", data)
    while r:
        variable = r.group(0).split(" ")[1:][0]
        cut_data = data.split(r.group(0), 1)[1]
        data = cut_data
        r = re.search("delete\s*" + variable, data)
        if r:
            cut_after_delete_data = cut_data.split(r.group(0), 1)[1]
            delete_variable = r.group(0).split(" ")[1:][0]
            if variable == delete_variable:
                r = re.search("(" + variable + "\s*=\s*)|(" + variable + "\s*\[[a-zA-Z_0-9][a-zA-Z0-9_]*\]\s*"
                                                                         "=\s*[a-zA-Z_0-9][a-zA-Z0-9_]*)",
                              cut_after_delete_data)
                if r:
                    deleted_pointer_warning(variable, r.group(0), result_file)
        r = re.search("((" + '|'.join(var_types) + ")\*\s[a-zA-Z_][a-zA-Z0-9_]*)", data)
```

Рисунок 5.5 - Функция поиска обращения к ресурсам после их освобождения

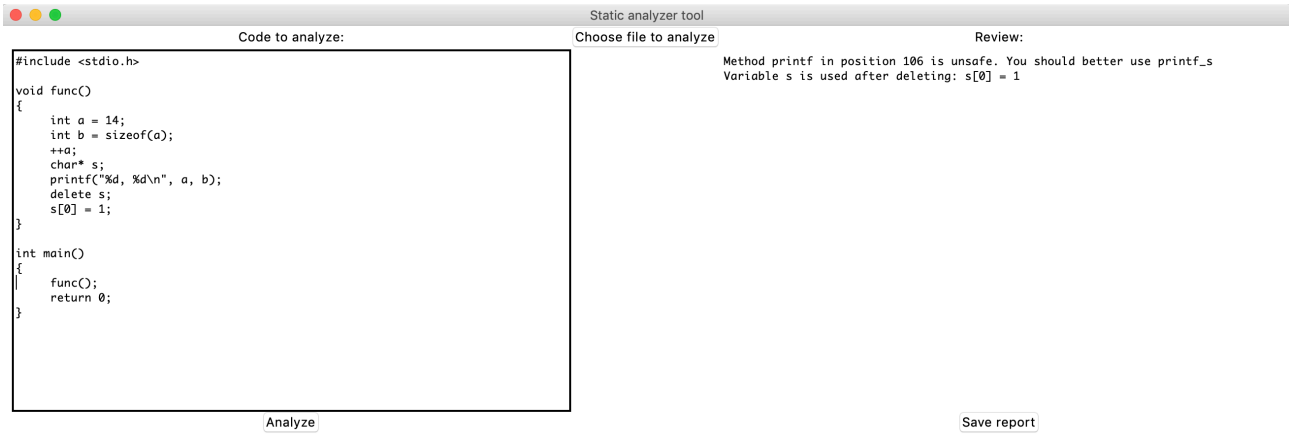


Рисунок 5.6 - Результат работы анализатора

На рисунке 5.6 изображен результат работы анализатора на примере. В данном случае было обнаружено некорректное обращение к переменной s.

4. Поиск мест, где происходит переполнение буфера

Для поиска мест, где происходит переполнение буфера было создано регулярное выражение следующего вида (рисунок 5.7), которое сначала осуществляет поиск объявления буферов различной длины, а также объявления переменных и их текущего значения, и затем информация о всех буферах и переменных заносится в контейнер array_name_and_range, где ключу соответсвует название переменной или буфера, а значению — текущий размер буфера или текущее значение переменной.

```
array_name_and_range = {}

def find_out_of_range_arrays(data, result_file):
    r = re.search("((" + '|'.join(var_types) + ")\s[a-zA-Z_][a-zA-Z0-9_]*\[[0-9]*\];)|" +
                                               "((" + '|'.join(var_types) + ")\s[a-zA-Z_][a-zA-Z0-9_]*\s*=\s*[0-9])", data)
    while r:
        r_string = r.group(0)
        if r_string.find("[") != -1:
            var = r.group(0).split(" ")[1:][0]
            array_name = var.split("[")[0]
            array_range = (var.split("[")[1]).split("]")[0]
        else:
            var = r.group(0).split(" ")[1:][0]
            array_range = r.group(0).split("= ")[1]
            array_name = var.split(" ")[0]

        array_name_and_range[array_name] = array_range
        cut_data = data.split(r.group(0), 1)[1]
        data = cut_data
        find_direct_overflow(array_name, array_range, data, result_file)
        r = re.search("((" + '|'.join(var_types) + ")\s[a-zA-Z_][a-zA-Z0-9_]*\[[0-9]*\];)", data)
```

Рисунок 5.7 - Функция поиска переполнения буфера

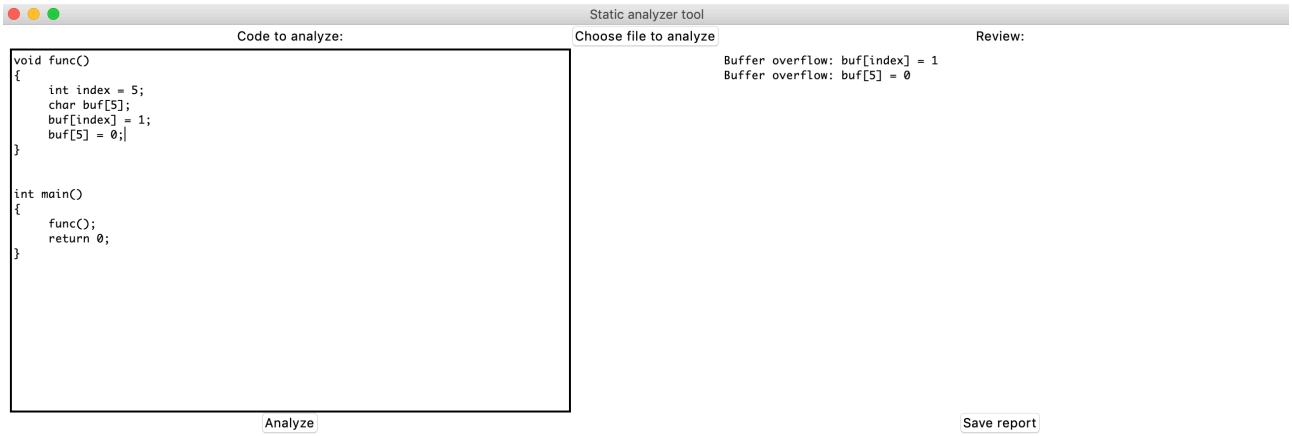


Рисунок 5.8 - Результат работы анализатора

На рисунке 5.8 изображен результат работы анализатора на примере. В данном случае было обнаружено переполнение буфера buf.

5. Сохранение результатов статического анализатора

Полученные результаты, которые анализатор выводит на экран справа, пользователь также может сохранить в выбранный файл (рисунок 5.9, рисунок 5.10, рисунок 5.11).

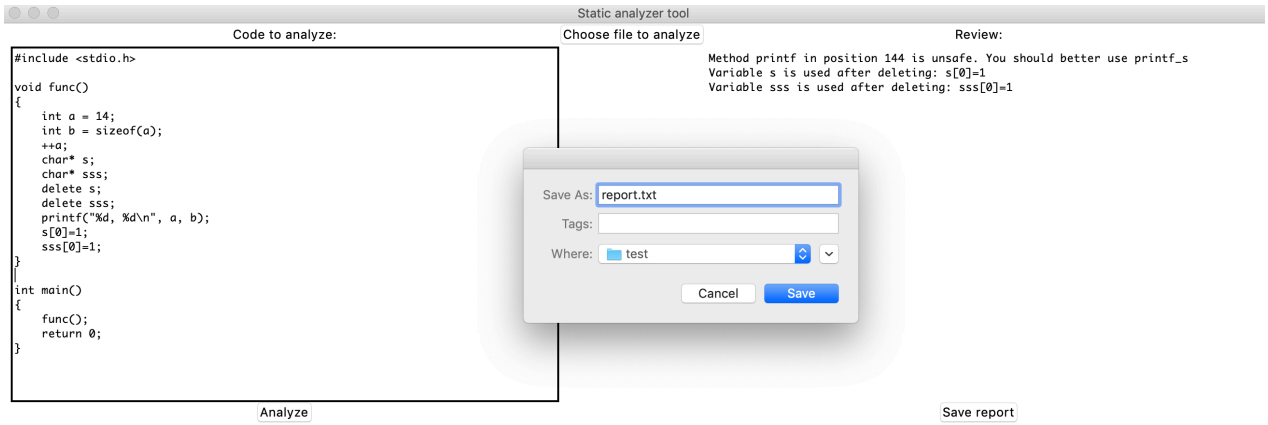


Рисунок 5.9 - Выбор файла для сохранения результата работы анализатора

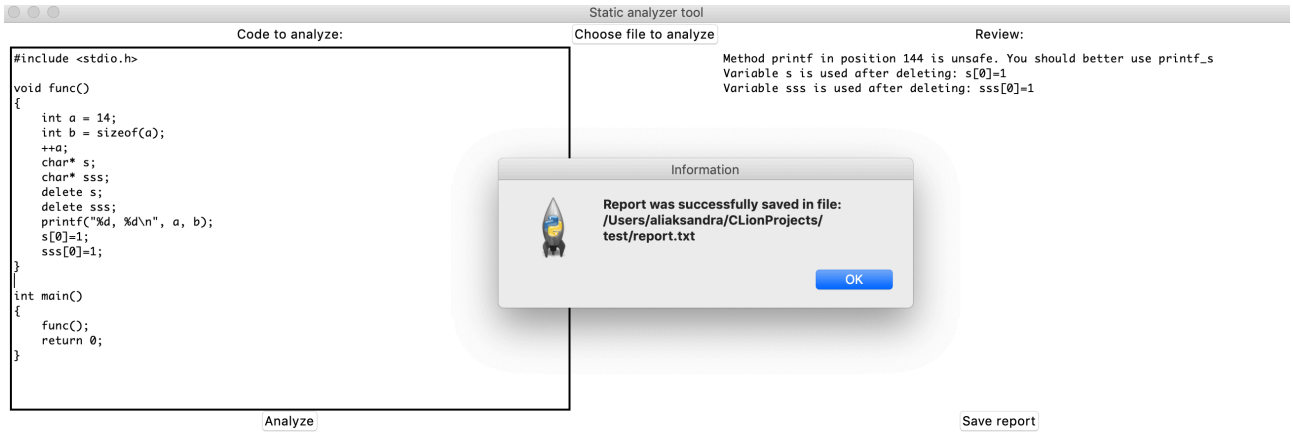


Рисунок 5.10 - Подтверждение сохранения результата работы анализатора

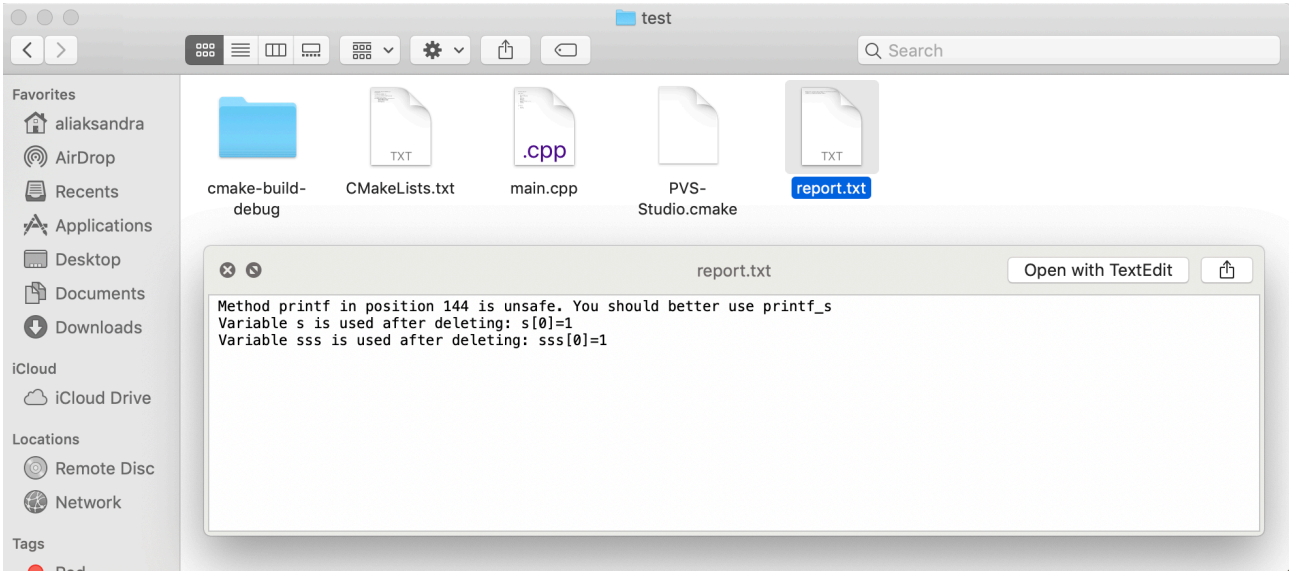


Рисунок 5.11 - Результат сохранения работы анализатора в файл

5.3 Выводы

На языке Python был разработан инструмент статического анализа, позволяющий обнаружить места использования небезопасных функций, переполнения буфера, а также обращения к ресурсам после их освобождения в исходном коде программы на языках C и C++. Также реализована возможность удобного добавления файла с исходным кодом для анализа в файловую систему, и возможность сохранения результатов работы анализатора в файл. Функциональность инструмента можно расширить и добавить поиск уязвимостей, более специфичных для программирования встроенных систем, поскольку таких статических анализаторов недостаточно, их гораздо больше для общих приложений на C и C++.

ЗАКЛЮЧЕНИЕ

Статический анализ кода является важной частью обеспечения правильного функционирования приложения. Он не только повышает общую производительность команды разработчиков, но и снижает риски, связанные с выпуском потенциально опасного программного обеспечения. В то время как цена ошибки в программном обеспечении для встроенных систем может быть очень высока.

В работе были рассмотрены основные принципы и методы работы статического анализатора, уязвимости, которые препятствуют правильной работе программного обеспечения. Было продемонстрирована работа с одним из самых популярных статических анализаторов для языка C++ PVS-Studio и создано приложение, которое ищет небезопасные функции в программном коде, обращает на это внимание разработчика и предлагает более безопасную альтернативу, а также ищет места переполнения буфера и обращения к ресурсу после его освобождения.

Поскольку уязвимостей с каждым годом не становится меньше, то для создания эффективных инструментов статического анализа необходимо рассмотреть подробнее примеры других уязвимостей и проанализировать, каким образом их количество в программном коде можно уменьшить на начальных этапах проектирования, т.е. с помощью средств статического анализа.

Описанное приложение требует отдельной проработки своей функциональности и добавления возможность поиска других видов уязвимостей. Однако реализованный функционал уже позволяет автоматизировать процесс анализа исходного кода программы и быстро получать результаты анализа. Добившись успехов в эффективном поиске ошибок безопасности в программном коде, можно начать работу над реализацией запуска анализатора в фоновом режиме.

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. Chess B., West J. Secure Programming with Static Analysis – 2007. – 618 с.
2. Область применения анализаторов кода [Электронный ресурс]. - 2017. - Режим доступа: http://www.controlengrussia.com/programmnye-sredstva/bezopasnost-programmnye-sredstva/appchecker/- Дата доступа: 07.12.2018
3. Static code analysis [Электронный ресурс]. - 2018. – Режим доступа: https://www.viva64.com/en/t/0046/ - Дата доступа: 04.12.2018
4. What is Static Code Analysis? [Электронный ресурс]. - 2018. - Режим доступа: https://www.perforce.com/blog/qac/what-static-code-analysis- Дата доступа: 05.12.2018
5. Обнаружение уязвимостей в теории и на практике, или почему не существует идеального статического анализатора [Электронный ресурс]. - 2018. - Режим доступа: https://habr.com/company/solarsecurity/blog/420337/- Дата доступа: 06.12.2018
6. Лекции портала “Информационно-коммуникационные технологии в образовании” про синтаксический анализ [Электронный ресурс]. - 2018. - Режим доступа: http://ict.edu.ru/ft/005128/ch6.pdf- Дата доступа: 06.12.2018
7. A study on the weakness analysis for binary code in embedded environments [Электронный ресурс]. - 2017. - Режим доступа: https://pdfs.semanticscholar.org/e3c3/78c2433bddb0e9c994144a6a49df23e8d9d9.pdf - Дата доступа: 15.03.2019
8. Exploit (computer security) [Электронный ресурс]. - 2019. - Режим доступа: https://en.wikipedia.org/wiki/Exploit_(computer_security) - Дата доступа: 16.03.2019
9. Program analysis and specialization for the C programming language [Электронный ресурс]. - 1994. - Режим доступа: https://www.cs.cornell.edu/courses/cs711/2005fa/papers/andersen-thesis94.pdf- Дата доступа: 14.03.2019

10. Static Analysis on Binary Code [Электронный ресурс]. - 2000. - Режим доступа: https://engineering.lehigh.edu/sites/engineering.lehigh.edu/files/_DEPARTMENTS/cse/research/tech-reports/2012/lu-cse-12-001.pdf-thesis94.pdf- Дата доступа: 11.03.2019

11. Minimizing code defects to improve software quality and lower development costs [Электронный ресурс]. - 2008. - Режим доступа: ftp://ftp.software.ibm.com/software/rational/info/do-more/RAW14109USEN.pdf- Дата доступа: 20.03.2019

12. Межпроцедурный статический анализ для поиска ошибок в исходном коде программ на языке C#[Электронный ресурс]. - 2018. - Режим доступа: http://www.ispras.ru/dcouncil/docs/diss/2017/koshelev/dissertacija-koshelev.pdf - Дата доступа: 12.04.2019

13. How Does Static Analysis Prevent Defects and Accelerate Delivery? [Электронный ресурс]. - 2018. - Режим доступа: https://dzone.com/articles/how-does-static-analysis-prevent-defects-and-accel- Дата доступа: 19.03.2019

14. Accelerating Automotive Software Safety with MISRA and Static Analysis[Электронный ресурс]. - 2018. - Режим доступа: http://blogs.grammatech.com/accelerating-automotive-software-safety-with-misra-and-static-analysis- Дата доступа: 18.03.2019

15. Common Vulnerabilities and Exposures [Электронный ресурс]. - 2019. - Режим доступа: https://ru.wikipedia.org/wiki/Common_Vulnerabilities_and_Exposures- Дата доступа: 20.04.2019

16. MISRA C:2012 Amendment 1 [Электронный ресурс]. - 2016. - Режим доступа: https://www.misra.org.uk/LinkClick.aspx?fileticket=V2wsZxtVGkE%3D&tabid=57- Дата доступа: 10.04.2019

Приложение А

ИСХОДНЫЙ КОД СТАТИЧЕСКОГО АНАЛИЗАТОРА

```
import re
methods_replacement_rule = {
    'strcpy': 'strcpy_s',
    'strcat': 'strcat_s',
    'sprintf': 'sprintf_s',
    'strlen': 'strlen_s',
    'printf': 'printf_s',
    'ctime': 'ctime_s',
    'rewind': 'fseek'
}

array_name_and_range = {}
var_types = ['char', 'int', 'uint8_t', 'uint16_t', 'uint32_t']

def deleted_pointer_warning(pointer, place, file):
    warning = "Variable " + pointer + " is used after deleting: " + place + "\n"
    file.write(warning)



def unsafe_functions_warning(unsafe_method, file, positions):
    if len(positions) == 1:
        pos_str = "position "
    else:
        pos_str = "positions "
    warning = "Method " + unsafe_method + " in " + pos_str + ",".join(map(str,
positions)) + \
              " is unsafe. You should better use " +
methods_replacement_rule[unsafe_method] + "\n"
    file.write(warning)
```

```
def buffer_overflow_warning(place, file):
    warning = "Buffer overflow: " + place + "\n"
    file.write(warning)


def find_unsafe_functions(data, result_file):
    for method in methods_replacement_rule:
        matches = [i.start() for i in re.finditer(method, data)]
        if len(matches):
            unsafe_functions_warning(method, result_file, matches)


def find_using_variables_after_deleting(data, result_file):
    r = re.search("((" + '|'.join(var_types) + ")\*\s[a-zA-Z_][a-zA-Z0-9_]*)", data)
    while r:
        # r_string = r.group(1)
        variable = r.group(0).split(" ")[1:][0]
        cut_data = data.split(r.group(0), 1)[1]
        data = cut_data
        r = re.search("delete\s*" + variable, data)
        if r:
            # r_string = r.group(0)
            cut_after_delete_data = cut_data.split(r.group(0), 1)[1]
            delete_variable = r.group(0).split(" ")[1:][0]
            if variable == delete_variable:
                r = re.search("(" + variable + "\s*=\s*)|(" + variable + "\s*\[[a-zA-Z_0-9]
[a-zA-Z0-9_]*\]\s*"
                                                    "=\s*[a-zA-Z_0-9][a-zA-Z0-9_]*)",
                              cut_after_delete_data)
                if r:
                    deleted_pointer_warning(variable, r.group(0), result_file)
        r = re.search("((" + '|'.join(var_types) + ")\*\s[a-zA-Z_][a-zA-Z0-9_]*)", data)
```

```
def find_direct_overflow(variable, max_range, data, result_file):
    r = re.search(variable + "\s*\[[a-zA-Z_0-9][a-zA-Z0-9_]*\]\s*\=\s*[a-zA-Z_0-9]
[a-zA-Z0-9_]*", data)
    while r:
        # r_string = r.group(0)
        var = r.group(0).split("[")[1]
        variable_range = var.split("]")[0]
        try:
            current_range = int(variable_range)
            if current_range >= int(max_range):
                buffer_overflow_warning(r.group(0), result_file)
        except ValueError:
            if array_name_and_range[variable_range] >= max_range:
                buffer_overflow_warning(r.group(0), result_file)

        cut_data = data.split(r.group(0), 1)[1]
        data = cut_data
        r = re.search(variable + "\s*\[[a-zA-Z_0-9][a-zA-Z0-9_]*\]\s*\=\s*[a-zA-
Z_0-9][a-zA-Z0-9_]*", data)


def find_out_of_range_arrays(data, result_file):
    r = re.search("((" + '|'.join(var_types) + ")\s[a-zA-Z_][a-zA-Z0-9_]*\[[0-9]*\];)|"
+
                                        "((" + '|'.join(var_types) + ")\s[a-zA-Z_][a-zA-
Z0-9_]*\s*=\s*[0-9])", data)
    while r:
        r_string = r.group(0)
        if r_string.find("[") != -1:
            var = r.group(0).split(" ")[1:][0]
            array_name = var.split("[")[0]
            array_range = (var.split("[")[1]).split("]")[0]
        else:
            var = r.group(0).split(" ")[1:][0]
            array_range = r.group(0).split("= ")[1]
```

```
            array_name = var.split(" ")[0]
            # array_range = range.split("= ")[1]

        array_name_and_range[array_name] = array_range
        cut_data = data.split(r.group(0), 1)[1]
        data = cut_data
        find_direct_overflow(array_name, array_range, data, result_file)
        r = re.search("((" + '|'.join(var_types) + ")\s[a-zA-Z_][a-zA-Z0-9_]*\
[[0-9]*\];)", data)



def analyze(data):
    file = "analyzer_report.txt"
    result_file = open(file, "w+")
    find_unsafe_functions(data, result_file)
    find_using_variables_after_deleting(data, result_file)
    find_out_of_range_arrays(data, result_file)
    result_file.close()

    review_code = open(file, "r").read()
    result_file.close()
    return review_code
```

Приложение Б

ИСХОДНЫЙ КОД ПОЛЬЗОВАТЕЛЬСКОГО ИНТЕРФЕЙСА СТАТИЧЕСКОГО АНАЛИЗАТОРА

```
import tkinter as tk

import analyzer

from tkinter import messagebox, filedialog


def show_error(error):
    messagebox.showinfo('Error', error)



def show_save_file_message(file_path):
    messagebox.showinfo('Information', 'Report was successfully saved in file: \n' +
file_path)



class MainAnalyzerWindow(tk.Frame):
    def __init__(self, *args, **kwargs):
        tk.Frame.__init__(self, *args, **kwargs)

        self.code_text_label = tk.Label(self, text='Code to analyze:')
        self.code_text_label.grid(row=0, column=0)

        self.ok_button = tk.Button(self, text="Choose file to analyze",
command=self._get_file)
        self.ok_button.grid(row=0, column=1)

        self.code_text = tk.Text(self)
        self.code_text.grid(row=1, column=0)

        self.review_text_label = tk.Label(self, text='Review:')
        self.review_text_label.grid(row=0, column=2)
```

```
        self.review_text = tk.Text(self)
        self.review_text.grid(row=1, column=2)

        self.analyze_button = tk.Button(self, text="Analyze",
command=self._send_analyze_command)
        self.analyze_button.grid(row=6, column=0)

        self.save_report_button = tk.Button(self, text="Save report",
command=self._send_save_report_command)
        self.save_report_button.grid(row=6, column=2)



    def _send_analyze_command(self):
        text = self.code_text.get(1.0, tk.END)
        self._on_analyze_code(text)



    def _get_file(self):
        code = ""
        file = filedialog.askopenfilename()
        if file:
            with open(file) as file:
                code = file.read()
        self.code_text.insert(tk.END, code)



    def _on_analyze_code(self, text):
        try:
            result = analyzer.analyze(text)
            self.review_text.delete(1.0, tk.END)
            self.review_text.update()

            self.review_text.insert(tk.END, result)
            self.review_text.update()
        except Exception as e:
            show_error(str(e))
```

```
    def _send_save_report_command(self):
        self.review_text.update()
        filename = filedialog.asksaveasfile(defaultextension='.txt')
        file = open(filename.name, "w")
        file.write(self.review_text.get("1.0", tk.END))
        if file:
            show_save_file_message(filename.name)
        file.close()


if __name__ == "__main__":
    root = tk.Tk()
    root.title('Static analyzer tool')
    root.geometry('1310x500')
    main = MainAnalyzerWindow(root)
    main.pack(side="top", fill="both", padx=10, expand=True)
    root.mainloop()
```