МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РЕСПУБЛИКИ КАЗАХСТАН
Некоммерческое акционерное общество
«АЛМАТИНСКИЙ УНИВЕРСИТЕТ ЭНЕРГЕТИКИ И СВЯЗИ ИМЕНИ ГУМАРБЕКА ДАУКЕЕВА»
Институт Систем Управления и Информационных Технологий
Кафедра«Системы информационной безопасности»

«ДОПУЩЕН К ЗАЩИТЕ»
Зав.кафедрой _____
(ученая степень, звание, Ф.И.О.)
_____ «_____» _____20____г.
(подпись)

ДИПЛОМНЫЙ ПРОЕКТ

На тему: Применение инструментальных средств анализа исходных текстов для выявления уязвимостей и закладок программного обеспечения

Специальность: Системы Информационной Безопасности

Выполнил(а) Абдуллаева Муяссар Суратовна (Ф.И.О.) Группа СИБ-16-2

Научный руководитель к.т.н., доцент Шайкулова Актоты Алиевна
(ученая степень, звание, Ф.И.О.)

Консультанты:
по специальной части:
старший преподаватель Дмитриева Маргарита Валерьевна

_____ «_____» _____20____г.
(подпись)

по безопасности жизнедеятельности:
к.т.н., доцент Приходько Николай Георгиевич

_____ «_____» _____20____г.
(подпись)

Нормоконтролер: старший преподаватель Дмитриева Маргарита Валерьевна
(ученая степень, звание, Ф.И.О.)

_____ «_____» _____20____г.
(подпись)

Рецензент: к.т.н., доцент Айтхожаева Евгения Жамалхановна
(ученая степень, звание, Ф.И.О.)

_____ «_____» _____20____г.
(подпись)

Алматы 2020

Задание на выполнение дипломного проекта

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РЕСПУБЛИКИ КАЗАХСТАН
Некоммерческое акционерное общество
«АЛМАТИНСКИЙ УНИВЕРСИТЕТ ЭНЕРГЕТИКИ И СВЯЗИ ИМЕНИ ГУМАРБЕКА ДАУКЕЕВА»

Институт Систем Управления и Информационных Технологий

Кафедра «Системы информационной безопасности»

Специальность «Системы информационной безопасности»

ЗАДАНИЕ
на выполнение дипломного проекта

Студенту Абдуллаевой Муяссар Суратовне
(Ф.И.О.)

Тема проекта «Применение инструментальных средств анализа исходных текстов для выявления уязвимостей и закладок программного обеспечения»

Утверждена приказом по университету № _____ от «___» _____2020 г.

Срок сдачи законченного проекта «____» _____ 2020 г.

Исходные данные к проекту (требуемые параметры результатов исследования (проектирования) и исходные данные объекта):Исходные коды проверяемых программ, среда разработки (Visual Studio, Intellij Idea и др.), установленный плагин PVS – Studio.

Перечень вопросов, подлежащих разработке в дипломном проекте, или краткое содержание дипломного проекта:

Перечень графического материала (с точным указанием обязательных чертежей):68 иллюстраций и 6 таблиц

Основная рекомендуемая литература: Аветисян А.И. Современные методы статического и динамического анализа программ для решения приоритетных проблем программной инженерии: автореф. дис. д-ра физ.-мат. наук: 05.13.11 / Аветисян Арутюн Ишханович. М., 2011., Фаулер, Кент. Рефакторинг. Улучшение проекта существующего кода. –Лондон:Вильямс, 2017.

Консультации по проекту с указанием относящихся к ним разделов проекта

Раздел	Консультант	Сроки	Подпись
Анали рисков информационной безопасности	старший преподаватель Дмитриева Маргарита Валерьевна	17.02.2020 – 09.05.2020	
Безопасность жизнедеятельности	к.т.н. доцент Приходько Николай Георгиевич	17.02.2020 – 09.05.2020	

График
подготовки дипломного проекта

Наименование разделов, перечень разрабатываемых вопросов	Сроки представления научному руководителю	Примечание
Поиск уязвимостей в программах	17.02.2020 – 20.02.2020	
Обзор существующих анализаторов	21.02.2020 – 28.02.2020	
Изучение методов сертификации программного обеспечения на отсутствие уязвимостей	01.03.2020 – 08.03.2020	
Анализ основных подходов к исследованию программ	09.03.2020 - 18.03.2020	
Изучение основ работы с PVS-Studio на Windows	19.03.2020 – 27.03.2020	
Системные требования и установка PVS-Studio	28.03.2020 - 07.04.2020	
Работа со списком диагностических сообщений	08.04.2020 - 18.04.2020	
Изучение диагностики анализатора	19.04.2020 - 30.04.2020	
Анализ рисков ИБ. БЖД	01.05.2020 - 09.05.2020	

Дата выдачи задания «_____»_____20___г.

Заведующий кафедрой _____(Бердибаев Рат Шындалиевич)
(подпись) (ФИО)

Научный руководитель
проекта _____ (Шайкулова Актоты Алиевна)
(подпись) (ФИО)

Задание принял к
исполнению студент _____ (Абдуллаева Муяссар Суратовна)
(подпись) (ФИО)

Аннотация

Целью данного дипломного проекта является определение потенциально опасных возможностей, скрытых в коде ПО и проведение анализа программных средств на выявления уязвимостей и закладок программного обеспечения.

Акцент в данной работе делался на защиту программного обеспечения во время разработки, то есть анализ и проверка исходных кодов программ для обеспечения их безопасности после сборки. Для анализа кодов было выбрано инструментальное средство – PVS – Studio.

Входные данные работы – исходные коды программных обеспечений для операционной системы Windows, написанные на языке программирования C#.

Аңдатпа

Бұл дипломдық жобаның мақсаты БҚ кодында жасырын ықтимал қауіпті мүмкіндіктерді анықтау және бағдарламалық қамтамасыз етудің осалдықтары мен бетбелгілерін анықтау үшін бағдарламалық құралдарға талдау жүргізу болып табылады.

Бұл жұмыста басты назар әзірлеу кезінде бағдарламалық қамтамасыз етуді қорғауға, яғни құрастырудан кейін олардың қауіпсіздігін қамтамасыз ету үшін бағдарламалардың бастапқы кодтарын талдау мен тексеруге аударылды. Кодтарды талдау үшін аспаптық құрал – PVS – Studio таңдалған.

Жұмыстың кіріс мәліметтері-C#бағдарламалау тілінде жазылған Windows операциялық жүйесіне арналған бағдарламалық қамтамасыз етудің бастапқы кодтары.

Abstract

The purpose of this diploma project is to identify potentially dangerous features hidden in the SOFTWARE code and analyze software tools to identify vulnerabilities and software bookmarks.

The focus of this paper was on software protection during development, i.e. analysis and verification of software source codes to ensure their security after Assembly. A tool called PVS – Studio was selected for code analysis.

The input data of the work is the source code of software for the Windows operating system, written in the C# programming language.

Содержание

Введение

В настоящее время развитие информационных технологий привело к пересмотру требований к надежности, надежности и безопасности программного обеспечения. Одной из важнейших предпосылок успешной работы практически важных объектов компьютеризации является защита используемого в ее составе программного обеспечения от возможных сбоев, а также его корректное функционирование во всех наборах входных данных. Ошибки в автоматизированных системах в значительной степени обусловлены уязвимостями (ошибками, ошибками) в программном обеспечении, в то время как они могут привести к сбою важных аппаратных и программных компонентов.

Задачи анализа исходного кода программного обеспечения можно разделить на два подхода: ручной (экспертный) анализ и автоматизированный (приближенный) анализ. Ручной анализ проводится высококвалифицированными специалистами, которые полагаются только на свои знания и опыт, в то же время этот метод имеет высокую стоимость и стоит очень дорого. Массовое использование ручного метода анализа не наблюдается по следующим причинам: усложнение аппаратной архитектуры компьютеров, увеличение исходного кода программ до ста мегабайт, появление программных слоев между операционной системой и программным обеспечением. При типичном подходе деятельность специалистов в связи с использованием инструментария облегчается применением статического анализа структуры (исходного кода) исследуемой программы к потенциально опасным конструкциям языка программирования (уязвимостям) в программном обеспечении.

В данном дипломном проекте показаны различные инструментальные средства, которые рекомендуются для использования при выявлении уязвимостей и закладок ПО. Наилучшим из них был выбран универсальный статический анализатор PVS – Studio, который работает как плагин и добавляется в функционал среды разработки, в данном случае это Visual Studio. PVS – Studio имеет широкий функционал, и помогает специалистам выявить как элементарные ошибки, дефекты, опечатки кода, так и наиболее опасные уязвимости, которые могут привести к наихудшим последствиям таким как аварийное завершение системы, атаки на наиболее важные аппаратные компоненты системы, потеря и распространение конфиденциальных данных и особо важной информации. Основным преимуществом использования статических анализаторов является снижение затрат на устранение дефектов в программе. Таким образом, согласно статистике, представленной McConnel в книге «Code Complete», исправление ошибки на этапе тестирования обходится в десять раз дороже, чем на этапе кодирования.

1 Поиск уязвимостей в программах

1.1 Классификация уязвимостей защиты

Уязвимости программного обеспечения – ошибки программистов на стадии разработки программного обеспечения. Они позволяют хакерам получить нелегальный доступ к функциям программы или к хранящимся в ней данным. Дефекты могут появиться на любом этапе жизненного цикла, от проектирования до выпуска готовой продукции.

При нарушении требования корректной работы программы на всех возможных входных данных могут появиться так называемые уязвимости безопасности. Уязвимости безопасности могут привести к тому, что одна программа может быть использована для преодоления ограничений безопасности всей системы в целом.

Классификация уязвимостей системы безопасности на основе программных ошибок:

1 Переполнение буфера (bufferoverflow). Эта ошибка возникает из-за отсутствия контроля над выходным полем в памяти во время выполнения программы. Когда слишком большой пакет данных переполняет кэш ограниченного размера, содержимое внешних ячеек памяти перезаписывается, и программа аварийно завершает работу.

2 Уязвимости (taintedinputvulnerability). Ошибки могут возникать, когда вводимые пользователем данные передаются интерпретатору, внешнему языку (обычно Unixshell или SQL) без достаточного контроля. В этом случае пользователь может настроить входные данные таким образом, что запущенный интерпретатор выполняет совершенно другую команду, которая идентифицирует авторов уязвимой программы.

3 Ошибки форматных строк (format string vulnerability). Данный тип уязвимостей защиты является подклассом уязвимости . Это происходит из-за недостаточного контроля параметров при использовании printf, fprintf, scanf и др. Эти функции принимают в качестве одного из параметров строку символов, которая определяет формат ввода или вывода последующих функциональных аргументов. Если пользователь сам может установить тип форматирования, то эта ошибка может возникнуть из-за неудачного применения функций форматирования строк.

4 Уязвимости, вызванные ошибками синхронизации (raceconditions). Проблемы, связанные с многозадачностью, приводят к так называемой ситуации: программа, которая не предназначена для работы в многозадачной среде, может полагать, что, например, файлы, которые она использует, не могут быть изменены другой программой. В результате злоумышленник, который вовремя заменяет содержимое этих рабочих файлов, может заставить программу выполнить определенные действия.

5 Слабая криптография. Решение: использование по возможности краткосрочных (сеансовых) ключей. Аутентификационные данные должны храниться строго централизованно, а обрабатываться – локально. В случае

возникновения задачи обмена ключами по небезопасному каналу – использовать ассиметричную криптографию, передавая таким способом лишь ключи к симметричному шифру. В случае оборота ЭД, использовать механизм ЭЦП.

6 Универсальная защита конфиденциальных данных. Проблема: её нет. Решение: шифрование данных мощным симметричным алгоритмом (AES, RC6, Blowfish, 3DESи т.д.), хранение пароля в надёжном разделе реестра (в случае несистемного использование – вообще нехранение пароля!), требование ввода пароля, защита списками ACLфайла и раздела реестра.

7 Опасность входных данных (SQL-injection, XSSи многое другое). Проблема: очевидна. Решение: особое внимание всякого рода регулярным выражениям; строгий контроль корректности и длины входных данных и их фильтрация. Указывайте полные пути в адресах, представленные в канонической форме.

8 Опасности работы с БД. Решения: а) Использование параметризованных запросов к БД, отсутствие конструирования запросов внутри приложения; жёсткий контроль корректности отправляемых запросов; б) не подключаться к БД от имени system; г) Используйте безопасно хранимые процедуры.

9 Защита от XSS-атак: а) кодирование выходных данных; б) использование двойных кавычек во всех атрибутах тэга; в) Как можно чаще используйте innerText; г) Используйте только одну кодовую страницу.

Уязвимости появляются вследствие добавления в состав ПО сторонних компонентов или свободно распространяемого кода (open source).

В зависимости от стадии появления этот вид угроз делится на уязвимости проектирования, реализации и конфигурации.

Ошибки, допущенные при проектировании, сложнее всего обнаружить и устранить. Это – неточности алгоритмов, закладки, несогласованности в интерфейсе между разными модулями или в протоколах взаимодействия с аппаратной частью, внедрение неоптимальных технологий. Их устранение является весьма трудоемким процессом, в том числе потому, что они могут проявиться в неочевидных случаях – например, при превышении предусмотренного объема трафика или при подключении большого количества дополнительного оборудования , что усложняет обеспечение требуемого уровня безопасности и ведет к возникновению путей обхода межсетевого экрана.

Уязвимости реализации появляются на этапе написания программы или реализации в ней алгоритмов безопасности. Это неправильная организация вычислительного процесса, синтаксические и логические ошибки. Однако существует риск, что ошибка приведет к переполнению буфера или другим проблемам. Их обнаружение занимает много времени, а устранение предполагает ремонт определенных частей механизма кода.

Ошибки конфигурации аппаратного и программного обеспечения очень распространены. Распространенными причинами являются недостаточное качество разработки и отсутствие тестов для правильного функционирования дополнительных функций. Эта категория также включает слишком простые пароли и учетные записи по умолчанию, которые остаются неизменными. Согласно статистике, уязвимости чаще всего встречаются в популярных и распространенных продуктах - настольных и мобильных операционных системах, браузерах [1].

1.2 Способы выявления уязвимостей

Есть два базовых способа обнаружения уязвимостей и закладок кода. Это структурный статический и динамический анализ исходного кода, который регулируется РД. И, сигнатурно - эвристический анализ потенциально опасных операций, сканирование программного кода на наличие таких операций и последующий ручной или автоматический анализ фрагмента кода для определения реальной угрозы программному обеспечению [2].

Второй способ лишен недостатков" измерительного проектирования " - перегрузки всего программного обеспечения и полного тестирования содержимого. Время" ручного " анализа исходного текста сокращается в десять-двадцать раз, так как количество потенциально опасных операций, как правило, не превышает 5-10% объема программного обеспечения. Что касается статистического и динамического анализа, следует отметить, что результаты статистического анализа сравниваются с исходными текстами по сложности интерпретации, а динамический анализ дополнительно требует разработки и внедрения соответствующих маршрутных тестов. Таким образом, сигнальный и эвристический анализ сокращает время поиска недекларированных возможностей в десятки раз. Время - немаловажный фактор оценивания анализа и проверки.

Таблица 1 показывает, что для определения уязвимостей и закладок кода программных обеспечений необходим комплексный подход. С использованием критериев уязвимости и вероятности осуществления ее последствий можно выработать предложения по уровням контроля уязвимости кода для использования программного обеспечения в системах определенного класса.

Этапы тестирования кода на наличие уязвимостей:

1 Формирование программной политики безопасности, которая может касаться технических условий для объекта информатизации;

2 Сигнатурно-эвристический анализ исходного и выполняемого кода на предмет потенциально опасных операций и ошибок кодирования;

3 Анализ подсистем безопасности (контроль подсистем защиты пароля, подсистемы авторизации, подсистемы аутентификации) и др.;

4 Функциональные и стрессовые, тестирования на нагрузку и производительность системы;

5 Структурный анализ распределения и контроль целостности;

6 Анализ скрытых каналов;

7 Анализ открытых каналов;

8 Ограничение использования продукта в соответствии с политикой безопасности;

9 Формирование условий обновления и модификации политики безопасности.

Таблица 1.1 – Комплексный подход для выявления уязвимостей

Уязвимость кода	Метод тестирования	Комментарии
Ошибки, возникшие в случае применения редко используемых входных данных	Функциональное тестирование на редко используемых входных данных	Проверка декларированных в документации функциональных возможностей для редко используемых входных данных
Недекларированные входные параметры и режимы, связанные с деградацией производительности системы	Профилирование производительности	Выявление фрагментов кода, в которых возникает деградация производительности
Ошибки, связанные с отказом в обслуживании	Нагрузочное и стрессовое тестирование	Создание специфических условий для работы программы с большим объемом входных данных, при увеличенной нагрузке на процессор, в условиях минимально допустимых аппаратных ресурсов и др.
Уязвимости подсистемы парольной защиты и других подсистем	Тестирование подсистемы аутентификации, управления доступом и других подсистем безопасности	Определение режимов управления безопасностью, трассировка парольной системы, поиск встроенных паролей и др.

Продолжение таблицы 1.1

Уязвимость кода	Метод тестирования	Комментарии
Программные закладки	Сигнатурно – эвристический анализ	Выявление программных закладок по сигнатурам потенциально опасных операций и по событиям
Некорректности кодирования	Сигнатурно – эвристический анализ	Выявление программных закладок по сигнатурам конструкций
Хулиганский код , «подписи программистов»	Сигнатурно – эвристический анализ	Выявлений программных закладок по осмысленным сообщениям, константам, идентификаторам, фактам скрытия «горячих» клавиш и т.д.
Скрытые каналы	Анализ трафика и памяти в изолированной среде	Мониторинг и аудит журналов
Программные закладки, ниициируемые скрытым переходом	Контроль над избыточностью, сигнатурно – эвристический анализ	Контроль над избыточностью и выявление программных закладок по конструкциям скрытой передачи управления

1.3 Статический анализатор кода для выявления уязвимостей и закладок ПО

Меры обеспечения качества кода, в зависимости от требований конкретного проекта, могут включать мероприятия, предусмотренные стандартами в данной области, например:

- надлежащее документирование и учет изменений в коде;
- управление версиями (репозиторий кода);
- функциональное тестирование (на соответствие документации);
- тестирование по методам "черного", "серого", "белого" ящиков;
- верификация соответствия кода функциональным спецификациям;

– тестирование исполнимого кода сканерами уязвимости;
– анализ качества исходных текстов программ;
– сертификация кода по требованиям информационной безопасности [3].

В настоящее время активно развивается технологическое направление, связанное с автоматизацией анализа исходных кодов программ. Соответствующие средства, называемые статическими анализаторами, обладают следующими достоинствами:

– независимостью от функциональной специфики программ, что позволяет создавать универсальные анализаторы, направленные на выявление широкого спектра уязвимостей;
– высоким уровнем абстракции описания ветвей алгоритмов и функциональных объектов, отсутствием необходимости задания исходных (входных) данных анализируемых программ;
– стандартизованными способами описания и распространения сведений об уязвимостях, что повышает качество (полноту) контроля и снижает их реактивность (время, проходящее от момента выявления уязвимости до включения в БД анализатора описания уязвимости);
– высоким уровнем масштабируемости контролирующих процедур, которые могут применяться как в отношении отдельных фрагментов программы (вплоть до одной строки кода), так и в отношении сложных комплексов программ;
– минимальными ограничениями, налагаемыми на процесс разработки и тестирования программ.

Разберем принципы работы данного анализатора, совмещающего результаты синтаксического и семантического анализа кода.

Результаты объединяются путем сопоставления позиций элементов с позициями шаблонов (шаблонов) конструкций, возникающих при анализе подписи, для иерархического представления программы, полученной в результате лексического и синтаксического анализа. Зная позицию каждого элемента (в исходных файлах), эксперт может сравнивать абстрактные синтаксические деревянные узлы с обнаруженными сигналами и оценивать возможность обнаружения потенциально опасных конструкций на фрагментах этих исходных кодов с помощью логических правил инференции.

Для реализации указанных принципов был предложен анализатор безопасности кода со следующей структурой: CSA=áSDB, HDB, MLA, MSA, MSY, MLO, MRPñ, где SDB – БД сигнатур (паттернов) ПОК; HDB – БД структурной информации о коде (иерархического представления программы); MLA – программный модуль лексического анализа; MSA – программный модуль синтаксического анализа; MSY – программный модуль сигнатурного анализа; MLO – программный модуль логического вывода; MRP – программный модуль построения отчетов.

Рассмотрим алгоритм работы анализатора безопасности кода. Модули лексического и сигнатурного анализа принимают на вход исходные тексты программ, их бинарный код, а также общие сведения о декомпозиции ПО. На выходе генерируется отчет о ПОК, состоящий из перечня участков кода, где они предположительно находятся, и необходимые комментарии.

Для классификации и группировки различных видов потенциально опасных конструкций используется международная таксономия CWE (Common Weakness Enumeration), где наиболее полно перечислены дефекты в области безопасности ПО [4].

Задача синтаксического анализатора заключается в том, чтобы на основе цепочки распознанных лексем сформировать в базе структурной информации дерево структурной декомпозиции программы, одной из ветвей которого является абстрактное синтаксическое дерево (дерево Канторовича [5]). Данный модуль, как правило, содержит в себе конечный автомат для распознавания цепочек лексем, который может быть реализован, к примеру, с помощью LR(1)-парсера, то есть синтаксического анализатора, выполняющего восходящий синтаксический разбор с предосмотром текста на один символ вперед. Надо сказать, что компилятор любого языка программирования содержит подобный модуль синтаксического разбора. К сожалению, большинство из этих модулей замкнуто и не позволяет выдать результат своей работы в унифицированном для обмена с другими программами виде. Кроме того, у средств аудита безопасности кода несколько иные требования к синтаксическому анализатору, например, он должен быть более толерантным к структуре исходных текстов и, встретив незнакомую лексему, попытаться продолжить анализ файла насколько это возможно, а не остановиться, выдав ошибку, как это сделает синтаксический анализатор из состава компиляторов.

Сигнатурный анализатор осуществляет сквозной поиск в исходных текстах программы паттернов ПОК, используя базу сигнатур, которая представляет собой набор следующих кортежей: SDB=áPE, DLñ, где PE – описание фрагмента ПОК (например, используя регулярные выражения); DL – оценка степени их опасности (например числовая шкала 1–10).

Модуль логического вывода сравнивает структурную информацию кода и данные анализа подписи, на основе которой выводится о возможности ПОК в определенном разделе кода.

Статические анализаторы могут использоваться на всех этапах разработки программного обеспечения, поэтому они могут упорядочить разработчика и снизить затраты, выявив уязвимости на предыдущих этапах разработки.

Они могут быть использованы в качестве основы для интегрированной системы оценки качества кода, т. е. их соответствие конкретным проектам стандартов организации, степень аутентичности кода (например, степень получения кода из открытых источников). Таким образом, анализаторы статического кода могут служить основой для всестороннего аудита качества

программных продуктов. Перспективы развития статистических аналитических технологий определяются следующими факторами:

– широкое распространение кода с открытыми лицензиями, относительно которого отсутствуют юридические гарантии безопасности, подтвержденные независимыми экспертами;

– значительный объем корпоративного сектора разработки специализированного ПО (бизнес-приложений). Такие приложения в подавляющем числе случаев разрабатываются в строго ограниченные сроки и в значительной степени ориентированы на функциональные требования (зачастую в ущерб качеству и безопасности). Характерно периодическое внесение изменений в такие приложения на протяжении жизненного цикла. Большинство компаний-разработчиков на данном рынке не имеют в штате специалистов, обладающих должным уровнем знаний и квалификации в области анализа уязвимостей кода;

– миграция корпоративных ресурсов, разного рода платформ и инфраструктур в область облачных сервисов. В ряде случаев, учитывая высокую стоимость традиционных (необлачных) платформ, для небольших компаний-разработчиков кода может оказаться дешевле приобретение не платформы анализатора, как таковой, а соответствующей "услуги". Кроме того, эксплуатация средств анализа кода потребует соответствующего сопровождения потребителем, которое, впрочем, может оказаться не определяющим фактором;

– значимость угроз внесения в код так называемых программных закладок (или недекларированных функций). На текущий момент противодействие подобного рода уязвимостям реализуется только для систем специального назначения в рамках его сертификации или тематических исследований в организациях, имеющих соответствующую аккредитацию ФСТЭК или ФСБ России.

Практически все доступные на текущий момент статические анализаторы используют в том или ином виде следующие технологии выявления уязвимостей в коде:

– структурный анализ текстов программ – основывается на задании правил построения законченных и семантически целостных фрагментов кода, на соответствие которым оцениваются исходные тексты программ. Для задания указанного множества правил используются контекстно-свободные грамматики, БНФ-нотация, и другие формальные методы описания языков программирования. Данный метод обладает наивысшим уровнем абстракции и может использоваться для выявления широкого спектра уязвимостей. Его недостатком является относительно высокая вероятность ошибок классификации, включая ошибки первого (пропуск уязвимости) и второго (ложные выявления уязвимостей) рода;

– структурная оценка сложности исходных текстов основывается на построении моделей покрытия кода, относительно которых могут оцениваться, например, межмодульная связность (маршруты, связывающие

функциональные объекты по данным, по управлению), сложность покрытия кода (например, на основе оценки цикломатического числа в графе покрытия кода). Метод может использоваться для оценки качества кода в целом (то есть качества кодирования алгоритмов);

– сигнатурный анализ кода – основывается на ранее имевшемся опыте выявления уязвимостей, сведенном в базу сигнатур уязвимостей. Сигнатура определятся заданием минимального множества фрагмента кода, по которым может быть идентифицирована уязвимость. Для описания сигнатур уязвимостей используются параметризованные шаблоны (например, регулярные выражения). Достоинство сигнатурного метода состоит в его относительно высокой точности.

В зависимости от классов контролируемых уязвимостей все анализаторы уязвимостей можно отнести к одному из следующих типов:

– анализаторы для выявления специальных классов ошибок работы с системными ресурсами (использование динамической памяти, синхронизация многопоточных вычислений и пр.). К числу таковых относится, например, анализатор PVS-Studio;

– анализаторы общего назначения для выявления большинства типов ошибок, как правило, поддерживающие множество платформ разработки (Coverity SAVE и Klocwork);

– анализаторы для выявления недекларированных функций программного кода, например анализаторы (АК-ВС и Appercut CCS).

– В зависимости от специализации оцениваемых программных средств анализаторы кода можно отнести к одному из следующих типов:

– анализаторы, ориентированные на поиск уязвимостей в бизнес-приложениях и функциональном ПО для ERP-систем (анализатор ERPscan для платформы SaaS);

– анализаторы, предназначенные для поиска уязвимостей в общесистемном ПО, распространяемом по открытым лицензиям (анализаторы Coverity, Klocwork);

– специализированные анализаторы, предназначенные для поиска уязвимостей в программном коде Web-приложений (например, анализатор IBM Rational APP Scan, APPS).

1.4 Обзор существующих анализаторов

В настоящее время разработано большое количество инструментальных средств, предназначенных для автоматизации поиска уязвимостей защиты программ на языках Си и Си++. В данном разделе мы рассмотрим наиболее распространённые инструментальные средства.

По виду использования инструментальные средства можно разделить на два типа: инструменты, добавляющие дополнительные динамические проверки в программу и инструменты только статического анализа программ. В нашей работе мы рассмотрим инструменты, которые выявляют уязвимости защиты с помощью статического анализа программ.

Для обнаружения уязвимостей защиты в программах применяют следующие инструментальные средства:

Динамические отладчики. Инструменты, которые позволяют производить отладку программы в процессе её исполнения.

Статические анализаторы (статические отладчики). Инструменты, которые используют информацию, накопленную в ходе статического анализа программы.

Статические анализаторы указывают на те места в программе, в которых возможно находится ошибка. Эти подозрительные фрагменты кода могут, как содержать ошибку, так и оказаться совершенно безопасными [4].

Далее предложен обзор нескольких существующих статических анализаторов.

BOON. Основанный на глубоком семантическом анализе инструмент Boon, который автоматизирует процесс сканирования исходного кода на si для поиска уязвимостей, приводящих к переполнению буфера. Он определяет возможные дефекты, некоторые значения являются частью неопределенного типа буфера с определенным размером.

CQual. CQual-C-это аналитический инструмент для выявления ошибок в программах. Программа расширяет язык БД с помощью дополнительных пользовательских спецификаторов. Программист объясняет свою программу с помощью соответствующих спецификаторов и проверяет наличие ошибки cqual. Неверные замечания отражают возможные ошибки. Cqual можно использовать для определения потенциальной уязвимости строки.

MOPS. MOPS (MOdelcheckingProgramsforSecurity) - инструмент для поиска уязвимостей в защите Си-программ. Цель инструментального средства: динамическая коррекция программного кода на Си, чтобы убедиться, что программа соответствует статической модели. MOPS использует модель аудита программного обеспечения, соответствует ли приложение набору правил, определенных для создания безопасных программ.

ITS4, RATS, PScan, Flawfinder. Для поиска ошибок переполнения буфера и ошибок форматных строк используют следующие статические анализаторы:

ITS4. Простой инструмент, который статически просматривает исходный Си/Си++-код для обнаружения потенциальных уязвимостей защиты. Он отмечает вызовы потенциально опасных функций, таких, например, как strcpy/memcpy, и выполняет поверхностный семантический анализ, пытаясь оценить, насколько опасен такой код, а так же дает советы по его улучшению.

RATS. Утилита RATS (RoughAuditingToolforSecurity) обрабатывает код, написанный на Си/Си++, а также может обработать еще и скрипты на Perl, PHP и Python. RATS просматривает исходный текст, находя потенциально опасные обращения к функциям. Цель этого инструмента - не окончательно найти ошибки, а обеспечить обоснованные выводы, опираясь

на которые специалист сможет вручную выполнять проверку кода. RATS использует сочетание проверок надежности защиты от семантических проверок в ITS4 до глубокого семантического анализа в поисках дефектов, способных привести к переполнению буфера, полученных из MOPS.

PScan. Сканирует исходные тексты на Си в поисках потенциально некорректного использования функций, аналогичных printf, и выявляет уязвимые места в строках формата.

Flawfinder. Как и RATS, это статический сканер исходных текстов программ, написанных на Си/Си++. Выполняет поиск функций, которые чаще всего используются некорректно, присваивает им коэффициенты риска (опираясь на такую информацию, как передаваемые параметры) и составляет список потенциально уязвимых мест, упорядочивая их по степени риска.

Все эти инструменты схожи и используют только лексический и простейший синтаксический анализ. Поэтому результаты, выданные этими программами, могут содержать до 100% ложных сообщений.

Bunch. Bunch - средство анализа и визуализации программ на Си, которое строит граф зависимостей, помогающий аудитору разобраться в модульной структуре программы.

UNO. UNO - простой анализатор исходного кода. Он предназначен для обнаружения ошибок, таких как неизвестные переменные, нулевые указатели и вне массива. UNO позволяет легко анализировать поток управления и потоки данных, проводить внутренний и междисциплинарный анализ для описания характеристик пользователя. Но этот инструмент не был предназначен для анализа конкретных приложений, не поддерживает многие стандартные библиотеки и не позволяет анализировать важные программы на данном этапе разработки.

FlexeLint (PC-Lint). FlexeLint (PC-Lint) - этот анализатор предназначен для анализа исходного кода с целью выявления различных ошибок. Программа выполняет семантический анализ исходного кода, анализ потоков данных и управление ими. В конце работы даются некоторые основные сообщения:

- Может быть нулевой индикатор;
- Проблемы с выделением памяти (например, free to malloc ());
- Поток управления проблемами (например, недоступный код);
- Буфер может быть переполнен, арифметика переполнена;
- Предупреждение о плохом и потенциально опасном фрагменте кода.

Viva64. Инструмент Viva64, который помогает специалисту отслеживать в исходном коде Си/Си++-программ потенциально опасные фрагменты, связанные с переходом от 32-битных систем к 64-битным. Данный продукт очень популярен своей технической поддержкой. Viva64 встраивается в среду MicrosoftVisualStudio 2005/2008 и в другие среды разработки в зависимости от языка программирования , что способствует

удобной работе с этим инструментом. Анализатор помогает писать корректный и оптимизированный код для 64-битных систем.

Parasoft C++ Test. Parasoft C++ Test - Специальный инструмент для Windows, который позволяет автоматизировать анализ качества кода на языке C++. Тестовый пакет C++анализирует проект и создает код для проверки компонентов, захваченных проектом. Тестовый пакет C++очень важен для анализа классов C++. После загрузки проекта необходимо настроить методы тестирования. Программное обеспечение проверяет каждый аргумент метода и возвращает соответствующие типы значений. Вы можете определить тестовые данные для пользовательских представлений и классов. По умолчанию можно переопределить используемые аргументы теста C++и указать значения, полученные из теста. Особого внимания заслуживает возможность тестирования незаконченного C++тестового кода. Программное обеспечение выпускает код для любого метода и функции, которые еще не существуют. Поддержка моделирования внешних устройств и входящих данных, задаваемых пользователем. И та и другая функции допускают возможность повторного тестирования. После определения тестовых параметров для всех методов пакет C++Test готов к запуску исполняемого кода. Пакет генерирует тестовый код, вызывая для его подготовки компилятор Visual C++. Возможно формирование тестов на уровне метода, класса, файла и проекта.

Coverity. Инструменты Coverity используются для обнаружения и исправления недостатков безопасности и качества в опасных областях применения. Технология Coverity устраняет барьеры для сложных записей и реализаций, автоматизируя поиск и устранение критических программных ошибок и уязвимостей в процессе разработки. Компания способна обрабатывать десять миллионов дорожных кодов с минимальными положительными ошибками и обеспечивает 100-процентное покрытие маршрута Coverity.

KlocWork K7. Продукты Klocwork используются для автоматического анализа статического кода, выявления и предотвращения ошибок программного обеспечения и проблем безопасности. Средства этой компании служат для выявления недостатков качества и безопасности программного обеспечения, отслеживания и устранения недостатков в процессе разработки.

Frama-C. Frama-C - Открытый, интегрированный набор инструментов для анализа исходного кода на языке Си. ACSL (ANSI / ISO C SpecificationLanguage) - это специальный язык, который позволяет более точно описать особенности функции si, такие как диапазон допустимых входных значений функции и диапазон нормальных выходных значений.

Этот инструментарий помогает производить такие действия:

- Осуществлять формальную проверку кода;
- Искать потенциальные ошибки исполнения;
- Произвести аудит или проверку кода;

– Проводить реверс-инжиниринг кода для улучшения понимания структуры;

– Составление официальной документации.

CodeSurfer. CodeSurfer - инструмент анализа программ, который не предназначается непосредственно для поиска ошибок уязвимости защиты. Его основными достоинствами являются:

Анализ указателей;

Различные анализы потока данных (использование и определение переменных, зависимость данных, построение графа вызовов);

Скриптовый язык.

CodeSurfer может быть использован для поиска ошибок в исходном коде, для улучшения понимания исходного кода, и для реинженерии программ. В рамках среды CodeSurfer велась разработка прототипа инструментального средства для обнаружения уязвимостей защиты, однако разработанное инструментальное средство используется только внутри организации разработчиков.

FxCop. FXCop предоставляет средства автоматической проверки .NET-сборок на предмет соответствия правилам Microsoft .NET FrameworkDesignGuidelines. Откомпилированный код проверяется с помощью механизмов рефлексии, парсинга MSIL и анализа графа вызовов. В результате FxCopспособен обнаружить более 200 недочетов (или ошибок) в следующих областях:

– Архитектура библиотеки;
– Локализация;
– Правила именования;
– Производительность;
– Безопасность.

FxCop предусматривает возможность создания собственных правил с помощью специального SDK. FxCop может работать как в графическом интерфейсе, так и в командной строке.

JavaChecker. JavaChecker - это статический анализатор Java программ, основанный на технологии TermWare.

Это средство позволяет выявлять дефекты кода, такие как:

– небрежная обработка исключений (пустые catch-блоки, генерирование исключений общего вида и.т.п.);

– сокрытие имен (например, когда имя члена класса совпадает с именем формального параметра метода);

– нарушения стиля (вы можете задавать стиль программирования с помощью набора регулярных выражений);

– нарушения стандартных контрактов использования (например, когда переопределен метод equals, но не hashCode);

– нарушения синхронизации (например, когда доступ к синхронизированной переменной находится вне synchronized блока).

Набором проверок можно управлять, используя управляющие комментарии.

Вызов JavaChecker можно осуществлять из ANT скрипта.

Simian. Simian - анализатор подобия, который ищет повторяющийся синтаксис в нескольких файлах одновременно. Программа понимает синтаксис различных языков программирования, включая C#, T-SQL, JavaScript и VisualBasicR, а также может искать повторяющиеся фрагменты в текстовых файлах. Множество возможностей настройки позволяет точно настраивать правила поиска дублирующегося кода. Например, параметр порога (threshold) определяет, какое количество повторяющихся строк кода считать дубликатом.

Simian - это небольшой инструмент, разработанный для эффективного поиска повторений кода. Он не имеет графического интерфейса, но с командной строки вы можете запустить его или получить доступ к программному обеспечению. Результаты выводятся в текстовом формате и могут быть представлены в одном из встроенных форматов (например, XML). Хотя разреженный интерфейс Simian и ограниченные возможности вывода требуют некоторых упражнений, это поможет сохранить целостность и эффективность продукта. Simian подходит для поиска повторяющегося кода в больших и небольших проектах.

Резервный код снижает ремонтопригодность и обновление проекта. Вы можете использовать Simian, чтобы быстро найти дубликаты фрагментов кода в нескольких файлах одновременно. В связи с тем, что Simian может быть активирован из командной строки, можно включить в процесс сбора напоминаний при повторении кода или остановить процесс.

2 Методы сертификации программного обеспечения на отсутствие уязвимостей.

2.1 Анализ основных подходов к исследованию программ

Сертификация программной продукции, как и любой другой, введена в целях защиты пользователей от продукции недоброкачественной. Сертификат предоставляет определенную гарантию того, что приобретаемый программный продукт соответствует всем указанным в нем и его приложении требованиям нормативных документов, условиям и характеристикам. Необходимо учитывать, что в процессе сертификации происходит совершенствование, повышение качества программного средства и в части уточнения нормативных требований, и в части решаемых проектных задач, и в части назначения и области применения.

Проектирование и разработка программного обеспечения, как правило, производятся при недостаточной однозначности понимаемых требований к программному продукту. Требования к программному продукту должны быть сформулированы до начала разработки, проанализированы и оценены специалистами и согласованы с заказчиком. Начиная со стадии кодирования

с использованием инструментальных средств моделирования, анализа исходных текстов и анализа архитектуры разрабатываемого программного обеспечения необходимо проводить поэтапное тестирование программного средства.

При проведении предварительных испытаний появляется необходимость каких-либо изменений, связанных с доработкой заложенных в программы алгоритмов. При отсутствии описания программ данного комплекса возникает избыточность функциональных объектов, появляется неоправданное ветвление в алгоритмах, возникают тупиковые («незакрытые») переходы и ссылки. Отсутствие правильного организованного эталонного места хранения версий продукта не позволяет корректно отказаться от внесенных изменений и вернуться к предыдущему этапу работ. Все это является критичным по безопасности для программных средств.

Исследование программ и анализ программного обеспечения (ПО) базируется на методах и моделях нескольких научных направлений — теоретического программирования и прикладной теории алгоритмов, теории надежности и функциональной безопасности программного обеспечения, программной дефектологии, програм- мометрики, теории обеспечения качества ПО и других.

В последние годы развивается актуальное научное направление — теория обеспечения безопасности программ и их комплексов. Оно интегрирует основные научные положения прикладной теории алгоритмов, теории управления качеством программного обеспечения, теории надежности и с единых системных позиций изучает методы предотвращения, случайного или преднамеренного раскрытия, искажения или уничтожения хранимой, обрабатываемой и передаваемой информации в информационных системах, а также предотвращения нарушения их функционирования [1]. Вопросы обеспечения безопасности программ рассматриваются на всех этапах жизненного цикла, при этом выделяется обеспечение технологической и эксплуатационной безопасности ПО на двух обобщенных этапах — этапах его создания и эксплуатации. При обеспечении технологической безопасности рассматриваются методы анализа безопасности ПО и создания алгоритмически безопасных процедур.

Вопросы исследования программ неразрывно связаны с решением вопросов формализации семантик языков программирования, где четко определились три принципиальных направления: функциональный подход, дедуктивный метод и алгебраическая теория. Смысловая разноориентированность указанных направлений состоит в различном выражении семантики вычисляемой функции.

Взаимосвязь основных научных направлений, подходов и используемых методов приведена на рисунке 2.1.

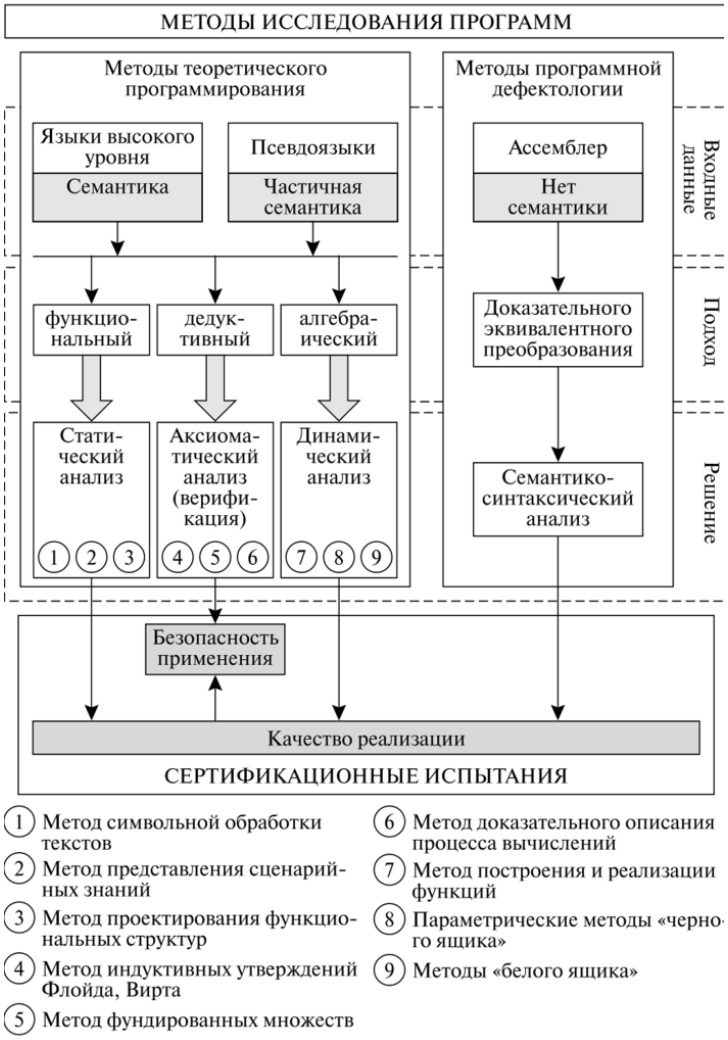

Рисунок 2.1 – Взаимосвязь основных научных направлений, подходов и методов исследования программ

Функциональный подход задает смысл через описание всех возможных типов функций переходов над возможными типами состояний, поэтому понятие состояния процесса вычислений присутствует явно, а последовательность требуемых переходов представлена лишь потенциально. Указанные действия базируются на утверждении о совпадении класса рекурсивных функций с классом вычисляемых. В этом направлении можно выделить ряд методов, применимых для решения задач статического анализа исходных текстов программ [5].

Это, прежде всего, такие достаточно эффективные методы, как методы символьной обработки текстов, представления сценарийных знаний, проектирования функциональных структур, доказательства свойств рекурсивных языков и программ. Формализация семантики здесь состоит в том, что смысл вычислений задается совокупностью потенциально допустимых вычислительных функций, отображающих все возможное множество состояний процесса вычислений в себя, в то время как традиционное определение смысла вычислений предполагает задание

конкретной последовательности состояний и функций перехода между ними. Такая семантическая модель получила в литературе название денотационной семантики.

В то же время, денотационная трактовка смысла, испытывая трудности описания собственно вычислений, дает значительные преимущества в выражении функциональной структуры и может эффективно использоваться для моделирования задачи вычисления и ее последующей декомпозиции (типизации). Для жизненного цикла программы — это подпроцессы спецификации и проектирования.

Алгебраическая теория, обобщающая так называемую операционную семантику языка и важная для задач динамического анализа исходных текстов, содержит основную информацию о вычислениях с заданием смысла путем указания функции переходов из состояния в состояние. Эта форма семантики использовалась (в различных вариантах) для определения практически всех императивных языков программирования. Наиболее полезными методами здесь являются метод построения реализации функций, параметрические методы «черного ящика» и методы «белого ящика».

В дедуктивном подходе семантика — это самостоятельное направление формализации, основанное не на задании состояний и функций над ними, а на определении и логической формулировке утверждений о свойствах состояний и вычисляемости (свобода, тотальность, эквивалентность).

Следует отметить, что представленные формы семантического описания однозначно отражают смысл конструкций языка на своем, отличном от других уровне абстракции. Вопросам соотношения методов семантического описания уделено достаточное внимание, главное состоит в утверждении эквивалентности языкового выражения вычисляемости в денотационной, аксиоматической и операционной семантиках.

2.2 Сертификация программного обеспечения

Программное обеспечение обеспечивает нормальную работу самого компьютера, обработку данных и их передачу другим устройствам обработки данных. Компьютерные программы разрабатываются программистами, а потом тестируются и отлаживаются другими IT- специалистами, прежде чем они станут доступны для пользователей и начнется их массовая эксплуатация. На программное обеспечение выдается сертификат в соответствии с ГОСТ 19791-90. В определенных случаях осуществляется сопровождение ПО, для осуществления которого необходимо получить сертификат в соответствии с ГОСТ Р ИСО/МЭК 14764-2002.

В силу специфики работы самой компьютерной техники (при этом выдается сертификат на ПО в соответствии с ГОСТ 25123-82) выделяют две основные категории ПО: системные и прикладные программы.

Системные программы управляют взаимодействием компонентов самого компьютера, а прикладные программы предназначены для решения внешних задач, например обработки текстов или воспроизведения

видеоинформации. В этом случае выдается сертификат на ПО в соответствии с ГОСТ 19781-90.

Прикладные программы могут быть разработаны в виде пакетов прикладных программ, на которые выдается сертификат на ПО в соответствии с ГОСТ Р ИСО/МЭК ТО 12119-2000, например для банковской сферы, офиса. Так, офисные пакеты прикладных программ включают в себя обычно электронные органайзеры, программы-переводчики, программы для распознавания считанной сканером информации, коммуникационные программы, в том числе и для работы в Интернете [6].

Прикладные программы также делятся на программы общего пользования (общераспространенные) и специальные программы. К специальным программам относятся, например, бортовые программные системы сбора информации.

Разрабатываются также специфические программы, например системы управления базами данных и групповое ПО (средства одновременной работы с файлами, корпоративной электронной почты, средства телеконференций и планирования проектов).

Базы данных разделяют на документальные (архивы) и фактографические (картотеки), централизованные и распределенные, табличные и иерархические. На их ПО выдается сертификат в соответствии с ГОСТ Р ИСО/МЭКТО 15271-2002, который отражает стандартный жизненный цикл любого программного средства, и ГОСТ Р ИСО/ МЭКТО 16326-2002.

В качестве третьего основного класса программного обеспечения часто также выделяют программные инструменты разработки программ.

Согласно другой классификации ПО можно разделить на базовое и сервисное.

Базовое программное обеспечение, в свою очередь, включает в себя:

- операционные системы (OS/2, Windows NT/ XP, Unix, Solaris);
- операционные оболочки:
- сетевые операционные системы (обеспечивают обработку, передачу и хранение данных в Интернете);
- систему управления файлами;
- системные утилиты.

Современное состояние развития ПО показывает, что эти сегменты сливаются в глобальные операционные системы, выполняющие функции всех этих элементов.

Сервисное программное обеспечение (прикладные программы) можно классифицировать по функциональному признаку (ГОСТ Р ИСО/МЭК ТО 12182-2002) и разделить на следующие категории:

- программы диагностики работоспособности компьютера и обслуживания дисков (утилиты):
- архиваторы;
- антивирусные программы;

– текстовые редакторы;
– программы для работы с видео;
– программы для работы с аудио;
– программы шифрования;
– программы для работы с почтой;
– интернет-браузеры;
– программы для закачки файлов из Интернета (менеджеры закачки) и многие другие.

2.3 Статистика выявления уязвимостей в программном обеспечении в рамках сертификационных испытаний

Методика анализа уязвимостей, используемых испытательными лабораториями (IL) в рамках сертификационных испытаний, основана на" общей методологии оценки " (международный стандарт ISO/IEC 18045) и, как правило, состоит из следующих этапов [7].

1 уровень. Определение определенной (проверенной) уязвимости объекта сертификации. На данном этапе специалисты Ил ищут определенные (подтвержденные) уязвимости в общедоступных источниках информации, например, в базе данных угроз информационной безопасности ФСТЭК России, ресурсах CVE, FWD и на ресурсе разработчика программного обеспечения. При обнаружении информации о уязвимостях тестирование приостанавливается до тех пор, пока обновление программного обеспечения "закрывает" определенные уязвимости.

2 уровень. Определение ранее неопубликованной уязвимости объекта сертификации. На этом этапе эксперты il обычно выполняют следующие шаги:

– Шаг 1: создание перечня потенциальных уязвимостей объекта сертификации на основе изучения различных источников данных (проектная документация для объекта сертификации, исходный код, информация из открытых источников)));
– Шаг 2: создание сценариев тестирования (атаки) для каждой выявленной потенциальной уязвимости;
– Шаг 3: Начните тесты, чтобы определить, правильно ли предположить.

В этом исследовании специалисты Ил НПО "Эшелон" использовали следующие методы для составления списка потенциальных уязвимостей на шаге 1:

– документально-аналитический метод, обеспечивающий формирование перечня потенциальных уязвимостей на основе информации об известных уязвимостях в аналогичных по функциональности продуктах, на основе информации о подъездных компонентах и типовых уязвимостях, в других вариантах объекта сертификации (например, выше), которые

являются особыми для технологий, используемых при разработке объекта сертификации. ;

– статистический анализ исходных текстов (при достижении оригинальных текстов в рамках сертификационного теста));

– fuzzing тесты.

В случае выявления уязвимостей на объекте сертификации разработчику программного обеспечения представляются подробные технические сведения для получения подтверждения заключений специалистов ЛЛ и формирования мер по нейтрализации выявленных уязвимостей (путем обновления программного обеспечения или определения руководства пользователя). Уязвимость, обнаруженная при сертификационных испытаниях/проверках инспекции, считается необходимым пройти процедуру раскрытия угрозы безопасности информации .

Исследования проводились в период 2016 — 2017 гг. в рамках испытаний 76 продуктов по линии системы сертификации во всех системах сертификации (сертификационные испытания, инспекционные контроли) в ИЛ НПО «Эшелон». Распределение исследованных продуктов по типам представлено на рисунке 2.2.

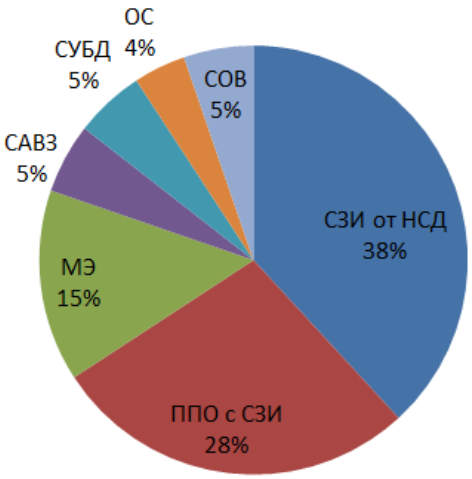

Рисунок 2.2 – Распределение исследованных продуктов по типам

Разработчиками продуктов являлись как отечественные, так и зарубежные организации.

В зависимости от критериев сертификации, определяемых потенциальной областью применения сертифицируемого продукта, экспертам ИЛ предоставлялся или не предоставлялся доступ к исходным текстам объекта сертификации (рисунок 2.3).

Рисунок 2.3 – Распределение исследованных продуктов в зависимости от доступа к исходным текстам

В результаты исследований специалистами ИЛ НПО «Эшелон» была выявлена 81 уязвимость (уязвимости были выявлены в 26 продуктах из 76 исследованных). Для всех выявленных уязвимостей ПО было получено подтверждение об их актуальности со стороны разработчика ПО, разработчиками ПО были приняты меры по устранению выявленных уязвимостей.

На рисунке 2.4 показано распределение выявленных уязвимостей по степени критичности (оценка выполнялась по методике CVSS версия 3.0).

Рисунок 2.4 – Распределение выявленных уязвимостей в зависимости от степени критичности

Наиболее популярным типом уязвимого ПО стало прикладное ПО со встроенными средствами защиты информации (рисунок 2.5).

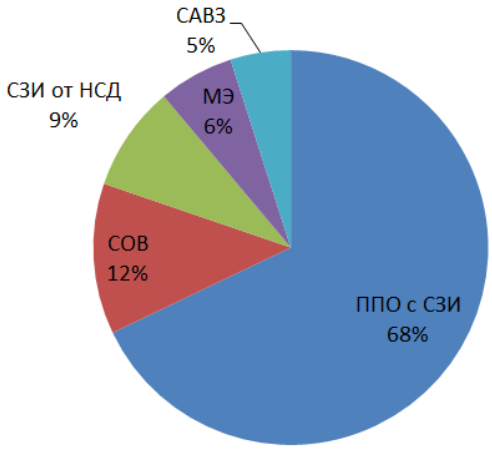

Рисунок 2.5 – Распределение выявленных уязвимостей в зависимости от типа исследованного ПО

Основными типами векторов успешных атак, которые были разработаны экспертами ИЛ с целью подтверждения актуальности уязвимости, стали (рисунок 2.6):

– межсайтовое выполнение скриптов (CAPEC-63);
– межсайтовая подделка запросов (CAPEC-62);
– повышение привилегий, связанное с обходом функций безопасности (CAPEC-233);
– атаки, направленные на отказ в обслуживании (CAPEC-2);
– раскрытие критичной информации ПО в сообщениях об ошибках (CAPEC-54);
– инъекция SQL-кода (CAPEC-66).

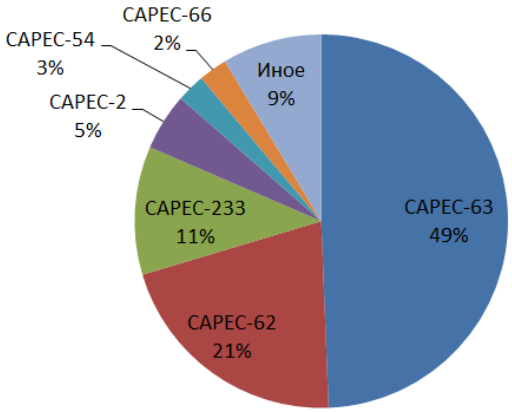

Рисунок 2.6 - Распределение выявленных уязвимостей в зависимости от типа вектора атаки

В категорию «Иное» попали такие типы векторов атак, как: удалённое выполнение команд операционной системы путем передачи данных в HTTP-запросах (CAPEC-76), XML-инъекция (CAPEC-250), фиксация сессии (CAPEC-61), выход за пределы назначенной директории (CAPEC-126), атаки

типа «Reparse-Point», «RegSafe/RegRestore». Основными типами недостатков ПО, ставших причинами уязвимостей, стали (рисунок 2.7):

– неверное использование данных, полученных из недоверенного источника, для генерации HTML-страницы (CWE-79);

– использование аутентификационных данных (данные cookie) для авторизации запроса (CWE-352);

– неверное использование данных, полученных из недоверенного источника, при выполнении функций безопасности (CWE-807);

– отсутствие авторизации при выполнении критичных операций (CWE-862);

– неверное использование данных, полученных из недоверенного источника, для генерации запроса к СУБД (CWE-89);

– неверная генерация сообщений об ошибках (CWE-209).

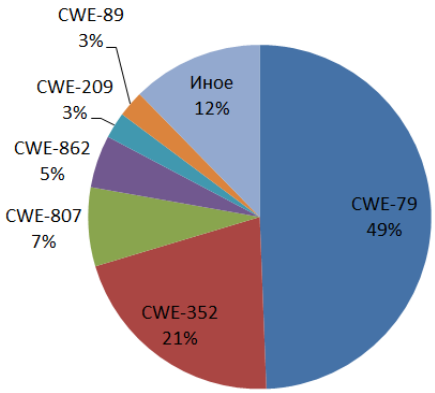

Рисунок 2.7 – Распределение выявленных уязвимостей в зависимости от ошибки (недостатка) ПО

В категорию «Иное» попали такие типы ошибок ПО, как: использование заданных в коде программы аутентификационных данных (CWE-798), переполнение буфера (CWE-120), ошибки, приводящие к фиксации сессии (CWE-384), неверное использование данных, полученных из недоверенного источника, для генерации команд ОС (CWE-22) и пр. Говоря о методах формирования перечня потенциальных уязвимостей, следует отметить, что большинство уязвимостей было обнаружено благодаря предположениям, сделанным на основе изучения документации на объект сертификации и данных об уязвимостях в схожих с объектом сертификации продуктах (рисунок 2.8).

Рисунок 2.8 – Распределение выявленных уязвимостей в зависимости от метода формирования перечня потенциальных уязвимостей

3 Практическая часть. Работа с PVS-Studio на Windows.

3.1 Выбор инструментального средства и обзор возможностей

PVS-Studio-это статический инструмент тестирования безопасности приложений. Другими словами, анализатор PVS-Studio находит ошибки, мертвые коды и опечатки, а также уязвимые места в программе.

PVS-Studio сортирует все диагностические сообщения на три уровня: высокий (high), средний (medium) и низкий (low). Некоторые сообщения классифицируются как особые сбои (Fails). Описание уровней более подробно:

– High (1) – диагностические сообщения высокого уровня. Эти предупреждения часто указывают на ошибки, которые необходимо устранить немедленно;

– Medium (2) – диагностические сообщения среднего уровня доверия, на которые тоже нужно уделить внимание;

– Low (3) – диагностические сообщения с минимальным уровнем доверия, которые указывают на незначительную долю дефекта в коде. Среди таких предупреждений процент ложной работы, как правило, выше;

– Fails - внутренние сообщения анализатора, информирующие о возникновении каких-то проблем при работе. В группу Fails попадают сообщения об ошибках анализатора (например, сообщения с кодами V001, V003 и т.п.), а также любой необработанный вывод вспомогательных программ, используемых самим анализатором во время анализа (препроцессор, командный процессор cmd), выдаваемый ими в stdout/stderr. Например, в группу Fails может попасть сообщение препроцессора об ошибках препроцессирования, доступа к файлам (файл отсутствует или заблокирован антивирусом) и т.п.

Стоит помнить, что конкретный код ошибки не обязательно привязывает её к определённому уровню достоверности, а распределение сообщений по уровням сильно зависит от контекста, в котором они были

сгенерированы. При использовании плагина для Microsoft Visual Studio или в приложении C and C++ Compiler Monitoring UI окно вывода диагностических сообщений содержит кнопки уровней, позволяющие сортировать сообщения по мере необходимости.

Анализатор содержит 5 видов диагностических правил:

– General (GA) - диагностики общего назначения. Базовый набор диагностических правил PVS-Studio;

– Optimization (OP) - диагностики микрооптимизации. Инструкции по повышению эффективности и безопасности кода;

– 64-bit (64) - диагностики, дающая возможность обнаружить определенные ошибки, связанные с разработкой 64-битных приложений, а также переносом кода с 32-битной на 64-битную платформу;

– Customers' Specific (CS) - узкоспециализированные диагностики, разработанные по желанию пользователей. По умолчанию эта диагностика автоматически отключается;

– MISRA - диагностики, разработанные в соответствии со стандартом MISRA (Motor Industry Software Reliability Association). По умолчанию этот набор диагностик отключен.

Краткие обозначения групп диагностик (GA, OP, 64, CS, MISRA), наряду с номерами уровней достоверности предупреждений (1, 2, 3), используются для сокращенной формы записи, например, в параметрах командной строки.

При выборе группы диагностических правил отображаются или скрываются соответствующие сообщения. Пример вывода диагностических сообщений показан на рисунке 3.1.

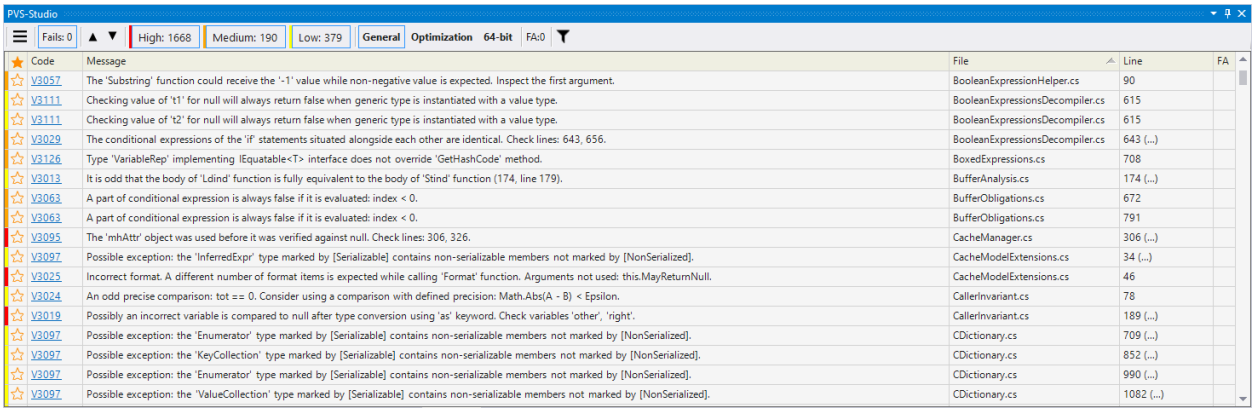

Рисунок 3.1 – Вывод диагностических сообщений

Диагностические сообщения, которые выводятся после анализа, можно группировать и фильтровать по различным параметрам.

3.2 Системные требования и установка PVS-Studio

Анализатор PVS-Studio интегрируется в различные среды разработки (IDE), в том числе Microsoft Visual Studio (2010-2019), Intellij Idea и др. PVS-

Studio – платное инструментальное средство, но есть возможность получить временную лицензию на 7 дней. Для получения лицензии необходимо перейти на официальный сайт и заполнить форму. В течении 3 дней разработчики дают обратную связь. Подробный интерфейс официального сайта показан на рисунке 3.2, выбор операционной системы и языков программирования показан на рисунке 3.3, получение лицензии от разработчиков через электронную почту показан на рисунке 3.4.

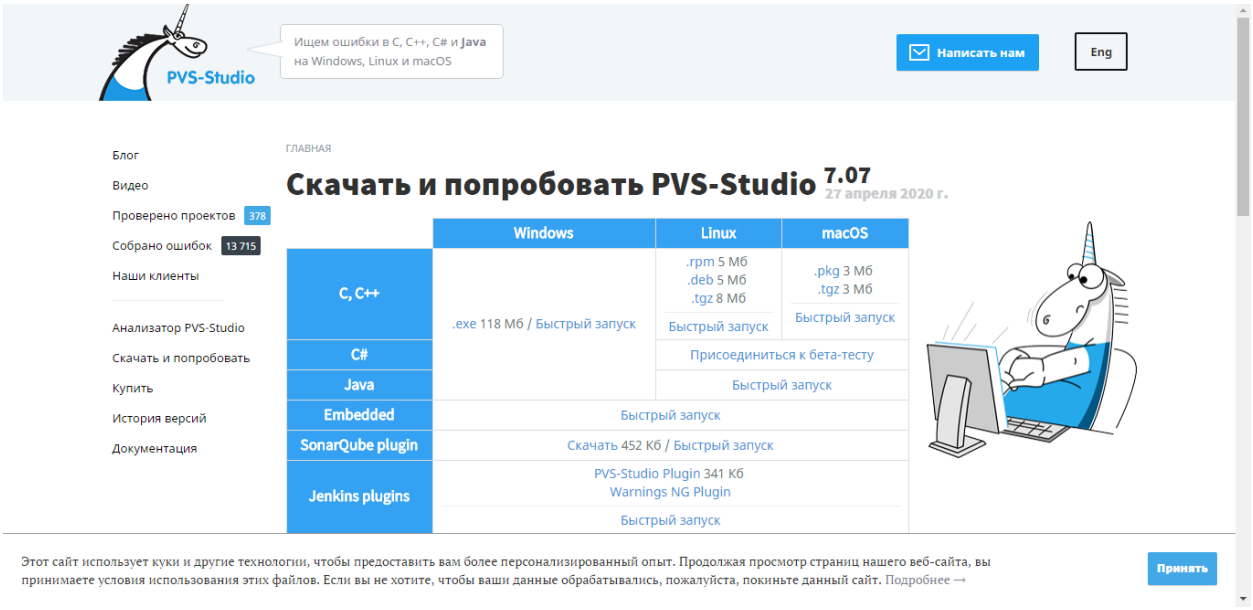

Рисунок 3.2 – Интерфейс официального сайта

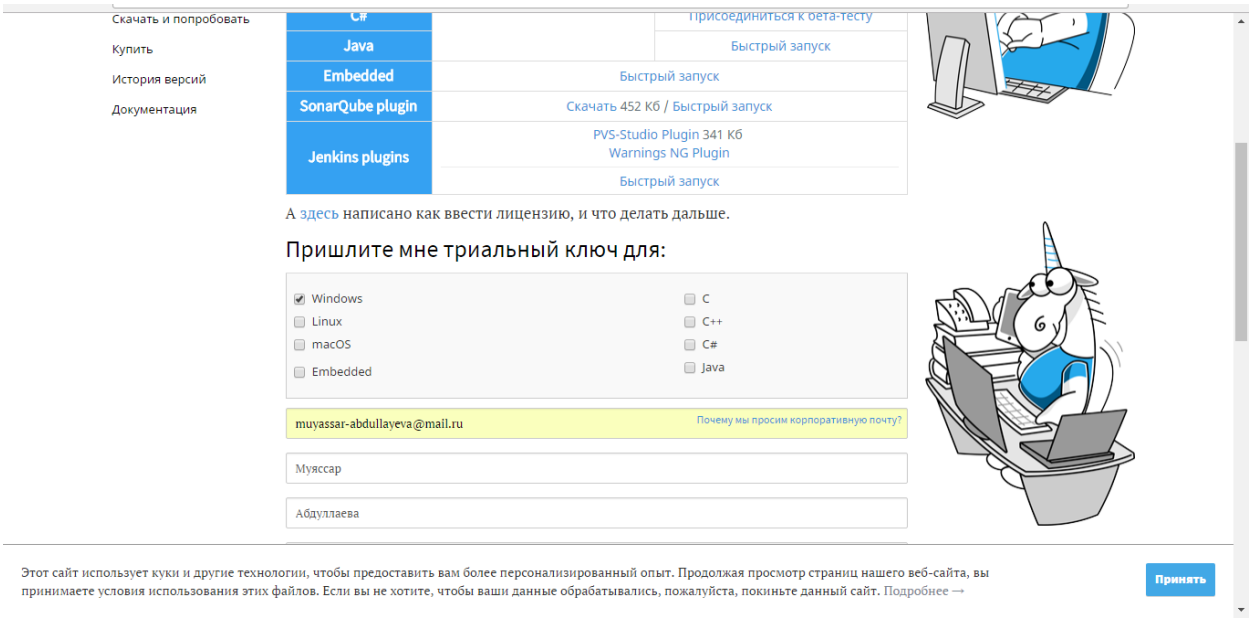

Рисунок 3.3 – Выборка операционной системы и языка программирования

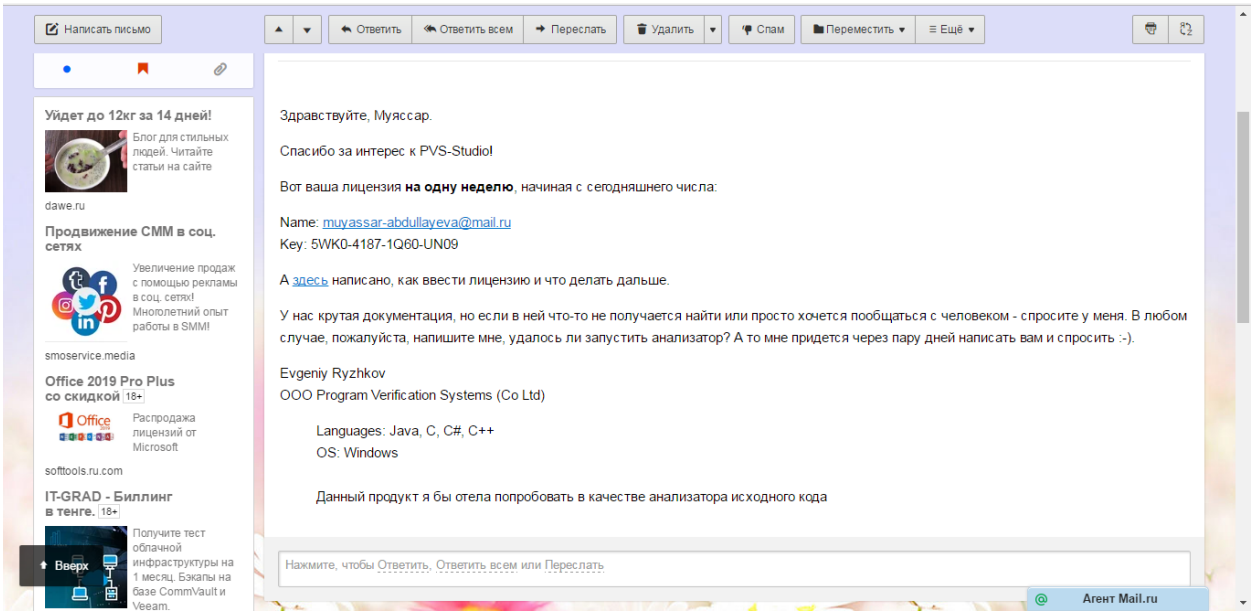

Рисунок 3.4 – Получение временной лицензии

Получив установочный пакет PVS-Studio, можно приступить к установке программы. Начальный интерфейс PVS – Studio показан на рисунке 3.5.

Рисунок 3.5 – Начальный интерфейс PVS – Studio

После подтверждения лицензионного соглашения необходимо получить выбор вариантов интеграции PVS-Studio в поддерживаемые среды

разработки: Microsoft Visual Studio. Параметры интеграции, недоступные в текущей системе, убираются. Если на персональном компьютере установлено несколько версий одной и той же или разной среды разработки, анализатор может быть интегрирован во все доступные версии. При установке в среду разработки имеется возможность выбора установочных компонент что показано на рисунке 3.6.

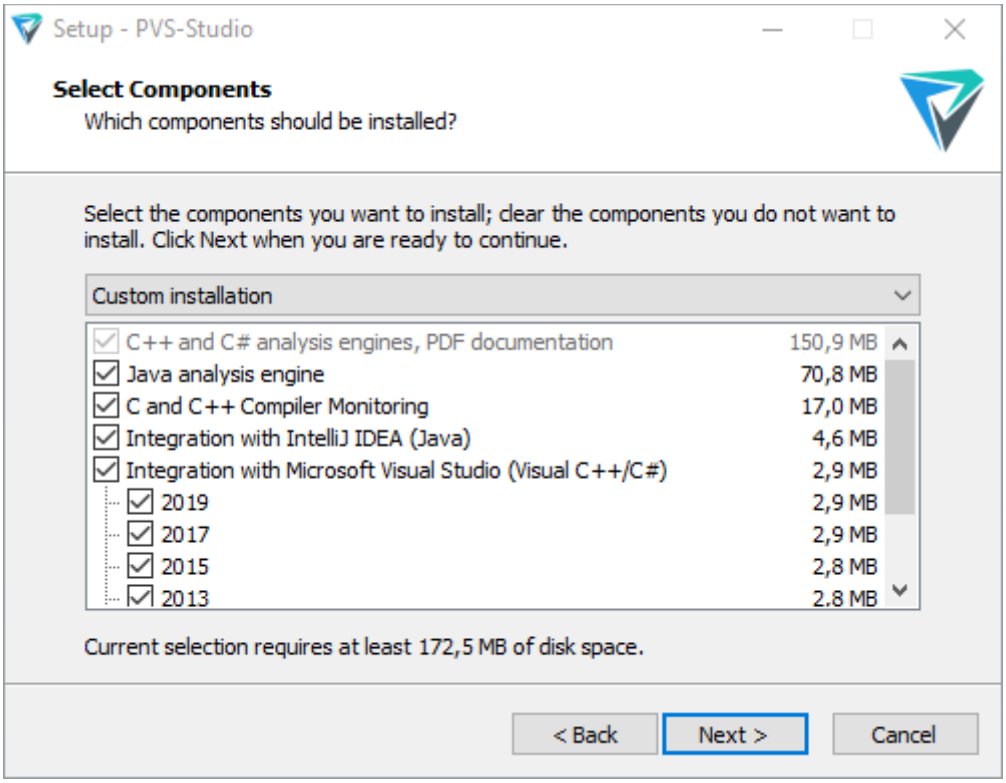

Рисунок 3.6 – Выборка установочных компонент

Необходимо запустить среду разработки и открыть окно Info (пункт меню справки),чтобы убедиться, что инструмент PVS Studio установлен правильно. Однако анализатор PVS-Studio должен присутствовать в списке установленных компонентов (рисунок 3.7).

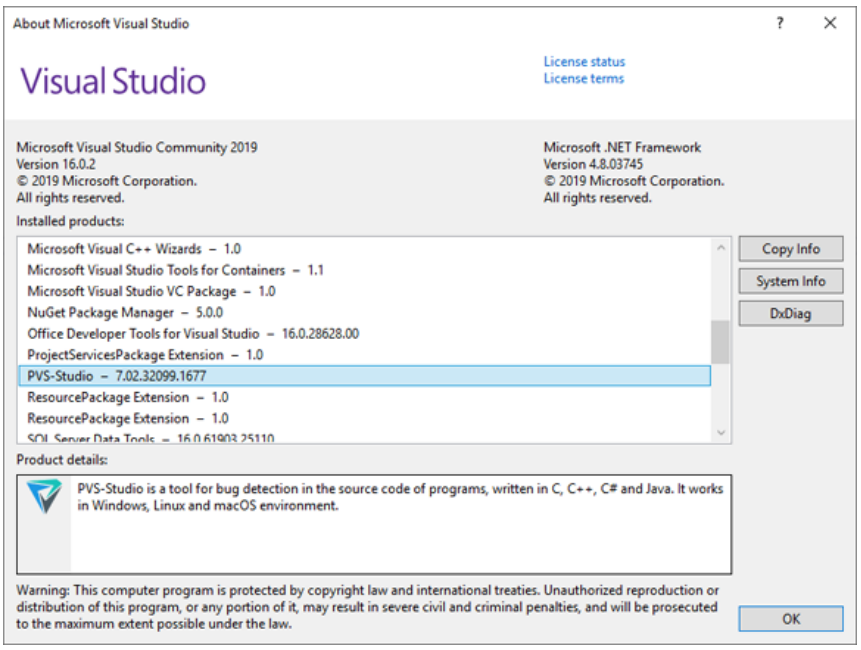

Рисунок 3.7 – Список установленных компонентов

3.3 Основы работы с PVS-Studio

При установке PVS-Studio можно выбрать, в какие среды его необходимо интегрировать (Visual Studio).

После выбора необходимых элементов и установки, PVS-Studio интегрируется в IDE. На рисунке 3.8 изображен соответствующий элемент в меню Visual Studio, а также окно отображения диагностических сообщений.

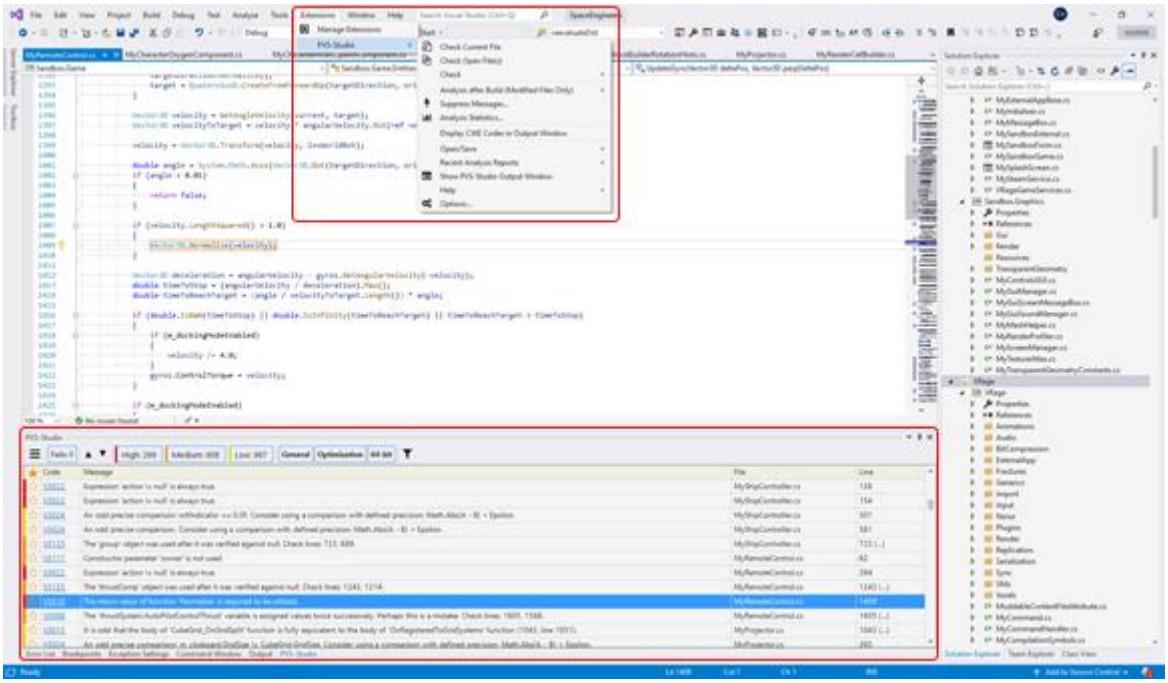

Рисунок 3.8 – Полный интерфейс среды разработки после анализа

В меню настроек можно настроить PVS-Studio чтобы облегчить работу. Например, существуют следующие функции:

– Выбор препроцессора;
– Исключение файлов и папок из проверки;
– Выбор диагностических сообщений, выводящихся при проверке;
– Множество прочих настроек.

При работе с PVS – Studio, интегрированным в Visual Studio, есть возможность запуска анализа различных видов: анализ на решение, анализ проекта, анализ файла, анализ выбранных элементов (рисунок 3.9).

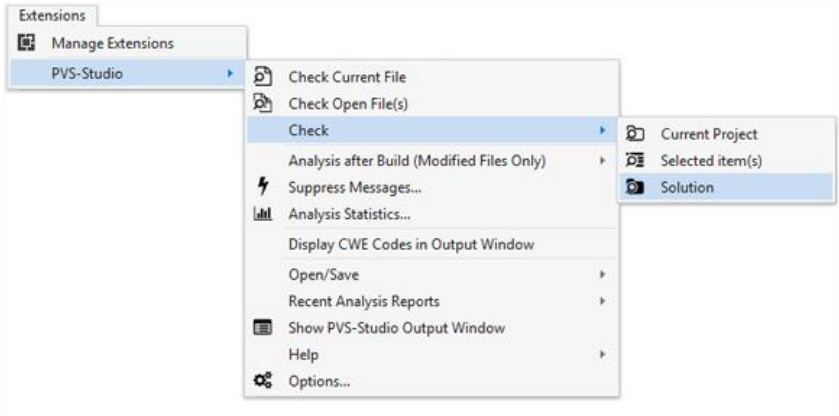

Рисунок 3.9 – Запуск анализа на решение

После запуска анализа на экране появится индикатор процесса с кнопками Pause (приостановить анализ) и Stop (прервать анализ) (рисунок 3.10). Найденные во время проверки дефекты и уязвимости выводятся в окно диагностических сообщений.

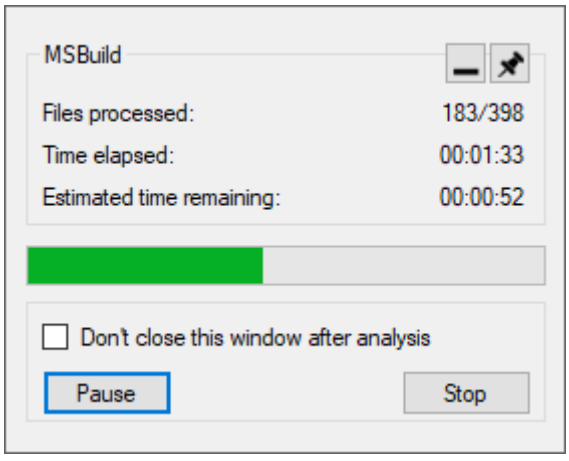

Рисунок 3.10 – Индикатор процесса

3.4 Работа со списком диагностических сообщений

При работе с большим количеством сообщений (а при первой проверке крупных проектов, фильтры еще не настроены и количество сообщений

может достигать десяти тысяч) в исходном окне PVS Studio лучше использовать средства навигации поиска и фильтрации.

3.4.1 Навигация и сортировка

Выходное окно PVS Studio предназначено, для упрощения навигации по анализируемому коду проекта и перехода к разделам кодов, которые содержат потенциальные ошибки. Двойной щелчок из одного сообщения в списке автоматически открывает в редакторе кода файл, отображающий это сообщение, перемещает курсор в строку интереса и выбирает его. Кнопки быстрой навигации (рисунок 3.11) позволяют легко просматривать потенциально опасные места, обнаруженные в исходном коде.

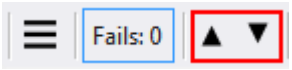

Рисунок 3.11 — Кнопки быстрого перехода

Для вывода результатов анализа, окно студии PVS использует виртуальную таблицу, которая позволяет быстро просматривать и сортировать сообщения, созданные даже для очень больших проектов. Левая колонка таблицы, например, предназначена для разделения необходимых сообщений, имеющих значение возврата. Эта колонка также поддерживает сортировку, поэтому не сложно найти все сообщения, отмеченные таким образом. Контекстное меню "отображение столбцов" позволяет настроить столбцы, указанные в таблице (рисунок 3.12):

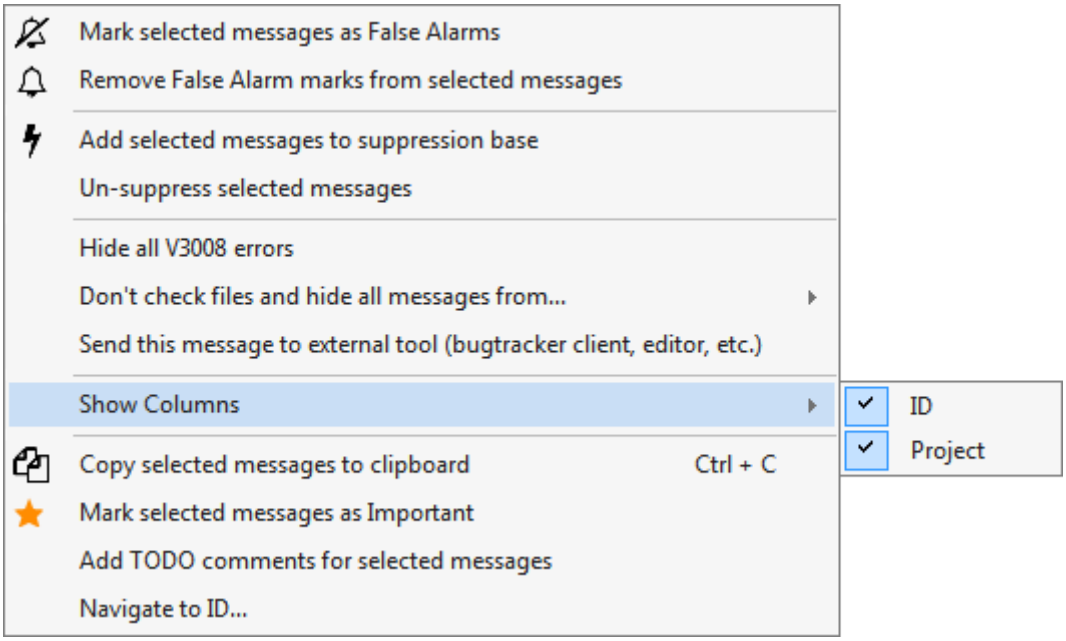

Рисунок 3.12 — Настройка отображения таблицы вывода результатов

Таблица поддерживает большое количество выделений с помощью стандартных комбинаций Ctrl и Shift, которая сохраняется и после пересортировки по любой другой колонке. Пункт меню "Copy selected messages to clipboard" (либо сочетание Ctrl+C) позволяет скопировать в буфер обмена содержимое всех выделенных в таблице строк.

3.4.2 Фильтрация сообщений

Механизмы фильтрации окна вывода PVS-Studio позволяют быстро находить и просматривать индивидуальные диагностические сообщения, а также их целые группы. На панели окон есть набор переключателей, для включения или выключения отображения сообщений из соответствующих групп сообщений (рисунок 3.13).

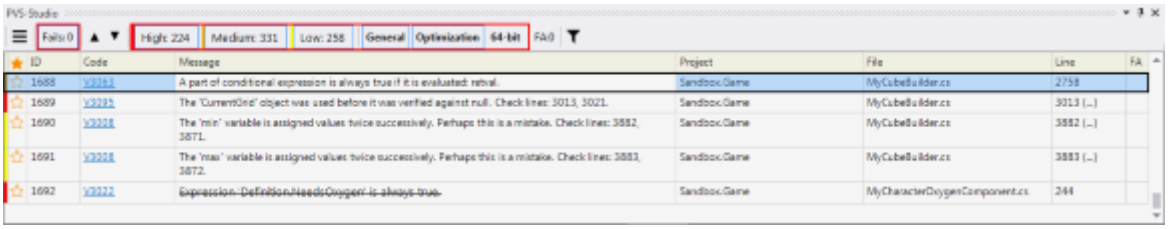

Рисунок 3.13 — Группы фильтрации сообщений

Все переключатели можно разделить на 3 группы: фильтры на основе диагностического доверия сообщений, фильтры на основе сообщений, относящихся к определенному виду диагностических правил, и ложно положительный фильтр, обозначенный кодом. Когда удаляете эти фильтры, в списке скрываются все соответствующие сообщения.

Механизм быстрого фильтрования позволяет сортировать отчет анализатора по указанным ключевым словам. Необходимо использовать кнопку быстрого фильтра на панели инструментов окна, чтобы открыть окно быстрого фильтра, (изображение3.14).

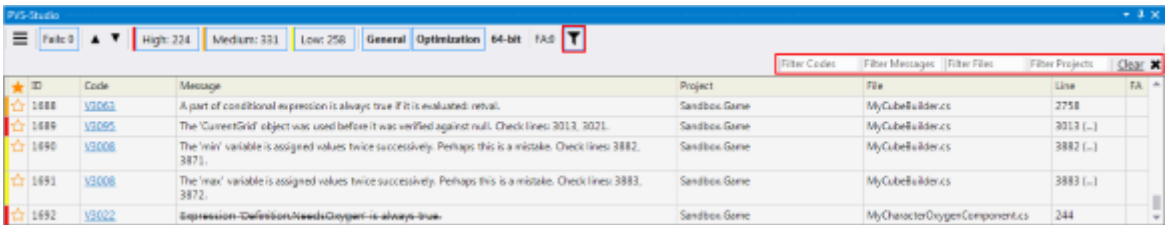

Рисунок 3.14 — Панель быстрой фильтрации

Быстрая фильтрация позволяет просмотреть сообщения в соответствии с фильтрами по 3-м ключевым словам: по коду сообщения, по тексту сообщения и по файлу, содержащему данное сообщение. Например, отобразить все сообщения, содержащие слово 'odd' из файла 'command.cpp'. Изменения в списке сообщений отображаются сразу после выхода из поля

ввода ключевого слова (при потере фокуса). Кнопка Reset Filters удаляет ключевые слова, установленные в данный момент.

Все эти фильтры могут быть объединены с помощью фильтрации, например, уровень отображаемых сообщений и файл, к которому сообщения должны относиться, исключая сообщения, помеченные как ложные срабатывания.

3.4.3 Быстрый переход к отдельным сообщениям

При необходимости перехода на определенное сообщение в таблице, можно воспользоваться быстрым переходом к диалогу строки, который вызывается через пункт контекстного меню "Navigate to ID..." (рисунок 3.15). Диалог быстрого перехода к сообщению показан на рисунке 3.16.

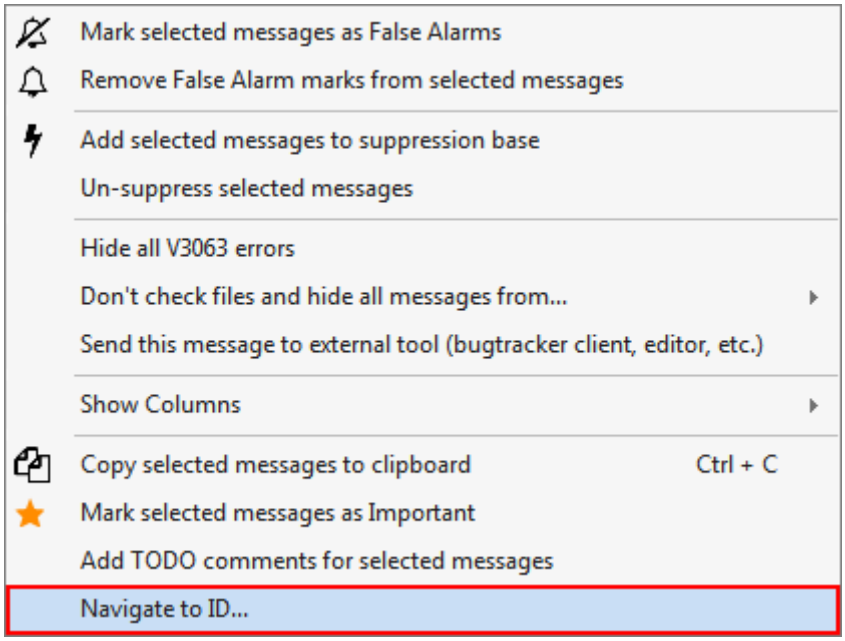

Рисунок 3.15 - Вызов диалога быстрого перехода

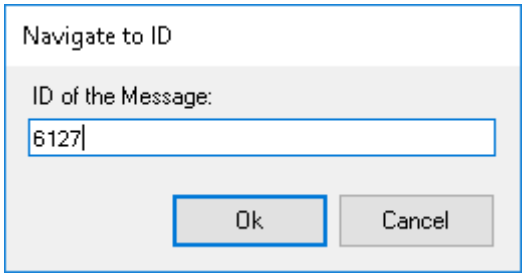

Рисунок 3.16 - Диалог быстрого перехода к сообщению

PVS - Studio имеет уникальный идентификатор каждого сообщения в списке исходящих - порядковый номер добавления этого сообщения в таблицу, указанную в графе ID. В окне быстрого доступа можно автоматически загрузить сообщение с указанным идентификатором, независимо от сортировки текущей таблицы и выбранных строк. Обратите внимание, что идентификаторы сообщений, указанные в таблице, не всегда

являются последовательными, так как некоторые сообщения могут быть скрыты с помощью механизмов фильтрации.

3.4.4 Организация работы с помощью Visual Studio Task List

Очень часто один человек не имеет возможности оценить каждое сообщение статического анализатора на предмет ложно-позитивного срабатывания и, тем более, внести коррективы в соответствующий участок исходного кода, а поэтому чаще всего в разработке крупных проектов принимают участие распределённые группы разработчиков. В этом случае имеет смысл передать такое уведомление разработчику, который непосредственно отвечает за данный раздел.

В PVS-Studio комментарии TODO позволяют автоматически создавать специальные комментарии и добавлять в код, которые содержат всю необходимую информацию для оценки и анализа фрагмента программы.

Такой комментарий будет сразу отображён в окне задач Visual Studio (окно Task List, для версии Visual Studio 2010 необходимо включить разбор комментариев в настройках Tools->Options->Text Editor->C++->Formatting->Enumerate Comment Tasks->true) при условии, что в настройках Tools->Options->Environment->Task List->Tokens задана соответствующая TODO лексема (присутствует в настройках по умолчанию). Комментарий добавляются с помощью команды контекстного меню 'Add TODO comments for selected messages' (рисунок 3.17)

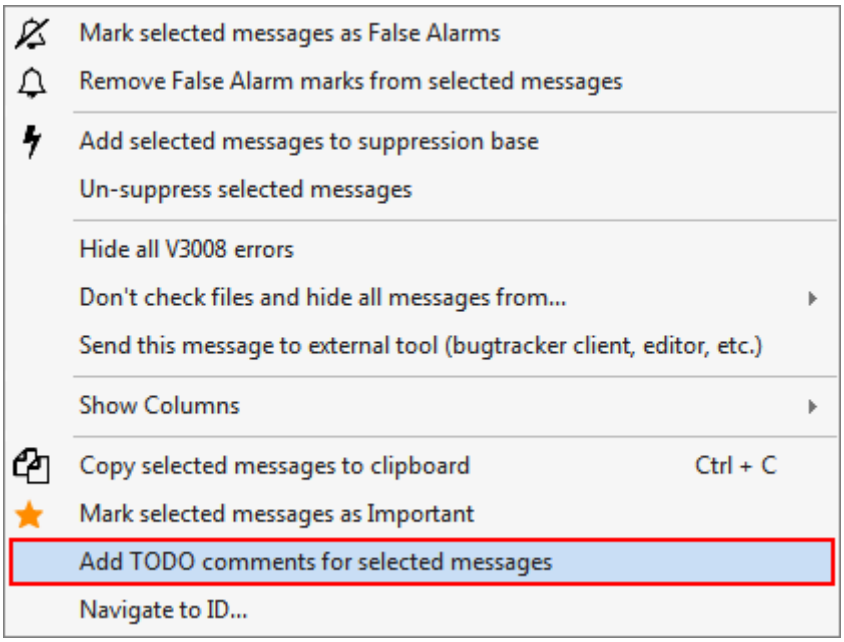

Рисунок 3.17 - Вставка TODO комментария

Комментарий TODO вставляется в строку с сообщением и содержит код ошибки, текст сообщения и ссылку на электронную документацию для данного типа ошибок. Это объяснение может легко найти любой разработчик, имеющий доступ к исходному коду, благодаря окну списка

задач, а сам текст комментариев позволяет обнаружить и исправить возможные ошибки и уязвимости даже в случае отсутствия у программиста установленной версии PVS-Studio или полного отчёта о работе анализатора (рисунок 3.18).

Рисунок 3.18 - Окно заданий Visual Studio

В Visual Studio открыть окно Task List можно через меню View->Other Windows->Task List. Комментарии TODO отображаются в разделе Comments окна.

3.5 Диагностика анализатора

Диагностические сообщения представляют собой ряд предупреждений, выведенных после проведения статического анализа. Диагностические сообщения зависят от языка программирования, на котором написан анализируемый код программы, и имеют отличия между собой. На официальном сайте все диагностические сообщения сгруппированы.

Поскольку распределение диагностики очень обусловлено, некоторые диагнозы добавляются в несколько групп. Например, ошибочный случай может быть объяснен как обычная ошибка " if (abc = abc)", так и проблемой безопасности, так как ошибка приводит к уязвимости кода, если входящие данные неверны.

Некоторые ошибки, наоборот, не имеют места в таблице - они очень необычные. Однако таблица дает представление о функционировании анализатора статического кода.

3.5.1 Диагностика исходных кодов, написанных на C++

Диагностика кодов, написанных на C++, показала около 556 дефектных сообщений. Данное число на много раз превышает количество дефектных сообщений кодов, написанных на C# и Java (Приложение А).

Диагностические сообщения, написанные на C++, имеют идентификационные номера V501 – V1057.

- V597. The compiler could delete the 'memset' function call, which is used to flush 'Foo' buffer. The RtlSecureZeroMemory() function should be used to erase the private data.

Анализатор обнаружил потенциальную ошибку, когда массив с конфиденциальной информацией не будет очищаться. Ошибочный код выглядит следующим образом (рисунок 3.19):

```
void Foo()
{
  char password[MAX_PASSWORD_LEN];
  InputPassword(password);
  ProcessPassword(password);
  memset(password, 0, sizeof(password));
}
```

Рисунок 3.19 – Ошибочный код с переполнением буфера

Функция на стеке создает временный буфер, в котором будет хранится пароль пользователя. По окончанию действий с паролем необходимо данный буфер удалить, чтобы этот пароль не сохранился. Сохранение пароля может привести к нежелательным последствиям. Код, показанный на рисунке 3.19 может оставить буфер неочищенным, поэтому необходимо его скорректировать. Для этого необходимо использовать специальную функцию RtlSecureZeroMemory или memset_s.

Исправленный вариант кода показан на рисунке 3.20.

```
void Foo()
{
  char password[MAX_PASSWORD_LEN];
  InputPassword(password);
  ProcessPassword(password);
  RtlSecureZeroMemory(password, sizeof(password));
}
```

Рисунок 3.20 – Исправленный вариант кода

По всем диагностическим сообщениям исправления предоставляется PVS – Studio.

- V631. Consider inspecting the 'Foo' function call. Defining an absolute path to the file or directory is considered a poor style.

Анализатор обнаружил потенциальную ошибку при вызове функции, предназначенную для работы с файлами. Один аргумент передаёт в функцию точный путь папки, или же его содержимого. Использование такой функции является небезопасным. Пример такого кода показан на рисунке 3.21.

```
FILE *text = fopen("c:\\TEMP\\text.txt", "r");
```

Рисунок 3.21 – Некорректная функция работы с файлами

Будет правильно если функция будет получать путь к директории исходя из определенных условий. Исправленный вариант кода показан на рисунке 3.22.

```
string fullFilePath = GetFilePath() + "text.txt";
FILE *text = fopen(fullFilePath.c_str(), "r");
```

Рисунок 3.22 – Исправленный вариант кода

-V755. Copying from unsafe data source. Buffer overflow is possible.

Анализатор установил, что данные были перенесены из незащищенного источника в буфер зарегистрированного размера.

Примером этого источника могут быть аргументы командной строки, длина которой неизвестна (рисунок 3.23):

```
char *tmp = (char*)malloc(1024);
....
strcat(tmp, argv[0]);
```

Рисунок 3.23 – Код, имеющий дефект переполнения буфера

Если количество копированных данных превышает размер буфера, он будет переполнен. Для его предотвращения необходимо предварительно рассчитать необходимый объем памяти (рисунок 3.24):

```
char *src = GetData();
char *tmp = (char*)malloc(strlen(src) + strlen(argv[0]) + 1);
....
strcpy(tmp, src);
strcat(tmp, argv[0]);
```

Рисунок 3.24 – Расчет необходимой памяти

-V773. The function was exited without releasing the pointer/handle. A memory/resource leak is possible.

Анализатор нашел возможную утечку памяти в коде. Такая ситуация возникает при высвобождении выделенной памяти с помощью" malloc " или "new".

Пример этого кода показан на рисунке 3.25:

```
int *NewInt()
{
  int *p = new (std::nothrow) int;
  ....
  return p;
}

bool Test()
{
  int *p = NewInt();
  int res = *p;
  return res;
}
```

Рисунок 3.25 – Утечка памяти в коде

В данном примере кода выделение памяти спрятано в вызове другой функции. После вызова другой функции необходимо освободить память. Исправленный код показан на рисунке 3.26.

```
int *NewInt()
{
  int *p = new int;
  ....
  return p;
}

bool Test()
{
  int *p = NewInt();
  int res = *p;
  delete p;
  return res;
}
```

Рисунок 3.26 – Исправленный код

У статического анализатора меньше информации об указателях, чем у динамического анализатора, поэтому он может выявлять некорректные диагностические сообщения, если память освобождается нетривиально или далеко от места, где выделяется. Для отключения таких предупреждений предусмотрен специальный комментарий: //+V773:SUPPRESS, class:className, namespace:nsName

3.5.2 Диагностика исходных кодов, написанных на C#

Диагностические сообщения выявленные в кодах, написанных на C# составляют 152 с V3001 по V3153.

- V3019. It is possible that an incorrect variable is compared with null after type conversion using 'as' keyword.

Анализатор обнаружил потенциальную ошибку, которая может привести к доступу по нулевой ссылке.

Анализатор заметил в коде следующую ситуацию. Сначала объект базового класса приводится к производному классу с помощью оператора 'as'. А затем этот же объект проверяется на значение null, хотя в этом случае скорее всего предполагалось проверить на null объект производного класса.

Рассмотрим следующий пример. Объект BaseObj не является экземпляром производного класса. В этом случае программа переключается на NullReferenceException, когда вы вызываете функцию FUNC. Анализатор дает предупреждение для этого кода и указывает две строки. В первой строке объект базового класса проверяется на 0. Во второй строке объект базового класса переливается в объект производного класса (рисунок 3.27).

```
Base baseObj;
Derived derivedObj = baseObj as Derived;
if (baseObj != null)
{
    derivedObj.Func();
}
```

Рисунок 3.27 – Код, приводящий к доступу по нулевой ссылке

Исправленный вариант кода приведен на рисунке 3.28:

```
Base baseObj;
Derived derivedObj = baseObj as Derived;
if (derivedObj != null)
{
    derivedObj.Func();
}
```

Рисунок 3.28 – Исправленный вариант кода

- V3061. Parameter 'A' is always rewritten in method body before being used.

Анализатор нашёл потенциальную ошибку в теле метода. Один из его параметров будет перезаписан перед его использованием. Таким образом, значение, приходящее к методу, исчезает.

Эта ошибка имеет различные представления. Рассмотрим пример кода (рисунок 3.29):

```
void Foo1(Node A, Node B)
{
  A = SkipParenthesize(A);
  B = SkipParenthesize(A);
  // do smt...
}
```

Рисунок 3.29 – Программный код с уязвимостью

Здесь допущена опечатка, из-за чего объект 'B' примет неверное значение. Исправленный код выглядит так (рисунок 3.30):

```
void Foo1(Node A, Node B)
{
  A = SkipParenthesize(A);
  B = SkipParenthesize(B);
  // do smt...
}
```

Рисунок 3.30 – Исправленный код

- V3063. A part of conditional expression is always true/false if it is evaluated.

Анализатор обнаружил потенциально возможную ошибку внутри логического условия. Часть логического выражения всегда истинно/ложно и оценено, как подозрительное (рисунок 3.31).

```
uint i = length;
while ((i >= 0) && (n[i] == 0)) i--;
```

Рисунок 3.31 – Код программы с ошибкой

Выражение "I >= 0" всегда верно, так как переменная имеет тип UINT" I". Таким образом, когда значение 'I' достигает нуля, цикл while не приостанавливает и принимает максимальное значение типа ' I ' uint. Дальнейшая попытка доступа к массиву 'n' приводит к предотвращению OverflowException. Правильный код изображен на рисунке 3.32:

```
int i = length;
while ((i >= 0) && (n[i] == 0)) i--;
```

Рисунок 3.32 – Исправленный код

- V3075. The operation is executed 2 or more times in succession.

Анализатор обнаружил потенциальную ошибку, связанную с тем, что одна из операций '!', '~', '-' или '+' повторяется два или более раз. Такая ошибка может произойти в случае опечатки. Такое дублирование операторов бессмысленно и может содержать ошибку. Пример кода показан на рисунке 3.33.

```
if (!(( !filter )))
{
  ....
}
```

Рисунок 3.33 – Пример кода с ошибкой

Скорее всего, такая ошибка возникла после проведения рефакторинга кода. Например, была удалена часть сложного логического выражения, а отрицание всего результата осталось. В итоге, получилось противоположное по смыслу выражение. Правильный вариант кода показан на рисунке 3.34.

```
if ( filter )
{
  ....
}
```

Рисунок 3.34 – Исправленный код

- V3139. Two or more case-branches perform the same actions.

Анализатор обнаружил ситуацию, когда в операторе switch разные метки case содержат одинаковые фрагменты кода. Часто это свидетельствует об избыточном коде, который можно улучшить объединением меток. Но часто часть кодов может быть причиной программирования copy-paste и являются реальными ошибками. Пример с избыточным кодом (рисунок 3.35):

```
switch (switcher)
{
  case 0: Console.Write("0"); return;
  case 1: Console.Write("0"); return;
  default: Console.Write("default"); return;
}
```

Рисунок 3.35 – Пример с ошибочным кодом

Действия для нескольких значений 'switcher' действительно могут быть одинаковыми, поэтому код можно написать более компактно (рисунок 3.36):

```
switch (switcher)
{
  case 0:
  case 1: Console.Write("0"); return;
  default: Console.Write("default"); return;
}
```

Рисунок 3.36 – Исправленный код

Если в исходном коде используется 'case expression', то объединить такие выражения под одно условие не получится (рисунок 3.37):

```
private static void ShowCollectionInformation(object coll, bool cond)
{
  switch (coll)
  {
    case Array arr:
      if(cond)
      {
        Console.WriteLine (arr.ToString());
      }
      break;
    case IEnumerable<int> arr:
      if(cond)
      {
        Console.WriteLine (arr.ToString());
      }
      break;
   }
}
```

Рисунок 3.37 – Исходный код с использованием 'case expression'

В таком случае, необходимо вынести общий код в метод, что облегчит дальнейшее редактирование и отладку.

3.5.3 Диагностика исходных кодов, написанных на Java

Диагностические сообщения исходных кодов, написанных Java, аналогичны предупреждениям кодов, написанных на C. Количество предупреждений составляет 77, с V6001 по V6078.

- V6027. Variables are initialized through the call to the same function. It's probably an error or un-optimized code.

Анализатор обнаружил потенциальную ошибку, найдя в коде инициализацию двух различных переменных одинаковыми выражениями. Анализатор считает опасными не все выражения, а только в которых используется вызов функций (либо слишком длинное выражение) (рисунок 3.38).

```
sz1 = s1.length();
sz2 = s1.length();
```

Рисунок 3.38 – Ошибочный код

Двум разным переменным присваивается один и тот же размер строки. Глядя на переменные 'sz1' и 'sz2' можно сделать вывод, что произошла опечатка. Корректный фрагмент кода будет выглядеть как на рисунке 3.39.

```
sz1 = s1.length();
sz2 = s2.length();
```

Рисунок 3.39 – Исправленный код

- V6044. Postfix increment/decrement is senseless because this variable is overwritten.

Анализатор обнаружил потенциальную ошибку, связанную с бессмысленным использованием постфиксного инкремента или декремента в выражении присвоения в эту же переменную (рисунок 3.40).

```
int i = 5;
// Some code
i = i++;
```

Рисунок 3.40 – Ошибочный код

В этом случае рост является бессмысленным и имеет значение 'i ' переменная'5'после выполнения этого кода.

Это связано с ростом Pstfix и тем, что декремент будет исполнен после расчета положительной операнды оператора назначения, а результаты расчета будут временно удалены. Таким образом, результаты роста или декремента Postfix вновь определяются результатом всего выражения.

Правильный пример может быть разным в зависимости от начальной задачи.

Это может быть ошибка, и на самом деле программист представляет выражение случайной переменной ' i ' 2 раза. Тогда правильный вариант может выглядеть как на рисунке 3.41:

```
int i = 5;
// Some code
q = i++;
```

Рисунок 3.41 – Исправленный код

- V6066. Passing objects of incompatible types to the method of collection.

Анализатор обнаружил потенциальную ошибку, связанную с вызовом метода коллекции, в который передали объект, тип которого не совпадает с типом коллекции. Анализатор выдает предупреждения на такие функции как remove, contains, removeAll, containsAll, retainAll и т.д (рисунок 3.42).

```
List<String> list = ...;
Integer index = ...;
...
list.remove(index);
```

Рисунок 3.42 – Пример ошибочного кода

Здесь хотели удалить объект из списка по индексу, но не учли, что индекс не примитивный целый тип, а объект типа 'Integer'. Будет использоваться перегруженный метод 'remove', который ожидает на вход объект, а не 'int'. Объекты 'Integer' и 'String' несовместимы, и поэтому использование метода будет ошибочным.

Если нет возможности изменить тип переменной "index", то можно исправить код следующим образом (рисунок 3.43):

```
List<String> list = ...;
Integer index = ...;
...
list.remove(index.intValue());
```

Рисунок 3.43 – Исправленный код

- V6077. A suspicious label is present inside a switch(). It is possible that these are misprints and 'default:' label should be used instead.

Анализатор обнаружил потенциальную ошибку внутри оператора switch. Используется метка с именем похожим на 'default'. Возможно это опечатка (рисунок 3.44).

```
int c = 10;
double weightCoefficient = 0;
switch(c){
  case 1:
    weightCoefficient += 3 * (/*math formula #1*/);
  case 2:
    weightCoefficient += 7 * (/*math formula #2*/);
  defalt:
    weightCoefficient += 0.42;
}
```

Рисунок 3.44 – Ошибочный код

Кажется, после того, как этот код отработает, значение переменной 'weightCoefficient' будет 0.42. Но на самом деле значение 'weightCoefficient' останется равным нулю. Дело в том, что 'defalt' это метка, а не оператор 'default'. Исправленный вариант кода на рисунке 3.45.

```
int c = 10;
double weightCoefficient = 0;
switch(c){
  case 1:
    weightCoefficient += 3 * (/*math formula #1*/);
  case 2:
    weightCoefficient += 7 * (/*math formula #2*/);
  default:
    weightCoefficient += 0.42;
}
```

Рисунок 3.45 – Исправленный код

Проведенный анализ кодов на языках программирования C++, C#, Java, а также полученные результаты показали, что при помощи PVS – Studio наибольшее количество предупреждений было выведено на C++. Результаты сравнения диагностических сообщений показаны в таблице 3.1.

Таблица 3.1- Результаты сравнений

Предупреждения на C++	Предупреждения на C#	Предупреждения на Java
556	152	77

3.6 Справочная система и техническая поддержка

PVS-Studio имеет широкую справочную систему для диагностических сообщений. Эта база данных доступна как при работе с PVS Studio, так и на официальном сайте. Различные диагностические сообщения сопровождаются примерами кодов с аналогичными ошибками, описанием проблемы и возможными поправками.

Для описания конкретного диагноза достаточно нажать левую кнопку мыши по диагностическому номеру в окне сообщения. Эти номера считаются гиперссылкой.

Поддержка PVS Studio предоставляется по электронной почте. Поскольку разработчики аналитиков выступают непосредственно с клиентами, они могут быстро получить ответы на различные вопросы.

3.7 Сравнение анализатора Visual Studio и PVS – Studio

Для сравнения двух анализаторов необходимо взять один код, то есть исходный код одного продукта, и провести анализ на двух аналогичных анализаторах.

CruiseControl.NET — это сервер автоматической непрерывной интеграции, реализованный с использованием .NET Framework. Исходный код CruiseControl.NET доступен на GitHub. Проект уже некоторое время не развивается и не поддерживается, хотя, в недалёком прошлом пользовался определённой популярностью. Это не помешает применить его с целью сравнения анализаторов, скорее, даже наоборот, внесёт некий элемент стабильности в исследование. Дополнительным плюсом является небольшой размер CruiseControl.NET: проект содержит около 256 тысяч строк кода.

Проверка кода проекта при помощи встроенного в Visual Studio анализатора заняла всего пару минут. Сразу после этого результаты имеют следующий вид (никакие фильтры не включены) (рисунок 3.46):

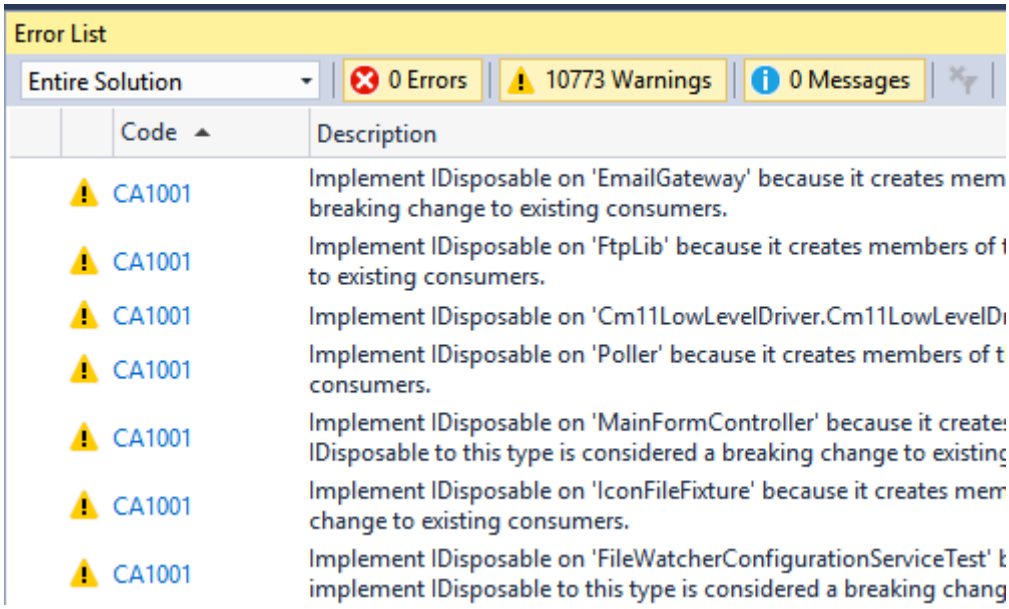

Рисунок 3.46 – Диагностические сообщения Visual Studio

Анализатор выдал 10773 предупреждения. Для начала нужно исключить из данного списка предупреждения по проекту «UnitTests» при помощи фильтра (рисунок 3.47):

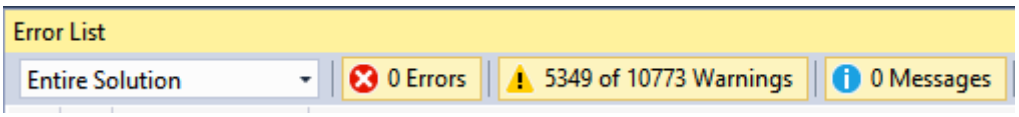

Рисунок 3.47 – Исключение тестовых предупреждений

После исключения тестовых предупреждений всего остается более 5 тысяч предупреждений.

Эти предупреждения относятся к следующим группам:

– Microsoft.Design: CA10XX (диагностик: 40, предупреждений: 1637);

– Microsoft.Globalization: CA13XX (диагностик: 7, предупреждений: 1406);

– Microsoft.Interoperability: CA14XX (диагностик: 2, предупреждений: 10);

– Microsoft.Maintainability: CA15XX (диагностик: 3, предупреждений: 74);

– Microsoft.Mobility: CA16XX (диагностик: 1, предупреждений: 1);

– Microsoft.Naming: CA17XX (диагностик: 17, предупреждений: 1071);

– Microsoft.Performance: CA18XX (диагностик: 15, предупреждений: 489);

– Microsoft.Portability: CA19XX (диагностик: 1, предупреждений: 4);

– Microsoft.Reliability: CA20XX (диагностик: 4, предупреждений: 158);

– Microsoft.Globalization, Microsoft.Security: CA21XX (диагностик: 5, предупреждений: 48);

– Microsoft.Usage: CA22XX (диагностик: 18, предупреждений: 440).

Также было выдано некоторое количество предупреждений компилятора.

Пример ошибок и дефектов, которые вывел Visual Studio:

Microsoft.Mobility

CA1601 you change the call to " timer.Timer (double) " FileChangedWatcher method.FileChangedWatcher (params string []) ' set the value of the timer interval is greater than or equal to one second. Major FileChangedWatcher.cs 33. The timer interval.

Данная ошибка указывает что интервал установлен на меньше одной секунды. И данная ошибка может привести к переполнению буфера, последствием чего может стать неправильная работа всей системы в целом (рисунок 3.48).

```
public FileChangedWatcher(....)
{
  ....
  timer = new Timer(500);
  ....
}
```

Рисунок 3.48 - Предупреждение о том, что для таймера установлен интервал меньше одной секунды

Microsoft.Globalization,Microsoft.Security
CA2101 to reduce security risks, marshal the parameter 'lpszDomain' as Unicode of dllimport.CharSet to CharSet.Unicode, or by explicit dissemination parameter as UnmanagedType.LPWStr. If you need to specify this string as ANSI or system-dependent MarshalAs, specify the MarshalAs and set BestFitMapping = false; for added security, set ThrowOnUnmappableChar=true. basic identity change.cs 100 . Предупреждение о том, что не указан тип маршалинга для строковых аргументов (рисунок 3.49):

```
[DllImport("advapi32.dll", SetLastError = true)]
private static extern bool LogonUser(
        string lpszUsername,
        string lpszDomain,
        string lpszPassword,
        int dwLogonType,
        int dwLogonProvider,
        ref IntPtr phToken);
```

Рисунок 3.22 – Предупреждение о том, что не указан тип маршалинга для строковых аргументов

Данный дефект необходимо исправить путем указания атрибутов:
[DllImport("advapi32.dll", SetLastError = true,
CharSet = CharSet.Unicode)].

Проверка кода проекта при помощи анализатора PVS – Studio заняло ровно минуту. Сразу после этого результаты имеют вид (никакие фильтры не включены) (рисунок 3.50):

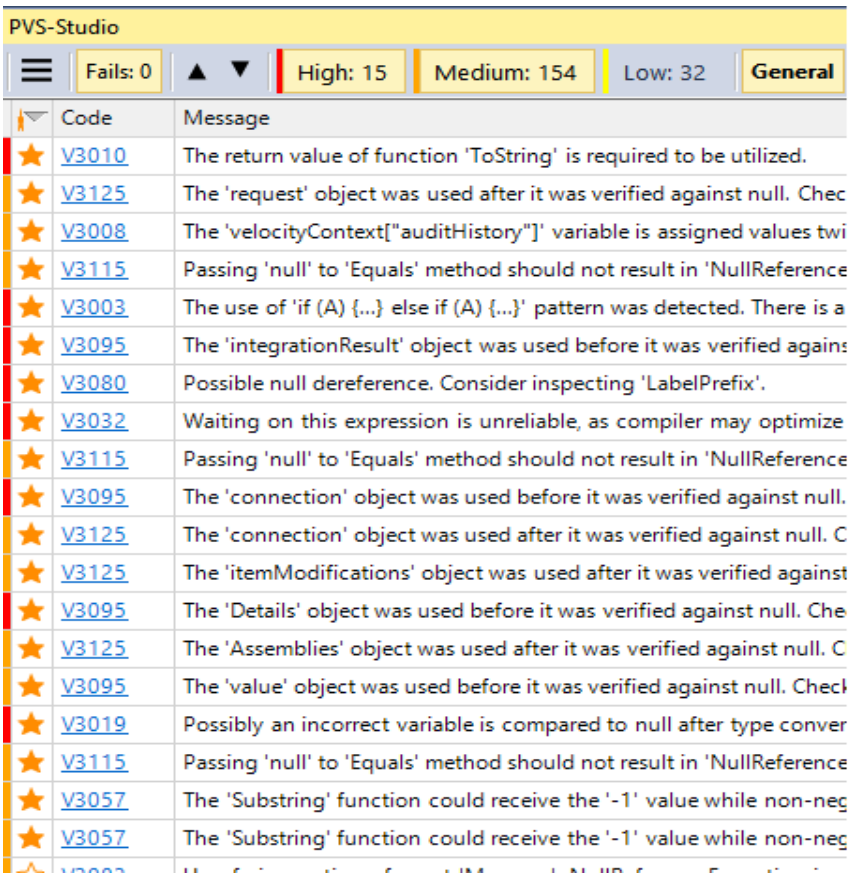

Рисунок 3.50 – Диагностические сообщения, которые вывел PVS-Studio

Анализатор выдал 198 предупреждений. И них 45 было получено для проекта «UnitTests», а ещё 32 предупреждения имеют низкий приоритет (среди них сложно найти реальные ошибки). Итого — 121 сообщение для анализа, на который я потратил 30 минут. В результате было выявлено 19 ошибок.

Вот пример одной из них: V3003 The use of 'if (A) {...} else if (A) {...}' pattern was detected. There is a probability of logical error presence. Check lines: 120, 125. CCTrayLib CCTrayProject.cs 120 (рисунок 3.51).

```
public override bool Equals(object obj)
{
  ....
  if ((buildServer != null) &&
      (objToCompare.buildServer != null))
  {
    // If both instances have a build server then compare the build
    // server settings
    isSame = string.Equals(buildServer.Url,
             objToCompare.buildServer.Url);
  }
  else if ((buildServer != null) &&
           (objToCompare.buildServer != null))
  {
    // If neither instance has a build server then they are the same
    isSame = true;
  }
  ....
}
```

Рисунок 3.51 – Дефект, выведенный PVS – Studio

Оба блока *if* содержат одинаковое условие. Допущена серьёзная ошибка, влияющая на логику работы программы и приводящая к неожиданному результату.

Результаты сравнительного анализа представлены в таблице 3.2.

Таблица 3.2 – Результаты сравнительного анализа

Анализатор	Всего предупреждений	Тестовые предупреждения	Уязвимости
Visual Studio	10773	5424	5
PVS – Studio	198	45	19

4 Анализ рисков информационной безопасности

4.1 Вычислительная часть

В первую очередь для расчета рисков информационной безопасности были определены защищаемые активы: операционная система, сетевая инфраструктура и среда разработки (PVS-Studio встраивается в среду разработки и только потом начинает функционировать, в данном дипломном проекте средой разработки служит Visual Studio).

Риски информационной безопасности были рассчитаны с учетом особенностей темы работы: применение инструментальных средств для выявления уязвимостей и закладок ПО. Были рассчитаны риски для вышеперечисленных активов (незащищенных). После этого в качестве мер обработки данных рисков были указаны меры защиты, рассмотренные в

данной работе. Расчет остаточных рисков был произведен с учетом данных защитных мер.

Для анализа рисков был выбран алгоритм из стандарта ISO-27005. Расчет по первому алгоритму (по двум шкалам) производится на основе приложения Е стандарта ISO-27005.

Таблица 4.1 - Ценность активов, уровни угроз и уязвимостей

Степень вероятности возникновения угрозы		Низкая			Средняя			Высокая		
Простота использования		Н	С	В	Н	С	В	Н	С	В
Ценность активов	0	0	1	2	1	2	3	2	3	4
	1	1	2	3	2	3	4	3	4	5
	2	2	3	4	3	4	5	4	5	6
	3	3	4	5	4	5	6	5	6	7
	4	4	5	6	5	6	7	6	7	8

Простой общий рейтинг рисков:

- низкий риск: 0-2;
- средний риск: 3-5;
- высокий риск: 6-8.

Остаточный риск – это риск, который остается после мер по контролю над рисками. Расчет остаточного риска осуществляется по формуле, представленной на рисунке 4.1.

Остаточный риск = Первичный риск — Влияние мероприятий по контролю над рисками

Рисунок 4.1 – Формула расчета остаточного риска

Таблица4. 2 – Анализ рисков информационной безопасности

№	Угрозы	Уязвимости	Максимальный уровень риска	Меры по обработке риска	Остаточный уровень риска	Комментарии, ресурсы, ответственный
Актив 1. Операционная система						
1	Изменение процессов и режимов работы ОС. Следствие: некорректная работа программ	Неправильная конфигурация средств защиты	7	Конфигурация средств защиты; Постоянный контроль корректности функционирования операционной системы, особенно ее подсистемы защиты	2	Системный администратор
2	Несанкционированный доступ к анализируемому коду программного продукта	Неправильная настройка политики безопасности, ошибки администратора системы	7	Конфигурация данных и политики безопасности ОС	2	Системный администратор

Продолжение таблицы 4.2

№	Угрозы	Уязвимости		Меры по обработке риска		
3	Сканирование файловой системы злоумышленником. Следствие: копирование или удаление программ	Отсутствие политик пользователей. Ошибки и недокументированные возможности программного обеспечения	8	Организация и поддержание адекватной политики безопасности. Все загруженные файлы хранить в базе данных, а не в файловой системе.	3	Системный администратор
4	Кража ключевой информации. Следствие: модификация алгоритмов анализа кода	Халатность или невнимательность сотрудников. Наличие «служебных входов», позволяющие обходить систему защиты	8	Осведомление пользователей ОС о необходимости соблюдения мер безопасности при работе с ОС и контроль за соблюдением этих мер	3	Сотрудник отдела безопасности
Актив2. Сетевая инфраструктура						
5	Искажение передаваемых данных	Недостаточная защита передаваемого трафика, плохая конфигурация брандмауэра	6	Шифрование трафика, конфигурация брэндмауэра	2	Сетевой администратор
6	Похищение конфиденциальной информации (логины/пароли) посредством сниффера	Плохая конфигурация брандмауэра, передача незашифрованных данных в текстовом формате	6	Конфигурация брандмауэра, шифрование передаваемых данных и использование антиснифферов	2	Сетевой администратор
7	Перехват сессии и получение доступа к общим сетевым ресурсам	Отсутствие контроля сетевого доступа	6	Привязывать сессию к ip адресу компьютера. Привязывать сессию к юзер агенту браузера. Шифровать передаваемые в сессию параметры.	2	Сетевой администратор
8	Модификация передаваемого фрагмента кода	Незащищенный доступ к сайту	7	Фильтрация трафика. Привязывать сессию к юзер агенту браузера.	2	Сетевой администратор
Актив3. Среда разработки Visual Studio						
9	Внедрение вредоносного кода	Слабые меры антивирусной защиты и разграничения доступа	6	Усиление антивирусной защиты	1	Системный администратор
10	Внесение несанкционированных изменений в программу	Отсутствие фильтрации атрибутов в сообщениях и комментариях, оставляемых пользователями, и, как следствие, возможность внедрить текст с атрибутами скрипта	6	Фильтрация атрибутов; Защита от атак на уровне приложения	1	Системный администратор

При первичной оценке риски оказались неприемлемыми (от 6 до 8 по 8-ми балльной шкале), поэтому для всех рисков были описаны защитные меры. После введения мер для обработки рисков риски были пересчитаны, получены остаточные риски. Все риски, оставшиеся после перерасчета с учетом защитных мер, стали приемлемыми (от 0 до 2 по 8-ми балльной шкале).

4.2 Моделирование угроз

Диаграммы взаимосвязей компонентов анализа рисков (на базе вышеуказанной таблицы анализа рисков) реализуются в программе CORAS.

На рисунке 4.1 представлена диаграмма защищаемых активов. Они разделены на категории: «Оборудование и аппаратура», «Информационные ресурсы» и «Программные средства». К примеру, актив «Сетевая инфраструктура» входит в категорию «Оборудование и аппаратура» и «Информационные ресурсы», активы «Операционная система» и «Среда разработки» - в категорию «Информационные ресурсы» и «Программные средства».

На рисунке 4.2 представлена диаграмма модели угроз. Элементы диаграммы слева направо: источники угроз, уязвимости, этапы реализации угроз, последствия реализации угроз (инциденты), понесшие от реализации угрозы ущерб активы. К примеру, источник угроз «Злоумышленник», используя уязвимость «Плохая конфигурация брэндмауэра», осуществляет «Похищение конфиденциальных данных посредством сниффера (логины/пароли, анализируемые коды)» и получает доступ к общему сетевому ресурсу. Последствием реализации данной угрозы может послужить «Копирование исходных кодов анализируемых программ» Актив, на который направлена угроза – «Сетевая инфраструктура» и «Операционная система».

На рисунке 4.3 представлена диаграмма модели угроз с учетом вероятности возникновения инцидента. Ее следует читать также, как и диаграмму, представленную на рисунке 3, только добавлен параметр вероятности возникновения инцидентов (высокая, средняя, низкая). К примеру, инцидент «Модификация исходных кодов анализируемых программ» имеет высокую вероятность возникновения.

На рисунке 4.4 представлена диаграмма рисков с характеристиками влияния угроз. Элементы диаграммы слева направо: источники угроз, уязвимости, способы реализации угроз, степень влияния реализации угроз, понесшие от реализации угрозы ущерб активы. К примеру, «Сотрудник», используя уязвимость «Отсутствие фильтрации атрибутов», осуществляет угрозу «Внесение несанкционированных изменений в программу», это сильно повлияло на атакуемые активы – среду разработки.

На рисунке 4.5 представлена диаграмма модели угроз с учетом защитных мер. Ее следует читать так же, как и диаграмму, представленную на рисунке 3, с единственным отличием: между уязвимостями и способами

реализации угроз добавлены защитные меры для уменьшения рисков. К примеру, для уязвимости «Наличие служебных входов» внедрена защитная мера «Соблюдение мер безопасности сессий, политики пользователей».

На рисунке 4.6 представлена диаграмма недопустимых рисков. Она построена на базе диаграммы, представленной на рисунке 5, однако на данной диаграмме показаны только те риски, которые имеют высокую степень влияния угроз.

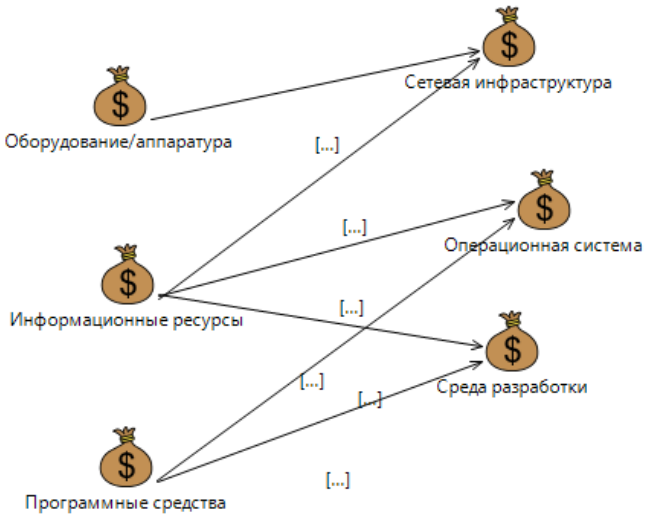

Рисунок 4.1 –Перечень активов

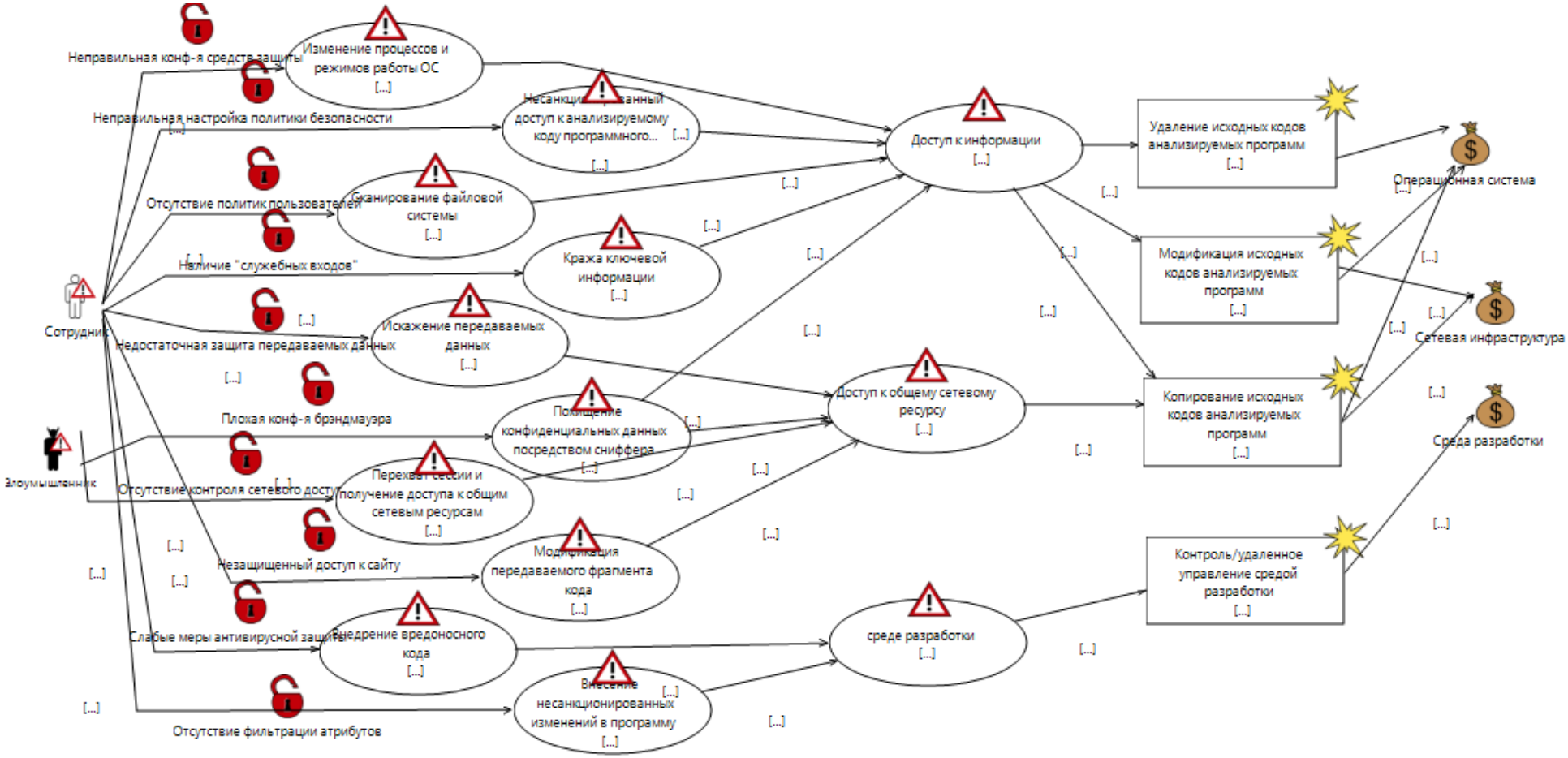

Рисунок 4.2 – Модель угроз

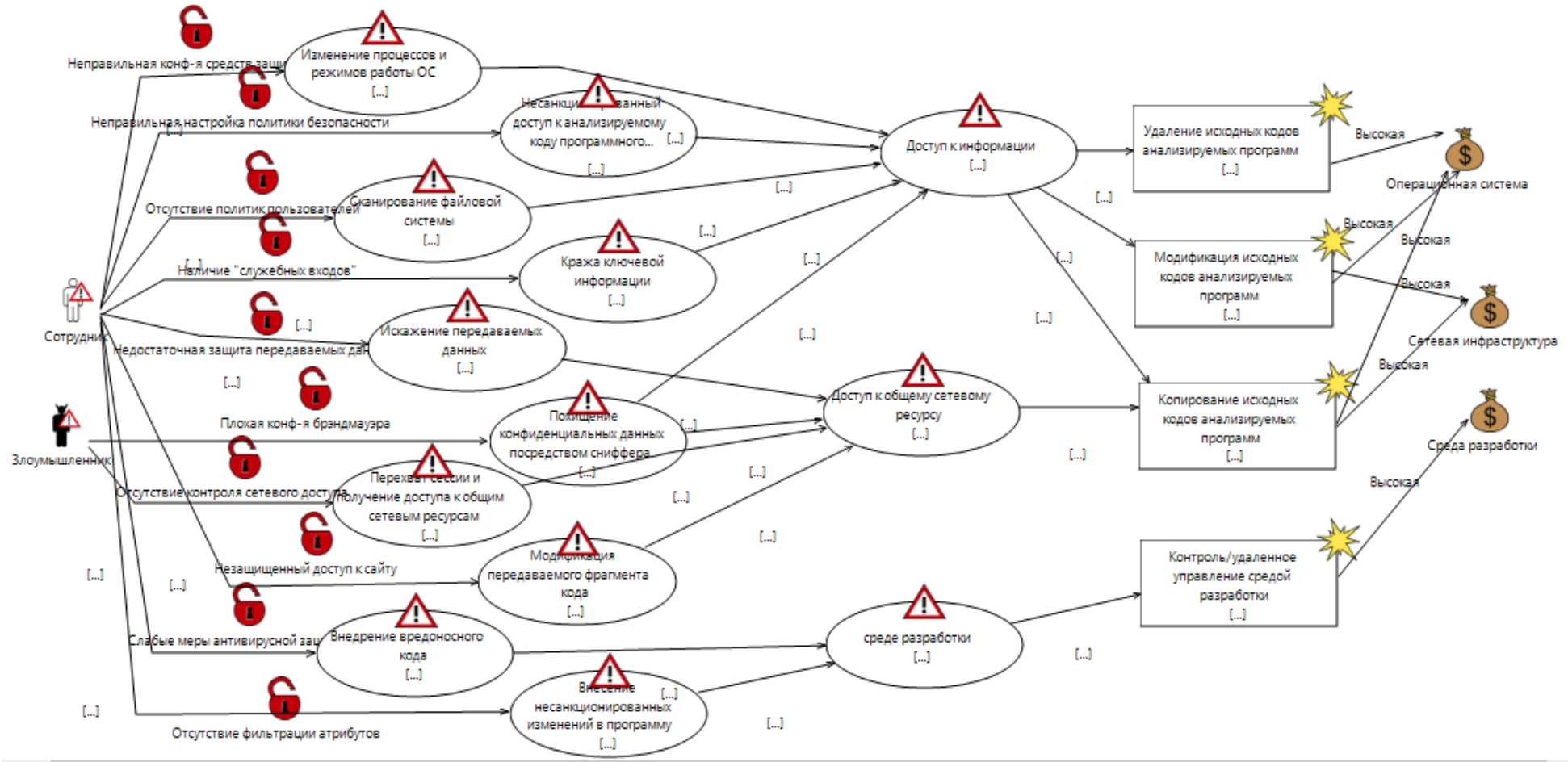

Рисунок 4.3 – Модель угроз с учетом вероятности возникновения инцидента

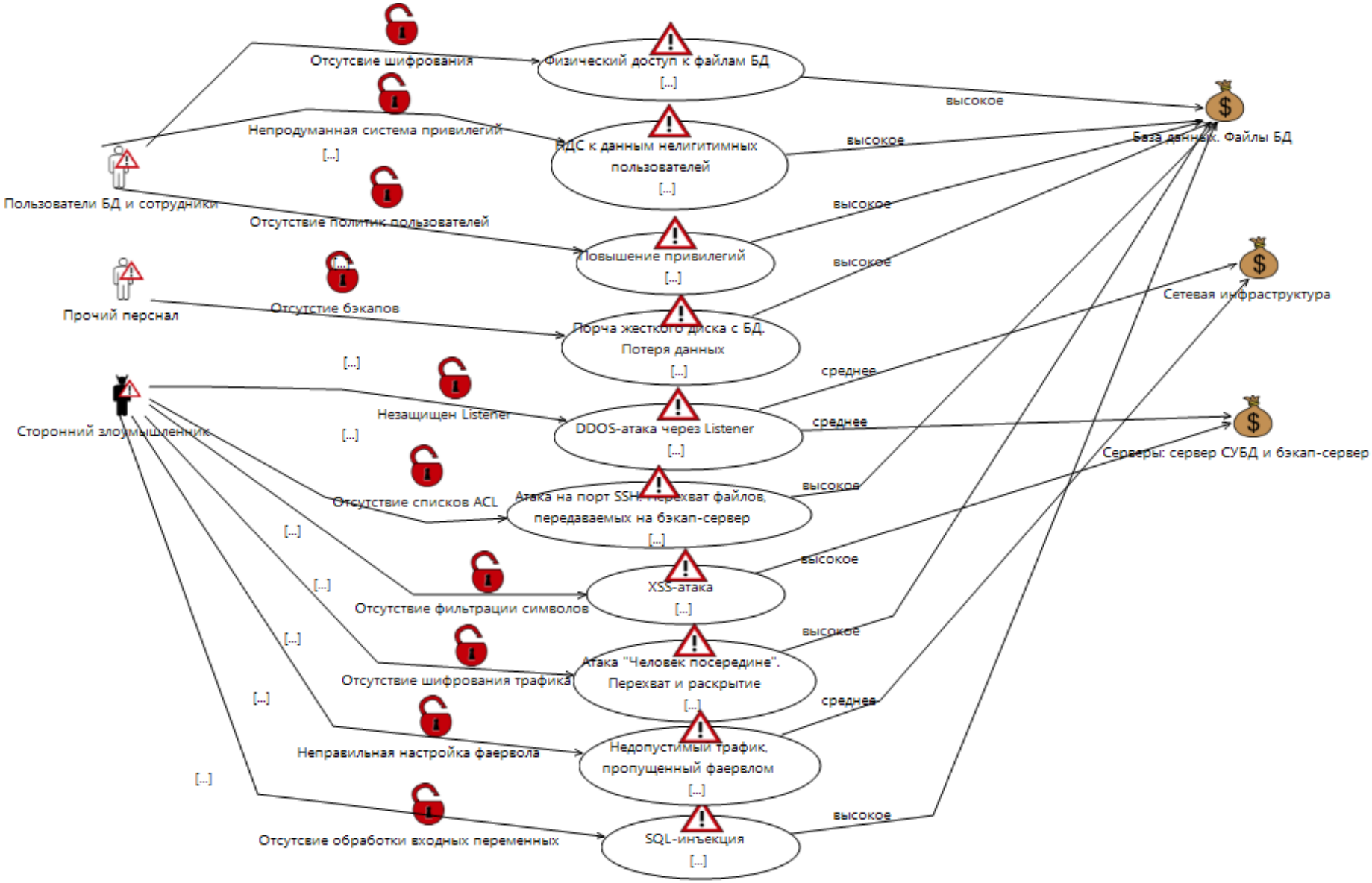

Рисунок 4.4 – Диаграмма рисков с характеристиками влияния угроз

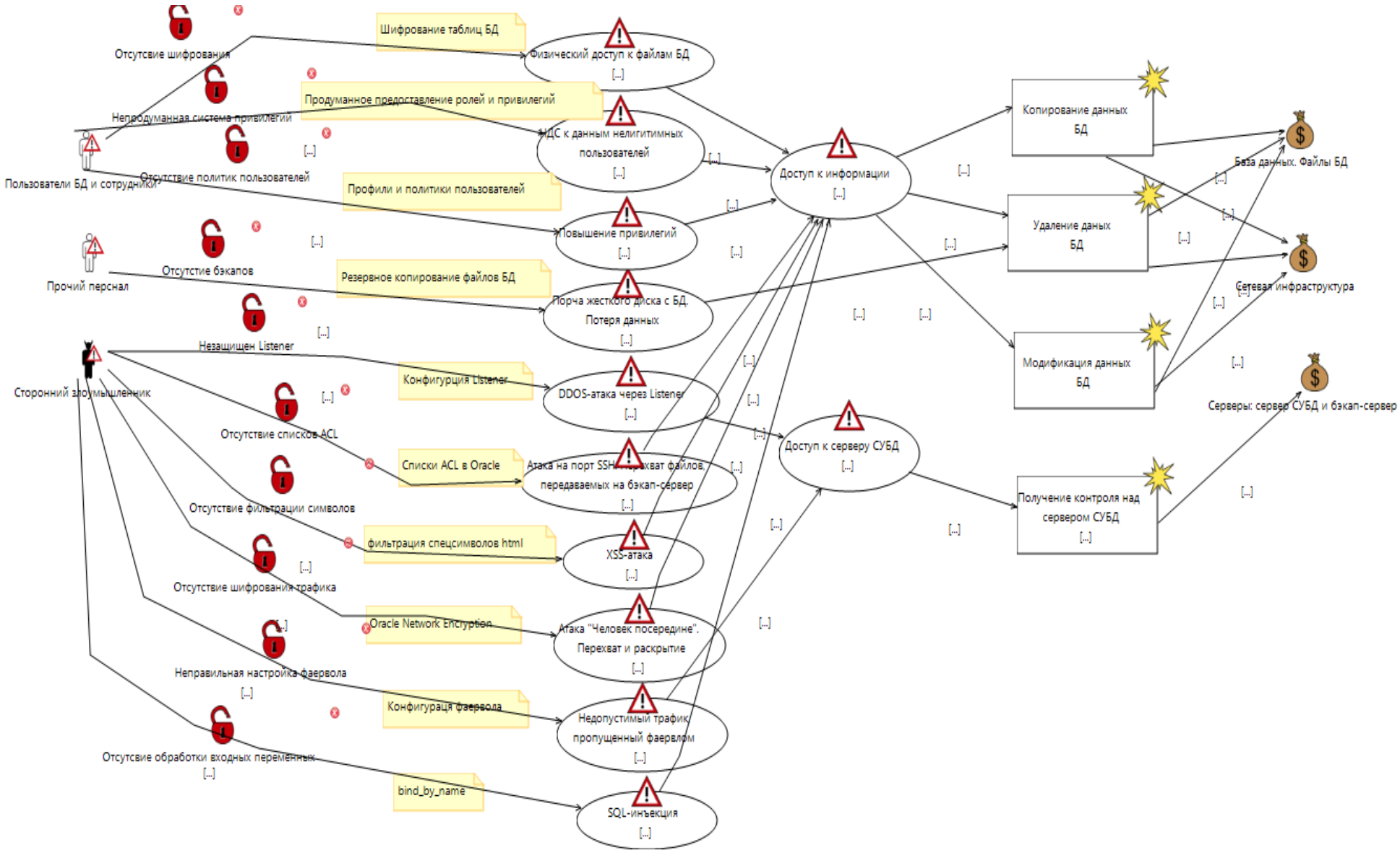

Рисунок 4.5 – Модель угроз с учетом защитных мер

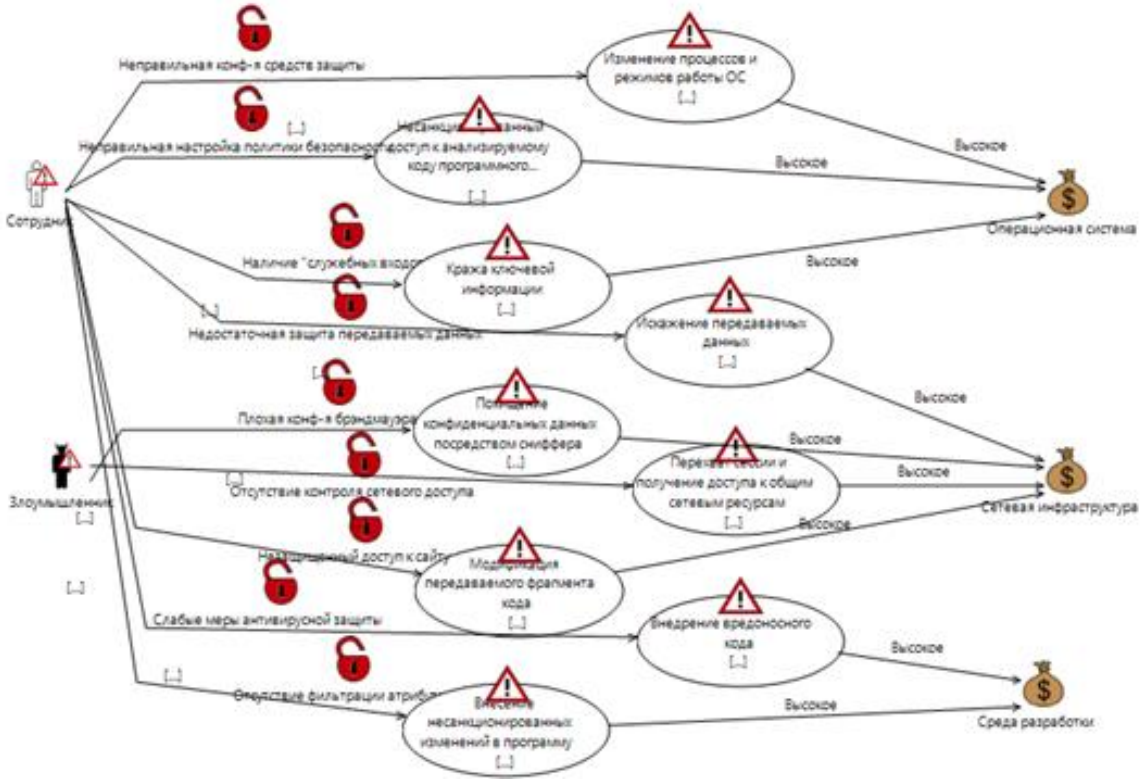

Рисунок 4.6 – Диаграмма недопустимых рисков

4.3 Выводы по разделу «Оценка рисков»

Были описаны и рассчитаны основные риски информационной безопасности для незащищенных активов «Операционная система», «Сетевая инфраструктура» и «Среда разработки». В результате расчета все риски оказались неприемлемыми (от 6 до 8 по 8-ми балльной шкале), поэтому для всех рисков были описаны защитные меры. После введения мер для обработки рисков риски были пересчитаны, получены остаточные риски. Все риски, оставшиеся после перерасчета с учетом защитных мер, стали приемлемыми (от 0 до 2 по 8-ми балльной шкале). После внедрения мер для обработки рисков риски активов «Операционная система» и «Сетевая инфраструктура» были уменьшены (в среднем) в 3 раза, риски актива «Среда разработки» уменьшились в 6 раз.

5 Безопасность жизнедеятельности

5.1 Анализ условий труда на рабочем месте

В данном дипломном проекте я исследовала эффективность инструментальных средств, чья работа основана на выявлении уязвимостей и закладок программного обеспечения.

Данный раздел дипломного проекта посвящен рассмотрению следующих вопросов:

- Анализ условий труда инженера – программиста;
- Обзор вредных особенностей работы инженера – программиста;
- Оптимальные условия труда инженера – программиста;

Немаловажную роль в процессе работы и выполнении трудовых обязанностейиграетбезопасностьчеловека.Всвязисчемсобственноиначала развиваться такая наука как безопасность жизнедеятельности. Цели и задачи БЖД чаще всего подразумевают выбор принципов защиты, разработки и рациональное использование способов защиты человека и природы от техногенных источников и стихийных явлений. Охрана здоровья трудящихся, обеспечение безопасности условий труда, ликвидация профессиональных заболеваний и производственного травматизма составляет одну из главных забот человеческого общества. Обращается внимание на необходимость широкого применения прогрессивных форм научной организации труда, сведения к минимуму ручного, малоквалифицированного труда, создания обстановки, исключающей профессиональные заболевания и производственныйтравматизм.Нарабочемместедолжныбытьпредусмотрены меры защиты от возможного воздействия опасных и вредных факторов производства. Рабочее место инженера-программиста не является исключением, будь то серверная или рабочее место в кабинете. Оба этих пункта описываются одинаковыми проблемами и опасностями для человека.

5.1.1 Обзор вредных особенностей работы инженера-программиста

Монитор представляет из себя электронную пушку, то есть он заряжен отрицательнымиионами,всвоюочередьвнемониторапроисходитнакопление положительно заряженных частиц. При соблюдении баланса между ними человек чувствует себя хорошо, но при накоплении избытка ионов перед экраном возможны следующие последствия:

- Бессонница;
- Головная боль;
- Усталость глаз;
- Раздражение кожи.

Неправильно расположенная клавиатура или компьютерная мышь стимулирует развитие запястного синдрома - болезненного поражения срединного нерва запястья. Данная болезнь возникает при неправильном расположении руки, в результате которой кисть образует изгиб между запястьем и предплечьем. Худший вариант развития данного события может

привести к ампутации руки.

Повышенный уровень шума вызывает трудности в распознавании цветовыхсигналов,снижаетбыстротувосприятияцветовыхсигналов,снижает быстроту восприятия цвета, остроту зрения, зрительную адаптацию, нарушает восприятие визуальной информации, снижает способность быстро и четко выполнять координированные действия, уменьшает на 5-10% производительность труда. Длительное воздействие повышенного уровня шума с уровнем звукового давления 90 Дб снижает производительность труда на 30-60% [8]. Поэтому важно сохранять оптимальный уровень шума в помещении, особенно от работы электротехнических устройств. Более комфортная среда стимулирует работать более продуктивно.

Любое техническое оборудование будет выделять тепло в свое рабочее время-это естественный физический эффект. Повышенная температура окружающей среды приводит к быстрой утомляемости, снижает скорость приема зрительной и слуховой информации, общее торможение человека в связи с нарушением сердечной деятельности (увеличение частоты сердца), изменением артериального давления. Влияние этих неблагоприятных факторов приводит к снижению работоспособности, вызванной развитием усталости. Возникновение и развитие усталости связано с изменениями, происходящими в процессе работы центральной нервной системы, тормозными процессами в оболочке головного мозга.

Одним из основополагающих факторов рабочего места является освещенность. Недостаточность освещения приводит к напряжению зрения, ослабляет внимание, приводит к наступлению преждевременной утомленности. Чрезмерно яркое освещение вызывает ослепление, раздражение и резь в глазах.

Неправильное направление света на рабочем месте может создавать резкие тени, блики, дезориентировать работающего. Все эти причины могут привести к несчастному случаю или профзаболеваниям, поэтому столь важен правильный расчет освещенности. Нормальная освещенность рабочего места составляет 300 лк [9].

5.1.2 Оптимальные условия труда инженера-программиста

Создание благоприятных условий труда и правильное эстетическое оформление рабочих мест имеет большое значение для облегчения труда и повышения его производительности. Окраска помещений и мебели должна способствоватьсозданиюблагоприятныхусловийдлязрительноговосприятия, хорошего настроения. В служебных помещениях, в которых выполняется

однообразная умственная работа, требующая значительного нервного напряжения и большого сосредоточения, окраска должна быть спокойных тонов - малонасыщенные оттенки холодного зеленого или голубого цветов.

Рабочее место–это часть пространства, в котором инженер осуществляет трудовую деятельность, и проводит большую часть рабочего времени. Рабочее место, хорошо приспособленное к трудовой деятельности инженера, правильно и целесообразно организованное, в отношении пространства, формы, размера обеспечивает ему удобное положение при работе и высокую производительность труда при наименьшем физическом и психическом напряжении. Таким образом, решение проблемы обеспечения здоровых и безопасныхусловийтрудаинженера-программистаявляетсяоднойизнаиболее важных задач в процессе производственной деятельности. При создании здоровых условий труда снижается утомляемость при работе с компьютером, понижаетсярискпроизводственноготравматизма.Следовательно,повышается эффективность производственного процесса. Конечно, выполнение только перечисленных здесь требований еще не является залогом безопасной и комфортной деятельности человека. При организации трудового процесса обязательно следует учитывать такие параметры, как организация правильной освещенности рабочего места, нормирование уровня шума в помещении, уровня электромагнитных излучений. Кроме того, обязателен контроль пожарной и электробезопасности рабочего помещения.

5.2 Расчет контурного защитного заземления ПК

Контур заземления выполняют из стальных стержней, уголков, некондиционных труб и прочего. В траншее глубиной до 0,7 м вертикально забиваются стержни (трубы, уголки и др.), выступающие из земли верхние концы соединяются сваркой внахлест стальной полосой или прутком (рисунок 5.1).

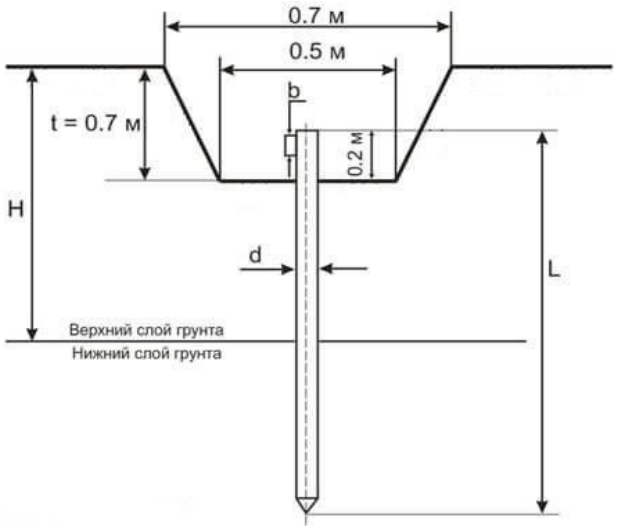

Рисунок 5.1 – Заземляющий стержень

Расчет заземляющего устройства сводится к определению числа вертикальных заземлителей ($R_в$) и длины соединительной полосы [10].

Заглубление полосы принимается равным 0,7 м, длину стержня 7 м, диаметр 0,15 м, расстояние между стрежнями равно 7 м и расположены треугольником между собой. (рисунок 5.2).

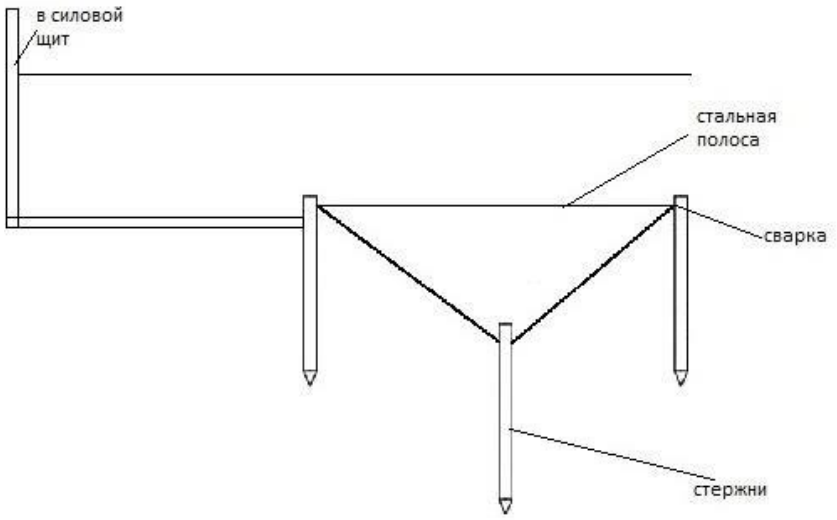

Рисунок 5.2 – Схема заземления

Удельное сопротивление грунта принимается равным 100 Ом*м, так как тип грунта суглинок (таблица 5.2).

Таблица 5.1 Удельное сопротивление грунта

Грунт	Удельное сопротивление р, Ом*м	Грунт	Удельное сопротивление р,Ом*м
Песок при глубине залегания вод менее 5 м	500	Садовая земля	40
Песок при глубине залегания вод менее 6 и 10 м	1000	Чернозем	50
Супесь водонасыщенная (текучая)	40	Кокс	3
Супесь водонасыщенная влажная (пластинчатая)	150	Гранит	1100

Продолжение таблицы 5.1

Грунт	Удельное сопротивление р, Ом*м	Грунт	Удельное сопротивление р,Ом*м
Супесь водонасыщенная слабовлажная (твердая)	300	Каменный уголь	130
Глина пластичная	20	Мел	60
Глина полутвердая	60	Суглинок влажный	30
Суглинок	100	Мергель глинистый	50
Торф	20	Известняк пористый	180

Сопротивление вертикального одиночного заземлителя согласно формуле:

$$R0 = \frac{p}{2\pi \cdot L}\left(\ln\frac{2L}{d} + 0{,}5\ln\frac{4T+L}{4T-L}\right), \quad (5.1)$$

и равна:

$$R0 = \frac{100}{6{,}28 \cdot 7}\left(\ln\frac{2 \cdot 7}{0{,}15} + 0{,}5\ln\frac{4 \cdot 4{,}2+7}{4 \cdot 4{,}2-7}\right) = 11{,}32 \text{ Ом}.$$

Т – заглубление горизонтального заземлителя и вычисляется по формуле:

$$T = \frac{l}{2} + t, \quad (5.2)$$

И равно:

$$T = \frac{7}{2} + 0{,}7 = 4{,}2 \text{ м} \ .$$

Количество вертикальных заземлителей без учета сопротивления соединительной полосы определяется поформуле:

$$n_0 = \frac{R0 \cdot \varphi}{Rн}, \quad (5.3)$$

Где φ – сезонный климатический коэффициент. Для вертикально заземляющегоэлектродавовторойклиматическойзонепринимаемегоравным 1,7.

$R_н$ – нормируемое сопротивление растекания тока заземляющего устройства. Оно не должно превышать 30 В, при удельном сопротивлении до 100 Ом. Принимаем значение равным 5 В [10].

Количество вертикальных заземлителей равно:

$$n0 = \frac{11{,}32 \cdot 1{,}7}{5} = 3{,}84 \text{ шт.}$$

Сопротивление растекания тока для горизонтального заземлителя вычисляется поформуле:

$$Rг = 0{,}366 \left(\frac{p \cdot \psi}{L_r \cdot n_r} \cdot lg \frac{2 \cdot {L_r}^2}{b \cdot t} \right).$$

Где $L_г$ - Длина горизонтального заземлителя. Исходя из формулы:

$$L_r = a \cdot (n_0 - 1), \quad (5.4)$$

Она равна:

$$L_r = 7 \cdot (4 - 1) = 21 \text{ м.}$$

Сопротивление растекания тока равна:

$$Rг = 0{,}366 \left(\frac{100 \cdot 3{,}5}{21 \cdot 0{,}45} \cdot lg \frac{2 \cdot 21^2}{2 \cdot 0{,}7} \right) = 37{,}82 \text{ Ом.}$$

Определим сопротивление вертикального заземлителя с учетом сопротивления растеканию тока горизонтальных заземлителей поформуле:

$$R_в = \frac{R_г \cdot R_н}{R_г - R_н}, \quad (5.5)$$

Следовательно, сопротивление вертикального заземлителя равна:

$$R_в = \frac{37{,}82 \cdot 5}{37{,}82 - 5} = 5{,}76 \text{ Ом.}$$

Полное количество вертикальных заземлителей определяется по формуле:

$$n = \frac{R_0}{R_в \cdot n_в}, \quad (5.6)$$

И равно:

$$n = \frac{11{,}32}{5{,}76 \cdot 0{,}69} = 2{,}84 \text{ шт.}$$

Коэффициент использования показывает, что токи, распространяющиеся

из отдельных мест, влияют друг на друга, при разном расположении последних. При параллельном соединении между собой распределяющиеся токи представляют собой одно заземлители взаимное влияние друг на друга, поэтому наиболее близко расположенные заземляющие стержни имеют общее сопротивление цепи большого заземления друг на друга. Количество земель, полученных в результате расчетов, округляется до ближайшего большого числа.

Все данные были получены путем анализа специальной литературы. Значения таких параметров, как∑, PG, установлены в соответствии с установленными критериями выбора коэффициентов эксплуатации заземляющих устройств и сезонно-климатическим коэффициентом почвенного сопротивления.

5.3 Расчет шума

Основными источниками шума на автоматизированном рабочем месте с использованием ПК – шум работы систем охлаждения и шум работы периферийных устройств.

Допустимыеуровнизвуковогодавленияиуровнязвуканарабочемместе операторов ПК устанавливаются в соответствии с требованиями СанПиН 2.2.4/2.1.8.562-96 [11].

В соответствии с требованиями СНиП 2.2.4./2.1.8.562-96 считается допустимым 50дБА, а в помещениях с шумными агрегатами вычислительных систем (сканеры и т.п.) не превышает 80 дБА.

Для снижения уровня шума и вибрации рекомендуется применять следующие методы:

– использовать звукопоглощающие материалы с максимальными коэффициентами звукопоглощения для отделки стен и потолкапомещений;

– предусмотреть акустическую обработкупомещения.

В качестве дополнительного звукопоглощающего эффекта можно использовать однотонные занавески из плотной ткани, повешенные на расстоянии 15-20 см отображения.

В качестве дополнительного звукопоглощающего эффекта можно использовать однотонные занавески из плотной ткани, повешенные на расстоянии 15-20 см отображения [12].

$$L_{\sum=} = 10\lg \sum_{i=1}^{n} 0.1L_j \,, \qquad (5.7)$$

где Li – уровень звукового давления i-го источника шума;

n – количество источников шума.

Полученные результаты расчета сравнивается с допустимым значением уровня шума для данного рабочего места. Если результаты расчета выше допустимого значения уровня шума, то необходимы специальные меры по снижению шума. К ним относятся: облицовка стен и потолка зала

звукопоглощающими материалами, снижение шума в источнике, правильная планировка оборудования и рациональная организация рабочего места оператора.

Уровни звукового давления источников шума, действующих на оператора на его рабочем месте:

Таблица 5.2 – Уровни шума

Наименование	Шум, дБ
жесткий диск	40дБ
монитор	45 дБ
вентилятор	17 дБ
клавиатура	45 дБ
принтер	10 дБ
материнская плата	40дБ

Подставив значения уровня звукового давления для каждого вида оборудования в формулу, получим следующее значение:

$$L\sum=10\cdot\lg(10^{4}+10^{4,5}+10^{1,7}+10^{1}+10^{4,5}+10^{4,2})=49{,}42 \text{ дБ}.$$

Полученное значение не превышает допустимый уровень шума для рабочего места оператора, равный 65 дБ. Если учесть, что вряд ли такие периферийные устройства как сканер и принтер будут использоваться одновременно, то эта цифра будет еще ниже. Кроме того при работе принтера непосредственное присутствие оператора необязательно, т.к. принтер снабжен механизмом автоподачи листов.

Заключение

В данном дипломном проекте рассматривались уязвимости и закладки программного обеспечения, методы их нахождения, а также инструментальные средства, которые предназначались для этого. Для выявления уязвимостей и закладок ПО было выбрано инструментальное средство – PVS – Studio.

PVS-Studio - это инструмент для выявления ошибок и потенциальных уязвимостей в исходном коде программ, написанных на языках C, 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.

В проекте полностью описан интерфейс данного плагина, функциональные возможности, а также ход выполнения статического анализа. Были рассмотрены исходные коды программ, написанные на языках программирования C++, C# и Java. Результаты показали, что наибольшее количество диагностических сообщений было выведено из программного кода, написанного на C++, данное количество составляет 556. На C# - 172, Java – 77.

При выводе диагностических сообщений данное инструментальное средство предусматривает автоматическую помощь пользователям, то есть имеется возможность перехода на официальный сайт и просмотра решений выведенных ошибок.

Проведен сравнительный анализ инструментальных средств PVS – Studio и встроенного анализатора Visual Studio. Результаты сравнения показали, что по общему количеству выявленных предупреждений PVS – Studio уступает Visual Studio, однако по выявленным уязвимостям PVS – Studio показал наилучшие результаты с разницей в 14 уязвимостей.

С целью уменьшения количества уязвимостей разработчикам ПО рекомендуется внедрять в процессы жизненного цикла основные активности, направленные на разработку безопасного ПО (ГОСТ Р 56939) – анализ архитектуры ПО с точки зрения реализации угроз безопасности информации, статический анализ исходных текстов, тестирование безопасности. Внедрение подобных процедур в практику отечественных разработчиков ПО, на наш взгляд, повысит уровень защищенности создаваемого ПО и, как следствие, значительно уменьшить число инцидентов информационной безопасности.

Список литературы:

1 Аветисян А.И. Современные методы статического и динамического анализа программ для решения приоритетных проблем программной инженерии: автореф. дис. … д-ра физ.-мат. наук: 05.13.11 / Аветисян Арутюн Ишханович. М., 2011. 36 с.

2 Чукляев И.И., Морозов А.В., Болотин И.Б. Теоретические основы построения адаптивных систем комплексной защиты информационных ресурсов распределенных информационно-вычислительных систем: монография. Смоленск: ВА ВПВО ВС РФ, 2011. 227 с.

3 Уязвимости программ // Security Vision. Увидеть безопасность. URL: https://www.anti-malware.ru/threats/programs-vulnerability (дата обращения: 20.04.2020).

4 Обнаружение уязвимостей кода в теории и на практике // Сообщество IT специалистов. URL: https://habr.com/ru/company/solarsecurity/blog/420337/ (дата обращения 20.04.2020).

5 Поиск уязвимостей в программах с помощью анализаторов кода // CodeNet. URL: http://www.codenet.ru/progr/other/code-analysers.php (дата обращения: 20.04.2020).

6 Выявление уязвимостей в исходных текстах программного кода // Information Security. Информационная безопасность. URL: https://rtmtech.ru/services/audit-programmnogo-koda/ (дата обращения: 20.04.2020).

7 Аудит программного кода // InfoSec. URL: http://itsec.ru/articles2/Oborandteh/vyyavlenie-uyazvimostey-v-ishodnyh-tekstah-programmnogo-koda/ (дата обращения: 22.04.2020).

8 Ляшко В.Г. «Безопасность жинедеятельности», г. Тула,издательство Тульского государственного университета, 2015 – 236 с.

9 ГОСТ 12.1.003-74-80 «Система стандартов безопасности труда (ССБТ). Электробезопасность», ИПК Издательство стандартов, 2011 – 10 с.

10 БЖД как наука. Определение, цели, структура и задачи // works.doklad.ru. URL: https://works.doklad.ru/view/Z8eQ0sScSu4.html (дата обращения: 22.04.2020).

11 Абдимуратов Ж.С., Мананбаева С.Е. Безопасность жизнедеятельности. Методические указания к выполнению раздела «Расчет производственного освещения» Алматы: АИЭС, 2009. —20с.

12 СНиП 2.04.09-84 Пожарная автоматика зданий и сооружений (с Изменением N 1) // docs.cntd.ru. URL: http://docs.cntd.ru/document/871001018 (дата обращения: 22.04.2020).

Приложение A. Диагностические сообщения C++

V501. There are the same subexpressions on the left and right of the operator "foo".

F502. Maybe"?: '- The operator works differently than expected. ?: "The operator has a lower priority than the operator "Foo".

V503. This is an unfavorable comparison: the indicator < 0.

F504. It is very unlikely that the comma & # 8217;;, missing after keyword & # 8216; return & # 8217;.

Volume. The" allocate " function is used inside the loop. It can quickly flow through the batteries.

F506. The pointer to the local variable ' X ' is stored outside the scope of this variable. This indicator is incorrect.

V507. The pointer to the local field ' X ' is stored outside the scope of this field. This indicator is incorrect.

V508. A sample of "new types"was identified. This probably meant " a new type [n]."

F509. The exception that was requested should be tested within the noexlcept function..Take the block.

V510. The "Foo" function is not expected to accept the class variable as " n " in the actual argument.

V511. The operator size () returns the size indicator, not the field, for that expression.

V512. The function of calling " Foo " causes a buffer overflow or overflow.

V513. Use the _beginfunctions readex / _endarreadex instead of the functions of CreateThread / ExitThread.

V514. Dividing the size of the indicator by a different value. There is a probability of a logical error.

V515. The operator "delete" is applied to the pointer without an indicator.

V516. You test a strange expression. The indicator of the non-zero function is comparable to zero.

V517. Use " if (a) {... Otherwise, (a) {... Sample revealed. There is a probability of a logical error.

F518. The "malloc" function allocates an odd amount of memory calculated by"strlen (expr)". Perhaps the correct option strlen (expr) + 1.

F519.Values are assigned to the variable " x " twice in a row. Maybe it's a mistake.

V520. Comma operator", " field Index expression.

V521. Such expressions with the operator", " are dangerous. Make sure that the expression is correct.

V522. Zero dereference indicator can be done.

F523. The statement "then"is equivalent to the statement"other".

V524. Strangely enough, the function of" Foo_1 "of the body is fully

equivalent to the function of "Foo_2" of the body.

V525. Code that contains a collection of similar blocks. Check items X, Y, Z... Rows N1, n2, n3...

F526. The "strcmp" function returns 0 if the corresponding strings are the same. Check the error status.

V527. Strangely enough, the value of" Zero " is given to the indicator. Probably means: * ptr = zero.

V528. Strangely enough, the indicator is compared with the value of "NULL". Probably means: * ptr != Zero.

V529. Odd comma"; "after operator" if / for / while".

V530. The return value of the "Foo" function is used.

V531. It is strange that the size of the operator() is multiplied by the size ().

V532. Check the application '* pointer++ ' for the sample. Probably meant:"(*pointer)".

V533. Invalid variable " for " operator is likely to increase. Consider a check & # 8216; X & # 8217;.

V534. It is likely that the wrong variable is within the framework of the "to" operator comparison. Consider a check & # 8216; X & # 8217;.

V535. The variable " X " is used for this loop and for the outer loop.

V536. Note that the constant value is represented in the Octal form.

V537. Check the accuracy of using the" X " point.

V538. The line contains the 0x0B check character (vertical rating).

V539. Check the iterators that are passed as arguments to the "Foo" function.

V540. The X member should point to the text to be played with two 0 characters.

V541. It is a complex thread for you.

V542. Look at the odd type of cast: "Type1"on"Type2".

V543.it strangely enough, the value "X" is assigned to the variable " y " of the HRESULT type.

V544.it strangely enough, the value of "X" of the HRESULT type is compared with"Y".

V545.Such a conditional expression "If"is invalid for the value of the HRESULT type"foo". Instead, the macro should be used successfully or unsuccessfully.

V546. The class member is initialized by himself:"foo (foo)".

V547. The expression is always true / false.

F548. You see a casting type rating. Type X [] [] is not equivalent to type * * X.

V549. The "first" argument of the function " Foo "is equivalent to the argument" second".

V550. Special precision comparisons. Probably it is best to use a

comparison with defined precision: fabs (a - b) < Epsilon or fabs (A-B) > Epsilon.

F551. The code for this label "case" is not available.

F552. The variable of the Bool type increases. Perhaps another variable should be increased instead.

V553. Body or class declaration length, function exceeding 2000 lines. You should consider a new code.

F554. Incorrect use of the smart indicator.

Volume. The expression Type "AB > 0" will work as " a != B'.

F556. The values of different types of enums are compared.

F557. Possible overload of the field.

F558. The function returns an indicator / link to a local temporary object.

F559. Suspicious assignment under the conditional expression " if / While / statement.

V560. Part of the conditional expression is always true / false.

V561.it perhaps it is better to assign the value of the variable " foo " than to claim it again.

V562. It is strange to compare the value of the bool type with the value of N.

V563. It is possible that this "other" branch of the previous one", if the application should be applied.

V564. The operator " E " or " / " is used for the value of the Bool type. Probably forgot to insert brackets or use operators ' & & & & 'or/////'.

V565. Empty exception handler. Silent suppression exceptions can hide the presence of errors in the source code during testing.

V566. The number constant is converted to an indicator. Maybe an error or a bad coding style.

V567.The variable modification is not comparable to another operation of the same variable. This can lead to insecure behavior.

V568. It is strange that the dimension of the operator's argument () is an expression.

V569. Constant cost reduction.

V570. The variable is assigned itself.

V571. I repeat, I'll check. This condition has already been confirmed in the previous line.

V572. Strangely enough, the object created by the "new" operator is immediately thrown into another type.

F573. The uninitialized variable "foo"was used. The variable was used for initialization.

V574. The indicator is used simultaneously as an array and as an indicator for one object.

V575. The function makes a special argument.

V576. Incorrect format. Check the N actual function of the argument

"Foo".

V577. Label present switch (). It is possible that these are typographical errors, and instead the operator "default:"should be used.

F578. A strange little intelligent operation was revealed. Look at that.

V579. The function " foo " takes the indicator and its size as arguments. It could be a mistake. Check the argument N.

V580. Strange explicit type cast. Look at that.

V581. Conditional expressions " IF " statements that are next to each other are identical.

V582. Check the source code running in the container.

V583. ?: "Operator, regardless of its conditional expression, always returns the same value.

V584. The same value is on both sides of the operator. The expression is incorrect or can be simplified.

V585. In an effort to free up the memory in which the local variable "foo"is stored.

V586. The function " Foo " is called twice deallocation of the same resource.

V587. Odd sequence of tasks of this type: A = B; B = a;.

V588. The expression Type "A = + B"is used. Consider checking how it is possible that it is meant "a + = B".

V589. The expression Type "A = B"is used. Consider checking how it is possible that it is meant 'a - = B'.

590. Look at this expression. The expression is exaggerated or contains a false expression.

V591. The empty function must return the value.

V592. The expression was twice surrounded by brackets: ((expression). A pair of brackets is useless or poorly printed.

V593. Tick the expression"A = B = C". The pronunciation is calculated as follows: A = (B = C)'.

V594. The indicator comes out inside the Matrix.

F595. The indicator was used until it was certified against nullptr. Control lines: N1, N2.

V596. The object was created, but not used. The keyword "shoot" may be absent.

V597. The compiler can remove the "memset", which is used to rinse the"foo" cache. RtlSecureZeroMemory () is a function to be used to delete personal data.

F598. The "memset / memcpy" function is used to copy fields from the " Foo " class to cancel/. This worsens the virtual table indicator.

F599. Virtual destructor is not present, although the Foo class includes virtual functions.

V600. Check his condition. The " Foo " indicator is still not equal to zero.

Продолжение приложения А

V601. A curious silent casting of type.

V602. A verification of this expression. "<"should be replaced by"<<".

V603. This object is done, but does not work. If you want to call manufacturer, ' it - > Foo:: Foo(....) must be used.

V604. It is curious that the number of repetitions in the loop, the size of the index finger.

V605. Check the print. An unsigned value is compared with the NN number.

V606. Owner is a token".

V607. Masterful Of The Expression "Foo".

V608. Repeated sequence of explicit type conversions.

V609. Split or mod from zero.

V610. Unspecified Behavior. Control the cursor.

V611. The methods of memory allocation and deallocation are incompatible.

V612. An unconditional "pause/ continue / Return / go" within a loop.

B613. Strange numerical index with 'malloc / new'.

V614. The non-initialized variable 'Foo'is used.

V615. A strange explicit conversion of the 'float *' type to the ' double*'type.

V616. The constant called ' Foo ' with the value 0 is used in the bitwise mode.

V617. Check his condition. The bitwise function argument ' / ' always contains a non-zero value.

V618. it is dangerous to call the function " Foo " in this way, because a given line can contain a format specification. An example of secure code: printf ("%s", str);

V619. A table is used as an indicator on a single object.

V620. It is unusual in that the type of expression sizeof (T)*is summed with the index to the type T

B621. An operator control"is". It is possible that the loop was executed incorrectly or not executed at all.

V622. Check the" switch " statement. It is possible that the first operator "boxes"is missing.

V623. Check the box";: 'player. A temporary object is created, and then destroyed.

V624. The NN constant must be used. The resulting values may be inaccurate. Using the fixed m_nn of < math.home>.

V625. An operator control"is". The initial and final values of the iterator are the same.

B626. Check for printing errors. Is it possible that","? " should be replaced.

V627. Check the expression. The sizeof argument () is a macro spanning a number.

V628. It is possible that someone is commented, which is the result of the operation of the logic of the program is modified.

Продолжение приложения А

V629. Check the expression. A piece offset at a value of 32 bits with a subsequent expansion to 64 bits is a formula.

V630. The "malloc" function is used to allocate memory for a number of objects, which are classes containing constructors / destroyers.

V631. To check the function call 'Foo'. Setting an absolute path to the file or directory is considered a poor style.

V632. Check the argument of the function 'Foo'. It is curious that the argument of the type "T".

V633. Check the expression. Maybe you!= 'should be used here.

V634. The priority of the function is higher than that of the"<<"function. It is possible that parentheses are used in an expression.

V635. Check the expression. The length should probably be multiplied by sizeof (wchar_t).

V636. this expression was implicitly involved in the entire real type. Explicit casting methods are used to prevent overflow or loss of cracks.

V637. The second condition is always wrong.

V638. Zero terminal string. The signs "\ 0n"were found. Well said: "xNN."

V639. check the call detection function. It is possible that one of & # 8217;) brackets were set incorrectly.

V640.the logical code of the function does not correspond to formatting.

V641. the size of a cache is not the size of many items.

V642.saving the resulting variable of the function "byte" does not matter. Important details can disappear and break the logic program.

V643. an extraordinary mathematical indicator. A value of the type "char"is added to the chain pointer.

V644.suspicion of this function. It is possible that a T-type object will be created.

Calling a function can cause the buffer to overflow. The boundaries must contain not the size of the cache, but several characters.

V646 helicopter. check the application logic. It is quite possible that the word "other"is missing.

V647.the value of Type A is assigned to the index of Type "B".

V648. the "& & & "mission priority is greater than the"/"operation.

V649. there are two "if" statements in the same categories. The first "if" app includes a recovery feature. Another "if" statement makes no sense.

V650. the type of cast is used twice in a row. The"+"function is then performed. This probably means: (T1) (T2) and + (b).

F651. the random function " Sizof (X) / sizof (T) "is executed if "X"is a"class"type.

F652. the procedure shall be performed at least three times in a row.

V653. suspect string is used in two parts. There may be no comma.

F654. loop mode is always correct / wrong.

F655. the ropes were tied with a chain, but were not used. Look at the handle.

F656. variables initiate the call of the same function. This is probably an error or non-optimized code.

F657. the odd thing is that this property always returns the same NN value.

F658. this value is subtracted from the unsigned variable. This could lead to flooding. In such cases, the comparison process can be unexpected.

F659. function declaration names "Foo" differ only from" const", but function bodies have a different composition. This is questionable and could be a mistake.

Variable chamber V660 with round print. The Program contains an unused label and a function call: "CC ()". It may have been intended for: CC ().

F661. the dubious expression " A [B < C]". It probably means "[b] < c".

F662. check the loop expression. Many reservoirs are used to determine the values of the initial and final iterators.

V663. An infinite loop is possible. "Crime.EOF () is not enough to pull the loop. Consider adding cinemas.Can () ' Calls the conditional expression of the function.

F664.the Pointer is indicated in the initialization availability list until the compilation function of Appendix zero is certified.

F665. "#Pragma warning (default: X) " may be incorrect. Instead, use " # Pragma warning (push / pop)."

Page f666. check the function of the parameter NN "Foo". It is possible that this value does not correspond to the length of the string passed to the argument YY.

F667. The "throw" Operator has no basis and is not in the "catch"block.

F668. the zero pointer should not be tested as a" new " user memory. Exceptions are made if a memory error occurs.

F669. that argument is fickle. The analyzer cannot determine where this argument has changed. This feature may contain an error.

V670.an an unintended class member is used to initiate the second element. Note that the members of the group are in the introductory class in their statements.

V671.the function can be replaced by changing the variable.

V672. there is probably no need to create a new variable. One parameter function has the same name, and this parameter is a reference.

V673.storing values requires more than n Bits, but the expression is evaluated as a T-type that can only contain a k-bit.

V674. pronunciation includes a fuzzy mix of Integer and real type.

V675. safe memory.

V676. the punch variable type cannot be compared with the real value.

V677. The customs declaration is standard. Use notification header files instead.

V678. the object is used as an argument in its own method. Check the first real function of the "Foo"argument.

V679. the variable " X " has not been initiated. This variable has been changed to the reference function "foo", where its value is used.

V680. the expression "remove a, b"is destroyed only by the object"a". Then the operator returns the resulting value to the right of the expression.

V681. the language standard does not define any command that works when evaluating arguments, called "Foo".

V682. no questionable word: '/ r'. It is quite possible that reversing is used instead of "\ r".

V683. check the loop expression. Instead of the variable "n", the variable" i"can be added.

V684. the value of the variable was unchanged. Look at the handle. It is quite possible that instead of "0"it will be represented by" 1".

V685. check the message. This expression contains a comma.

V686. preview: J /(L&...). This expression is exaggerated or contains a logical error.

V687. the field size is calculated according to the sizefrom () function that the operator has added to the pointer. It is possible that the number of elements is calculated according to size a / size (a[0]).

V688. the local variable " Foo " can cause confusion.

V689. The Fu-class destroyer was not classified as a virtual destroyer. It is quite possible that the smart indicator does not destroy the target properly.

V690. The class runs copier / operator=, but this operator = / copier builder does not.

V691. empirical analysis. There may be a typo in the literal chain. The word " Fu " is questionable.

V692. in the wrong company, no characters are added to the string of characters. The string length with the strlen function is correctly determined by the string with the zero Terminator.

V693. check the conditional expression of the loop. It is possible that & # 8216; and < X size () should be used instead of X size () & # 8217;.

V694. A State (PTR-const_value) has a false value if and only if the pointer value corresponds to the magic Constante.

V695. intersections in the area are conditional.

V696. sequels to " irony.".. When (wrong) loop, because space is always wrong .

V697.the number of factors given in the field is the same as the size of the byte index.

V698. strcmp () functions c