**Белорусский государственный технологический университет**

**Факультет информационных технологий**

**Кафедра программной инженерии**

**Реферат**

По дисциплине «Основы алгоритмизации и программирования»

На тему «Основные уязвимости языка C++: причины, примеры и методы предотвращения»

Выполнил:

Студент 1 курса 10 группы ПИ

Макаревич Кирилл Витальевич

Проверил: Белодед Николай Иванович

2024, Минск

Оглавление

[**Введение:** 3](#_Toc184074944)

[**Классификация уязвимостей в C++:** 3](#_Toc184074945)

[**Примеры кода для основных уязвимостей:** 4](#_Toc184074946)

[**Обзор алгоритмов кода:** 5](#_Toc184074947)

[**Анализ последствий:** 9](#_Toc184074948)

[**Методы предотвращения уязвимостей:** 11](#_Toc184074949)

[**Заключение:** 12](#_Toc184074950)

# **Введение:**

C++ — это один из самых мощных и широко используемых языков программирования, который часто используется для разработки программного обеспечения, требующего высокой производительности, таких как операционные системы, игры, драйвера и другие ресурсоёмкие приложения. Однако, несмотря на свою мощь и гибкость, C++ является языком с множеством потенциальных уязвимостей, которые могут быть использованы злоумышленниками для атак на системы. Это обусловлено низкоуровневым управлением памятью, а также возможностью манипуляций с указателями и ресурсами, что может привести к непредсказуемому поведению программы.

Безопасность в программировании на C++ является важнейшим аспектом, поскольку даже небольшие ошибки, такие как неправильное использование памяти или недостаточная проверка данных, могут привести к серьёзным уязвимостям. Такие ошибки, как переполнение буфера, утечка памяти или использование неинициализированных переменных, становятся потенциальными векторами атак, открывая двери для выполнения произвольного кода, повреждения данных или даже полного контроля над системой.

В данном реферате будет рассмотрена классификация уязвимостей C++, их примеры, последствия и эффективные методы предотвращения подобных проблем.

# **Классификация уязвимостей в C++:**

Уязвимости в C++ могут быть классифицированы по типу ошибки или дефекта в коде, а также по последствиям, которые они могут вызвать. Ниже приведены основные виды уязвимостей.

1. **Переполнение буфера (**[**CWE-121**](https://cwe.mitre.org/data/definitions/121.html)**) и запись за пределы буфера (**[**CWE-787**](https://cwe.mitre.org/data/definitions/787.html)**)**

Корпорация [MITRE](https://ru.wikipedia.org/wiki/Mitre) сообщает, что **переполнение буфера** составляет более 10 000 уязвимостей в программном обеспечении, из которых 23% являются критическими. Эта уязвимость возникает, когда программа записывает данные, превышающие выделенный размер буфера. Иными словами, если нет механизма, ограничивающего длину данных, происходит запись за пределы выделенной памяти, что может привести к перезаписи соседних участков памяти.

Для предотвращения переполнения буфера используется механизм, который ограничивает объем данных, записываемых в буфер, например, фиксированный размер блока или определенный участок памяти. Когда программа позволяет запись за пределы буфера, это может привести к повреждению данных или сбою.

**Переполнение буфера** считается одной из самых опасных уязвимостей в области работы с памятью. Несмотря на то, что использование этой уязвимости может быть сложным, сама проблема заключается в том, что буфер переполняется. Это явление часто связано с **переполнением буфера на стеке** или записью данных за пределы допустимого диапазона.

Эксплуатация таких уязвимостей может быть непростой задачей, так как для успешной атаки недостаточно просто перезаписать стек. В результате были разработаны различные **защитные механизмы**, такие как **ASLR** (рандомизация адресного пространства), **стековые канарейки** и **DEP** (предотвращение выполнения данных), которые предотвращают возможное повреждение памяти. В случае сбоя одного из этих механизмов ОС прерывает выполнение программы, выдав ошибку **SEGFAULT**, что усложняет эксплуатацию уязвимости.

2. **Использование после освобождения (**[**CWE-416**](https://cwe.mitre.org/data/definitions/416.html)**)**

Это уязвимость, которая возникает, когда программа пытается использовать память, которая была ранее освобождена. Когда переменная или объект освобождается из памяти, но затем используется снова, это может привести к непредсказуемым результатам и ошибкам, таким как сбои приложения или эксплуатация уязвимости злоумышленниками. Это связано с неправильным управлением памятью и может быть вызвано ошибками в коде, где ссылки на освобождённые объекты остаются активными и пытаются получить доступ к памяти, которая уже не принадлежит приложению.

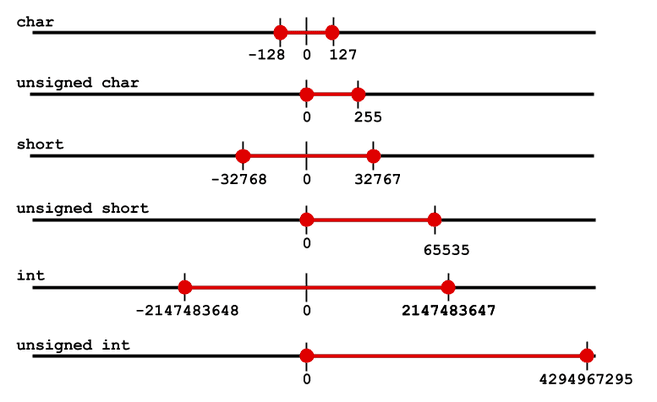
Уязвимость Use-After-Free (UAF) может быть использована злоумышленниками для перезаписи памяти, что открывает двери для дальнейших атак, таких как выполнение произвольного кода или изменение данных программы. Атака, основанная на использовании освобождённой памяти, часто использует уже устаревшие или повреждённые данные, что затрудняет обнаружение проблемы на этапе тестирования.

3. **Целочисленное переполнение и потеря значимости (**[**CWE-190**](https://cwe.mitre.org/data/definitions/190.html) **и** [**CWE-191**](https://cwe.mitre.org/data/definitions/191.html)**)**

Это две распространённые уязвимости, связанные с неправильным представлением чисел в компьютерах, которые могут привести к неожиданным результатам, таким как переполнение буфера или неправомерное изменение значений.

Целочисленное переполнение происходит, когда значение, которое программа пытается сохранить, превышает максимально возможное для данного типа данных. Например, если переменная типа int в C++ может хранить значения от -2,147,483,648 до 2,147,483,647, и программе передаётся значение, которое больше этого диапазона, то результатом будет переполнение: значение начнёт «переворачиваться» и вернётся к нулю или минимальному значению.

Целочисленная потеря значимости, с другой стороны, возникает, когда значимость значения теряется из-за попытки сохранить число в переменной, которая не может его адекватно представить. Например, попытка сохранить отрицательное число в беззнаковом целочисленном типе данных может привести к его некорректному отображению, так как беззнаковые типы данных не поддерживают отрицательные значения. На этой диаграмме показаны типы переменных и значения, которые они могут содержать:



4. **Разыменование нулевого указателя (**[**CWE-467**](https://cwe.mitre.org/data/definitions/476.html)**)**

Это уязвимость, которая возникает, когда программа пытается получить доступ к памяти по адресу, указанному нулевым указателем (то есть указателем, который не ссылается ни на одну область памяти). Это приводит к критической ошибке, известной как нарушение доступа к памяти, и вызывает завершение работы программы. Разыменование — это процесс обращения к данным, находящимся по адресу, на который указывает указатель. Если указатель равен NULL (или nullptr в C++), он не ссылается ни на какой объект в памяти. Попытка разыменовать такой указатель приводит к неопределённому поведению, которое может проявляться по-разному, например:

Ошибкой выполнения с завершением программы (segmentation fault или access violation).

В редких случаях — отсутствием ошибки, что делает уязвимость трудноуловимой.

5. **Чтение за пределами допустимого диапазона (**[**CWE-125**](https://cwe.mitre.org/data/definitions/125.html)**)**

Чтение за пределами допустимого диапазона возникает, когда программа обращается к памяти за пределами указанного буфера или области данных. Это приводит к уязвимостям, которые могут проявляться в виде ошибок, таких как:

**Сбой системы** (например, аварийное завершение программы).

**Утечка конфиденциальной информации**, включая пароли, данные пользователей или ключи шифрования.

Эта уязвимость опасна, поскольку злоумышленники могут использовать её для сбора данных, к которым у них не должно быть доступа, что может поставить под угрозу безопасность всего приложения.

# **Примеры кода для основных уязвимостей:**

**1. Переполнение буфера (CWE-121 и CWE-787)**

#include <iostream>

#include <cstring>

void vulnerableBufferOverflow() {

char buffer[8];

std::cout << "Enter some text: ";

std::cin >> buffer; // Уязвимость: нет проверки длины ввода

std::cout << "You entered: " << buffer << std::endl;

}

**Объяснение:**  
Если пользователь введет строку длиной больше 8 символов, это приведет к перезаписи данных за пределами буфера.

**2. Использование после освобождения (CWE-416)**

#include <iostream>

void useAfterFree() {

int\* ptr = new int(42); // Выделение памяти

delete ptr; // Освобождение памяти

std::cout << \*ptr << std::endl; // Использование после освобождения

}

**Объяснение:**  
После освобождения памяти указатель ptr становится "висячим". Попытка разыменовать его приводит к неопределенному поведению.

**3. Целочисленное переполнение (CWE-190)**

#include <iostream>

#include <limits>

void integerOverflow() {

int max = std::numeric\_limits<int>::max(); // Максимальное значение int

int result = max + 1; // Переполнение

std::cout << "Result: " << result << std::endl; // Результат: отрицательное число

}

**Объяснение:**  
Переполнение происходит, когда результат вычислений превышает диапазон допустимых значений типа данных.

**4. Разыменование нулевого указателя (CWE-467)**

#include <iostream>

void nullPointerDereference() {

int\* ptr = nullptr; // Нулевой указатель

\*ptr = 42; // Разыменование нулевого указателя

}

**Объяснение:**  
Попытка разыменовать указатель, который не ссылается на допустимую область памяти, вызывает сбой программы.

**5. Чтение за пределами допустимого диапазона (CWE-125)**

#include <iostream>

void readOutOfBounds() {

int arr[3] = { 1, 2, 3 };

std::cout << arr[5] << std::endl; // Чтение за пределами массива

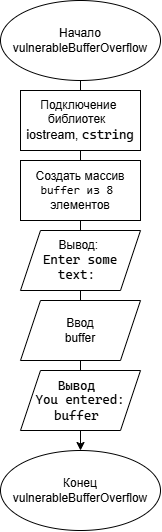
}

**Объяснение:**  
Попытка доступа к элементу за пределами массива вызывает неопределенное поведение, включая потенциальный доступ к чужим данным.

# **Обзор алгоритмов кода:**

**1. Переполнение буфера (CWE-121 и CWE-787)**

Блок-схема



**Словесно-формульное описание алгоритма**

1. Создается массив символов buffer длиной 8 для хранения строки, введенной пользователем.
2. Используется стандартный ввод для получения строки, которую пользователь вводит с клавиатуры. Однако метод не проверяет, чтобы ввод помещался в buffer, что может привести к переполнению буфера, если пользователь введет более 7 символов (8-й символ зарезервирован для завершающего \0).
3. Содержимое buffer выводится на экран.
4. Если введенная строка превышает длину буфера, происходит запись за пределы массива.

**Псевдокод**

НАЧАЛО

СОЗДАТЬ массив символов buffer длиной 8

ВЫВЕСТИ "Enter some text: "

ВВЕСТИ строку в buffer (ввод пользователя через std::cin)

ЕСЛИ длина введенной строки > 7

ТО ВОЗМОЖНО переполнение буфера

ВЫВЕСТИ "You entered: " + содержимое buffer

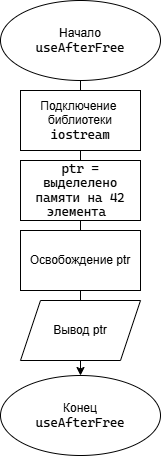
КОНЕЦ

**2. Использование после освобождения (CWE-416)**

**Словесно-формульное описание алгоритма**

1. Выделяется память для хранения целого числа 42 и иницилизируется указатель ptr.
2. Освобождение памяти, на которую указывает ptr. После этого указатель ptr становится висячим (dangling pointer).
3. Выполняется попытка разыменовать указатель ptr, чтобы вывести значение, хранящееся по адресу, который уже был освобожден. Такое действие приводит к неопределенному поведению программы.

**Блок-схема**



**Псевдокод**

НАЧАЛО

ВЫДЕЛИТЬ память для переменной типа int с помощью оператора new

ПРИСВОИТЬ значение 42 в выделенную память

ОСВОБОДИТЬ выделенную память с помощью оператора delete

ВЫВЕСТИ значение из освобожденной памяти через указатель ptr

КОНЕЦ

**3. Целочисленное переполнение (CWE-190)**

**Словесно-формульное описание алгоритма**

1. Вычисляем максимально возможное значение для целых чисел типа int. Это значение сохраняется в переменной max.
2. К значению max прибавляется 1, и результат сохраняется в переменной result. Поскольку переменная result типа int не может хранить значения больше max, происходит переполнение, и результат становится отрицательным из-за циклической природы хранения данных в типе int.
3. Вывод результата: Значение переменной result, ставшее отрицательным из-за переполнения, выводится в консоль.

**Псевдокод**

НАЧАЛО

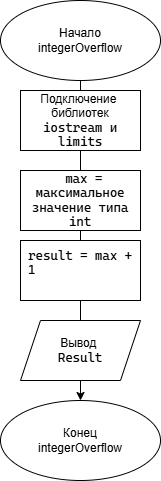
УСТАНОВИТЬ max = максимальное значение типа int (2^31 - 1 для 32-битной системы)

ПРИСВОИТЬ result = max + 1

ВЫВЕСТИ "Result: ", result

КОНЕЦ

**Блок-схема**

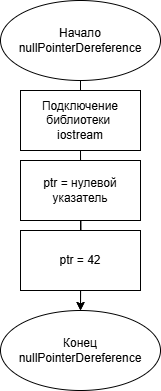


**4. Разыменование нулевого указателя (CWE-467)**

**Словесно-формульное описание**

1. Указатель ptr инициализируется значением nullptr, что означает, что он не указывает на какую-либо область памяти.
2. Выполняется операция разыменования ptr, чтобы записать значение 42. Однако указатель ptr не указывает на допустимую область памяти, что вызывает неопределённое поведение, обычно приводя к сбою программы.

**Блок-схема**



**Псевдокод**

НАЧАЛО

УСТАНОВИТЬ ptr = nullptr // Указатель не инициализирован на область памяти

РАЗЫМЕНОВАТЬ ptr И ПРИСВОИТЬ 42 // Ошибка: неопределённое поведение

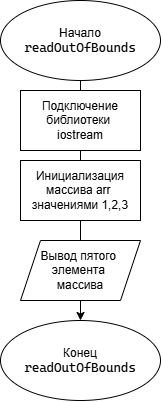
КОНЕЦ

**5. Чтение за пределами допустимого диапазона (CWE-125)**

**Словесно-формульное описание алгоритм**

1. Создаётся массив целых чисел arr размером 3 с элементами:
2. Осуществляется обращение к элементу массива с индексом 5, который находится за пределами допустимого диапазона индексов (от 0 до 2). Это вызывает неопределённое поведение, так как запрашиваемая область памяти не принадлежит массиву.

**Блок-схема**



**Псевдокод**

НАЧАЛО

СОЗДАТЬ массив arr из 3 элементов: {1, 2, 3}

ВЫВЕСТИ значение arr[5] на экран // Ошибка: выход за пределы массива

КОНЕЦ

# **Анализ последствий:**

Уязвимости программного обеспечения могут привести к целому ряду негативных последствий: от сбоев и повреждения данных до серьёзных угроз для безопасности информации и системы. Рассмотрим последствия каждой уязвимости в структурированном виде.

**1. Переполнение буфера (CWE-121 и CWE-787)**

**Функциональные последствия:**

* Сбой программы: Нарушение структуры данных из-за записи за пределы буфера.
* Повреждение данных: Изменение или уничтожение критически важных данных.

**Безопасностные последствия:**

* Исполнение произвольного кода: Злоумышленники могут модифицировать адрес возврата для запуска вредоносного кода.
* Утечка данных: Чтение за пределами буфера раскрывает содержимое памяти, включая пароли или ключи.

**Пример:**

* Переполнение буфера на стеке может позволить злоумышленникам записать вредоносный код в память программы.

**Долгосрочные последствия:**

* Высокие затраты на исправление: Обнаружение и устранение уязвимости требует значительных ресурсов.
* Потеря доверия: Раскрытие данных может подорвать репутацию компании.

**2. Использование после освобождения (CWE-416)**

**Функциональные последствия:**

* Сбой приложения: Доступ к освобождённой памяти приводит к аварийному завершению.
* Повреждение данных**:** Модификация ранее освобождённой памяти другим процессом.

**Безопасностные последствия:**

* Исполнение произвольного кода: Вредоносный код может быть помещён в освобождённую память.
* Эскалация привилегий: Злоумышленники могут получить доступ к защищённым ресурсам.

**Пример:**

* Уязвимость **CVE-2014-0160 (Heartbleed)** позволяла извлекать данные из освобождённых областей памяти.

**Долгосрочные последствия:**

* Юридическая ответственность: За утечку данных может быть наложен штраф.
* Масштабные риски конфиденциальности: Утечка личной информации пользователей.

**3. Целочисленное переполнение (CWE-190) и потеря значимости (CWE-191)**

**Функциональные последствия:**

* Ошибки логики: Нарушение расчётов из-за переполнения переменных.
* Сбой приложения: Непредсказуемое поведение системы.

**Безопасностные последствия:**

* Переполнение буфера: Может вызвать дополнительные уязвимости.
* Неавторизованный доступ: Манипуляция данными для обхода проверок.

**Пример:**

* В уязвимости **CVE-2002-0639 (OpenSSH)** переполнение приводило к выполнению вредоносного кода.

**Долгосрочные последствия:**

* Дизайнерские проблемы: Неоптимальные решения усложняют модернизацию.
* Финансовые потери: Устранение таких ошибок требует значительных усилий.

**4. Разыменование нулевого указателя (CWE-467)**

**Функциональные последствия:**

* Сбой приложения: Попытка разыменовать нулевой указатель вызывает крах.
* Непредсказуемое поведение: Отсутствие обработчиков ошибок ведёт к нестабильной работе.

**Безопасностные последствия:**

* Выполнение кода: При определённых обстоятельствах возможно управление содержимым памяти.
* Раскрытие данных: Доступ к критическим данным без авторизации.

**Пример:**

* Некорректная обработка нулевых указателей в системных библиотеках может вызвать серьёзные сбои.

**Долгосрочные последствия:**

* Снижение надёжности ПО: Такие ошибки приводят к потере доверия пользователей.
* Сложность отладки: Обнаружение подобных ошибок требует значительных усилий.

**5. Чтение за пределами допустимого диапазона (CWE-125)**

**Функциональные последствия:**

* Сбой программы: Попытка чтения из недопустимой памяти вызывает крах системы.
* Повреждение данных: Нарушение логики работы программы.

**Безопасностные последствия:**

* Утечка информации: Возможность извлечения данных из недопустимых областей памяти.
* Атаки на целостность данных: Чтение критических данных без разрешения.

**Пример:**

* Уязвимость **CVE-2020-9365 (PureFTPd)** позволяла злоумышленникам читать конфиденциальную информацию за пределами буфера.

**Долгосрочные последствия:**

* Финансовые потери: Утечка данных ведёт к убыткам компании.
* Юридические обязательства: Организации обязаны уведомлять пользователей о нарушениях.

# **Методы предотвращения уязвимостей:**

Обеспечение безопасности программного обеспечения требует применения различных подходов и методов на всех этапах разработки. Рассмотрим методы предотвращения для каждой из упомянутых уязвимостей.

**1. Переполнение буфера (CWE-121 и CWE-787)**

**Программные методы:**

* Использование функций с проверкой границ: Предпочтение функций strncpy, snprintf вместо небезопасных strcpy, sprintf.
* Статический анализ кода: Инструменты типа SonarQube помогают выявить потенциально уязвимые места.
* Защита стека (Stack Protector): Активация защиты от переполнения через компиляторы (например, флаг -fstack-protector в GCC).

**Архитектурные подходы:**

* Разделение данных и инструкций: Предотвращение исполнения данных как кода с помощью DEP (Data Execution Prevention).
* Использование технологий ASLR: Случайная адресация областей памяти уменьшает вероятность успешной атаки.

**2. Использование после освобождения (CWE-416)**

**Программные методы:**

* Явное обнуление указателей: После освобождения памяти устанавливайте указатель в NULL.
* Использование умных указателей: В языках, поддерживающих автоматическое управление памятью (C++, Java), такие структуры помогают избежать ошибок управления ресурсами.
* Инструменты динамического анализа: AddressSanitizer (ASan) позволяет обнаружить использование освобождённой памяти.

**Архитектурные подходы:**

* Изоляция процессов: Злоумышленник не сможет использовать уязвимость в одном процессе для атаки на другой.

**3. Целочисленное переполнение (CWE-190) и потеря значимости (CWE-191)**

**Программные методы:**

* Проверка диапазона значений: Используйте методы валидации данных до и после операций с числами.
* Безопасные типы данных: В C++ применяйте типы int64\_t или uint64\_t для предотвращения переполнения.
* Анализ кода: Используйте статические анализаторы, например Coverity, для выявления ошибок.

Архитектурные подходы:

* Механизмы управления переполнением: Современные процессоры и компиляторы включают флаги, которые могут остановить выполнение программы при переполнении.

**4. Разыменование нулевого указателя (CWE-467)**

**Программные методы:**

* Проверка на NULL: Перед разыменованием указателя всегда проверяйте его значение.
* Использование языков с автоматическим управлением памятью: Java, Python и другие языки снижают риск ошибок.
* Детальное логирование: Логи помогают выявить проблемные участки в случае сбоя.

Архитектурные подходы:

* Тестирование и покрытие кода: Увеличьте покрытие тестами, особенно в критических частях кода.

**5. Чтение за пределами допустимого диапазона (CWE-125)**

**Программные методы:**

* Использование стандартных функций проверки: Например, в C++ применять std::vector::at, который проверяет границы.
* Тестирование граничных значений: Проводите тестирование для всех возможных сценариев ввода данных.
* Инструменты анализа: Valgrind помогает выявлять обращения к неинициализированным или недопустимым областям памяти.

**Архитектурные подходы:**

* Ограничение доступа: Обеспечьте строгую политику доступа к данным на уровне ОС.

**Общие методы предотвращения уязвимостей**

1. **Обучение разработчиков:** Знание основных принципов безопасного кодирования минимизирует ошибки.
2. **Код-ревью:** Регулярная проверка кода другими специалистами позволяет выявить уязвимости на ранних этапах.
3. **Автоматизированное тестирование:** Интеграция тестов в CI/CD позволяет своевременно обнаружить ошибки.
4. **Обновление библиотек:** Используйте актуальные версии сторонних компонентов.
5. **Моделирование угроз:** Выявление потенциальных векторов атак и тестирование на их предотвращение.

# **Заключение:**

Уязвимости в программном обеспечении представляют собой критические проблемы, которые могут привести к серьёзным последствиям для безопасности, стабильности и репутации приложений и компаний. В процессе анализа мы рассмотрели различные виды уязвимостей, такие как переполнение буфера, использование после освобождения памяти, целочисленные ошибки, разыменование нулевого указателя и чтение за пределами допустимого диапазона. Каждая из них обладает уникальными особенностями, последствиями и методами предотвращения.

Основные выводы:

1. **Понимание угроз**: Уязвимости часто становятся следствием недостаточной проверки входных данных, неправильного управления памятью или использования устаревших библиотек.
2. **Превентивные меры**: Применение современных инструментов анализа кода, соблюдение принципов безопасного программирования и использование защитных механизмов, таких как ASLR или DEP, способны значительно снизить вероятность атак.
3. **Роль тестирования**: Интеграция статического и динамического анализа в процесс разработки помогает выявить проблемы на ранних стадиях и снизить расходы на их устранение.
4. **Обучение и практика**: Постоянное повышение квалификации разработчиков в области кибербезопасности остаётся одним из важнейших элементов в борьбе с уязвимостями.

Создание безопасного программного обеспечения требует системного подхода, включающего правильное проектирование, использование проверенных технологий и постоянное тестирование. Разработчикам важно не только исправлять уже существующие уязвимости, но и предотвращать их появление в будущем.

**Призыв к действиям**

Для достижения более высокого уровня безопасности программного обеспечения необходимо:

* Стремиться к созданию культуры безопасности в командах разработки.
* Использовать доступные инструменты и методологии, которые обеспечивают защиту программ на всех уровнях.
* Следовать лучшим практикам в проектировании и кодировании.

Инвестиции в безопасность на этапе разработки всегда оказываются более экономичными, чем последствия устранения уязвимостей после инцидентов.