# Информационная безопасность телекоммуникационных систем

Екатеринбург
2023

Министерство науки и высшего образования Российской Федерации
Уральский государственный экономический университет

# Информационная безопасность телекоммуникационных систем

Рекомендовано
Советом по учебно-методическим вопросам и качеству образования
Уральского государственного экономического университета
в качестве учебного пособия

Екатеринбург
2023

УДК 621.39(075.8)
ББК 32.348(я73)
И74

Рецензенты:

Ученый совет Уральского института фондового рынка
(протокол № 3 от 15 декабря 2022 г.);

директор учебно-научного центра «Информационная безопасность»
Института радиоэлектроники и информационных технологий — РтФ
Уральского федерального университета имени первого Президента России Б. Н. Ельцина,
доктор технических наук, профессор
*С. В. Поршнев*

Авторский коллектив:
*В. П. Часовских*, *Г. А. Акчурина, В. Г. Лабунец*, *Е. Н. Стариков*, *Е. В. Кох*

И74 **Информационная безопасность телекоммуникационных систем** : учебное пособие / В. П. Часовских, Г. А. Акчурина, В. Г. Лабунец [и др.] ; Министерство науки и высшего образования Российской Федерации, Уральский государственный экономический университет. — Екатеринбург : УрГЭУ, 2023. — 143 с.

В учебном пособии рассматриваются защита информации, закономерность и методы ее создания, организация хранения, кодирования и декодирования, поиска, преобразования, передачи средствами телекоммуникационных систем и применение в различных приложениях. Изучаются различные стандарты и передовые методы реагирования, процессы взлома данных и политики безопасности, базовые средства контроля безопасности. Рассматриваются методы шифрования информации с аутентификацией, безопасной случайности, а также функции хеширования.

Для студентов очной и заочной форм обучения направлений подготовки бакалавриата 02.03.03 «Математическое обеспечение и администрирование информационных систем», 09.03.03 «Прикладная информатика», 09.03.01 «Информатика и вычислительная техника», 10.03.01 «Информационная безопасность», а также может быть полезно студегтам магистратуры, обучающимся по направлениям 09.04.03 «Прикладная информатика» и 38.04.05 «Бизнес-информатика».

УДК 621.39(075.8)
ББК 32.348(я73)

© Авторы, указанные на обороте титульного листа, 2023
© Уральский государственный экономический университет, 2023

# Введение

Стратегия развития информационного общества Российской Федерации в 2017–2030 гг. направлена на создание условий для развития общества знаний, улучшение благосостояния и уровня жизни граждан путем повышения доступности и качества товаров и услуг, произведенных в цифровой экономике с использованием современных технологий, повышения степени информированности и цифровой грамотности, улучшения доступности и качества государственных услуг для граждан, а также безопасности как внутри страны, так и за ее пределами. Важное место в цифровой экономике занимают вопросы информационной безопасности телекоммуникационных систем.

В настоящее время большое внимание уделяется защите накапливаемой, хранимой и обрабатываемой в ЭВМ информации и построению на этой основе информационных систем. В последние годы появились информационные системы с технологиями искусственного интеллекта.

В теоретическом плане основное внимание уделяется исследованию уязвимости информации в системах электронной обработки, выявлению и анализу каналов утечки, обоснованию принципов защиты в больших информационных системах и разработке методик оценки надежности защиты. В результате исследований формулируются корректные постановки задач и достаточно строгие и практически значимые их решения. К таким вопросам, например, относятся: выбор оптимальной длины пароля и оптимальной структуры ключа защиты и оценка стойкости шифрования.

Наибольшие результаты достигнуты во втором направлении работ — в разработке конкретных средств, методов и мероприятий, с помощью которых можно обеспечить защиту информации в ЭВМ и информационных системах.

Материал учебного пособия знакомит студентов, изучающих проблемы вычислительной техники, с инженерными методами защиты программ и данных в современных вычислительных системах. Наряду с описанием стандартных методов в учебное пособие вошли также материалы, освещающие следующие проблемы: архитектура ЭВМ и ее влияние на безопасность, общий обзор теоретических моделей безопасности, полное описание нового федерального стандарта по шифрованию данных, некоторые новые методы шифрования файлов с произвольным доступом, методы предотвращения несанкционированного или неправильного использования данных в статистических файлах, законодательные тенденции в области защиты секретности и конфиденциальности и их воздействие на проблемы проектирования систем.

В учебном пособии приведены основные определения и дан всесторонний обзор современных принципов, которыми следует руководствоваться при проектировании систем, обеспечивающих безопасность. Рассмотрены методы, устанавливающие полномочия и подлинность пользователей, терминалов и других средств, традиционные криптографические методы, а также программные и аппаратные криптографические методы, являющиеся наиболее эффективными и действенными в современных вычислительных системах. Определены основные правовые и нормативные документы.

# Глава 1

# Основные определения и современные принципы проектирования систем, обеспечивающих безопасность

## 1.1. Технологические аспекты обеспечения информационной безопасности

Под защитой информации понимается создание в ЭВМ информационных систем с технологиями искусственного интеллекта, совокупности средств, методов и мероприятий, предназначенных для предупреждения искажения, уничтожения или несанкционированного использования защищаемой информации.

По мере развития и усложнения средств, методов и форм автоматизации процессов обработки информации, цифровой трансформации в рамках формирования цифровой экономики Российской Федерации повышается ее уязвимость.

Основными факторами, способствующими повышению уязвимости, являются:

1) резкое увеличение накапливаемых, хранимых и обрабатываемых с помощью ЭВМ и других средств автоматизации объемов информации. Появление технологий Big Data, блокчейн, искусственного интеллекта;

2) сосредоточение в единых базах данных информации различного назначения и различной принадлежности;

3) появление новых типов баз данных Big Data, баз данных в одноранговых компьютерных сетях с технологией блокчейн, нейронных сетей;

4) резкое расширение круга пользователей, имеющих непосредственный доступ к ресурсам вычислительной системы и находящимся в ней базам данных;

5) усложнение режимов функционирования технических средств вычислительных систем: широкое внедрение многопрограммного режима, а также режимов разделения времени таймера операционной системы и внешнего таймера, работа в сети Интернет и облачные технологии;

6) кроссплатформенная обработка обмена информацией, в том числе и на больших расстояниях.

В этих условиях возникает уязвимость двух видов: с одной стороны, возможность искажения или уничтожения информации (нарушения ее физической целостности), а с другой — возможность ее несанкционированного использования (опасность утечки информации ограниченного пользования).

Второй вид уязвимости вызывает особую озабоченность пользователей ЭВМ, различных информационных систем, технологий искусственного интеллекта и блокчейн, в связи с чем этому аспекту уделяется повышенное внимание. В Интернете приводится статистика большого количества конкретных примеров хищения информации из информационных систем обработки данных, которые весьма убедительно иллюстрируют серьезность и актуальность проблемы защиты.

Основными потенциально возможными **каналами утечки информации** являются:

1) прямое хищение носителей и документов, обращающихся в процессе функционирования различных информационных систем;

2) запоминание или копирование информации, находящейся на машинных и на немашинных носителях;

3) несанкционированное подключение к аппаратуре и линиям связи или незаконное использование «законной» (зарегистрированной) аппаратуры (чаще всего терминалов пользователей);

4) несанкционированный доступ к информации за счет специального приспособления математического и программного обеспечения;

5) перехват электромагнитных волн, излучаемых аппаратурой информационных систем и ЭВМ, при обработке информации.

Следовательно, при наличии такого количества каналов утечки необходимы специальные средства, методы и мероприятия, предназначенные для перекрытия этих каналов и предупреждения несанкционированного использования информации.

В создании и обеспечении функционирования таких средств, методов и мероприятий и заключается защита информации в плане уязвимости второго вида.

Следует отметить, что выпускаемые в настоящее время ЭВМ и программное обеспечение (операционные системы, системы управления базами данных и т. п.) имеют некоторые средства защиты.

Некоторые ЭВМ содержат специальные схемы прерывания, позволяющие физически отделять исполнение программы пользователя от исполнения управляющих процедур, специальный блок защиты памяти, позволяющий контролировать и регулировать доступ пользователей и задач к защищаемым полям памяти, специальные регистры защиты и некоторые другие.

Во всех операционных системах современных ЭВМ предусматривается регулирование доступа к информации с помощью специальных паролей, четкое разделение ресурсов системы между решаемыми задачами, протоколирование вычислительного процесса и др.

Как показывает практика, серийных средств недостаточно для надежной защиты информации в современных сложных системах электронной обработки данных. Как следствие, во всех странах широко ведутся исследования и разработки различных средств и методов, которые позволили бы надежно перекрыть все потенциально возможные каналы утечки информации и этим обеспечить практически любую необходимую степень ее защиты.

Можно выделить **три направления работ по защите информации**: теоретические исследования, разработка средств защиты и обоснование способов использования средств защиты в информационных системах.

Отдельно следует выделить теорию и практику применения криптокодирования. Наиболее ярким примером является технология блокчейн.

**Блокчейн** (дословно «цепочка блоков») — это технология (структура данных и программный код) децентрализованного хранения данных, цепочка блоков транзакций, выстроенная по определенным правилам и обеспечивающая специфическую защиту от изменений с помощью криптографических хеш-функций.

Особую группу, как и в случае с аппаратными средствами, составляют программы шифрования информации.

Криптографическое закрытие (шифрование) информации заключается в таком преобразовании защищаемой информации, при котором по внешнему виду нельзя определить содержание закрытых данных. Криптографической защите все специалисты уделяют особое внимание, считая ее наиболее надежной, а для информации, передаваемой по линиям связи большой протяженности, — единственным средством защиты от хищений.

Основные направления работ по рассматриваемому аспекту защиты можно сформулировать таким образом:

– выбор рациональных систем шифрования для надежного закрытия информации;

– обоснование путей реализации систем шифрования в информационных системах;

– разработка правил использования криптографических методов защиты в процессе функционирования ИС;

– оценка эффективности криптографической защиты.

Наиболее существенные результаты по каждому направлению сводятся к шифрам, предназначенным для закрытия информации в ЭВМ и информационных системах. К ним предъявляется ряд требований, в том числе: достаточная стойкость (надежность закрытия), простота шифрования и расшифрования, независимость шифрования и расшифрования от способа внутримашинного представления информации, нечувствительность к небольшим ошибкам шифрования, возможность внутримашинной обработки зашифрованной информации, незначительная избыточность информации за счет шифрования и ряд других.

Исследования показывают, что особенно эффективными являются комбинированные шифры, когда текст последовательно

шифруется двумя или большим числом систем шифрования. Считается, что при этом стойкость шифрования превышает суммарную стойкость составных шифров.

Следующим классом в арсенале средств защиты информации являются физические меры. Это различные устройства и сооружения, а также мероприятия, которые затрудняют или делают невозможным проникновение потенциальных нарушителей в места, в которых имеется доступ к защищаемой информации. Чаще всего применяются следующие меры:

– физическая изоляция сооружений, в которых устанавливается аппаратура автоматизированной системы, от других сооружений;

– ограждение территории дата-центров (вычислительных центров) на таких расстояниях, которые достаточны для исключения эффективной регистрации электромагнитных излучений, и организация систематического контроля этих территорий;

– организация контрольно-пропускных пунктов у входов в помещения дата-центров или оборудование входных дверей специальными замками, позволяющими регулировать доступ в помещения;

– организация системы охранной сигнализации и др.

Организационными мероприятиями по защите информации являются такие нормативно-правовые щиты, которые регламентируют процессы функционирования системы обработки данных, использование ее устройств и ресурсов, а также взаимоотношение пользователей и системы таким образом, что несанкционированный доступ к информации становится невозможным или существенно затрудняется.

## 1.2. Организационные аспекты и принципы обеспечения информационной безопасности

Одной из задач обеспечения безопасности информации является разработка систем, которые реализуют политику управления доступом к информационным ресурсам организации.

В рамках организации контроля безопасности информации управление доступом играет ключевую роль в вопросе взаимодействия между получающими доступ сторонами: пользователями и ресурсами автоматизированных (информационных) систем.

С развитием информационных сетей общего пользования информационные ресурсы могут размещаться в распределенных вычислительных сетях, которые предполагают наличие области единых правил управления доступом, включающей в себя политику, процессы, технологии, стандарты и типовые модели разграничения доступа.

Управление доступом представляет собой часть общих основ управления идентичностью и доступом. Управление доступом осуществляется после успешно выполненных идентификации и аутентификации субъектов доступа, претендующих получить доступ к информационным ресурсам. Последовательность действий в модели управления доступом показана на рис. 1.1.

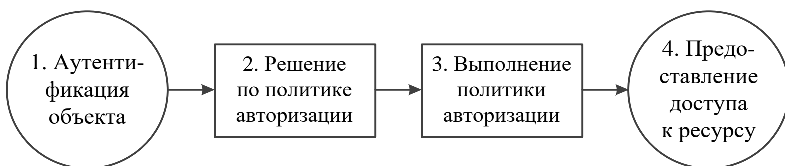


**Рис. 1.1.** Последовательность действий в модели управления доступом

В общем контексте **безопасность** понимается как обеспечение защиты данных от случайного или преднамеренного разрушения, раскрытия или модификации в телекоммуникационных системах. В Федеральном законе от 27 июля 2006 г. № 149-ФЗ «Об информации, информационных технологиях и о защите информации» в частности рассматриваются отношения, возникающие при обеспечении защиты информации. Эти отношения (в части определений) будут применяться в процессе изучения принципов обеспечения безопасности в телекоммуникационных системах. Некоторые определения будут вводиться и использоваться как учебный материал этого учебного пособия.

**Безопасность данных** — это защита данных от случайного или преднамеренного разрушения, раскрытия или модификации.

Безопасность в ЭВМ имеет отношение к технологическим мерам предосторожности и управляющим процедурам, которые могут быть приложены к аппаратным средствам, программам и данным, чтобы гарантировать защиту данных, помещаемых на хранение некоторой организацией или отдельным лицом. Лучшие практики по обеспечению безопасности данных включают такие методы защиты, как шифрование, управление ключами, редактирование, разделение на подмножества и маскирование данных, а также контроль доступа привилегированных пользователей, аудит и мониторинг.

**Секретность** — это право лица решать, какую информацию он желает разделить с другими, а какую хочет скрыть.

Проблема секретности возникла задолго до появления вычислительной техники, но повышенный интерес к ней лежит в возможности вычислительных машин хранить большие количества легко используемых данных.

Не существует прямой связи между секретностью и безопасностью. ЭВМ, обеспечивающую абсолютную безопасность (если такая существует), можно использовать таким образом, что секретные данные могут быть раскрыты.

**Конфиденциальность** — это статус, предоставленный данным и согласованный между лицом или организацией, предоставляющей данные, и организацией, получающей их. Она определяет требуемую степень защиты. Конфиденциальность данных относится не только к данным отдельных лиц, но и к любым данным, составляющим чью-либо собственность, или к секретным данным, обращение к которым требует конфиденциальности.

**Целостность** данных имеет место в том случае, когда данные в системе не отличаются от данных в исходных документах, т. е. не произошло их случайной или преднамеренной замены или разрушения.

Организационные мероприятия играют большую роль в создании надежного механизма защиты информации. Причины этого, по мнению специалистов, заключаются в том, что возможности несанкционированного использования информации в значительной мере обусловливаются нетехническими аспектами: злоумышленными действиями, нерадивостью или небрежностью пользователей, персонала систем обработки данных и т. п.

Влияния этих аспектов практически невозможно избежать или локализовать с помощью рассмотренных выше аппаратных и программных средств, криптографического закрытия информации и физических мер защиты. Для этого необходима совокупность организационных, организационно-технических и организационно-правовых мероприятий, которая исключала бы (или сводила к минимуму) возможность возникновения опасности утечки информации подобным образом. Основными мероприятиями этой совокупности являются следующие:

– мероприятия, осуществляемые при проектировании, строительстве и оборудовании вычислительных центров и других объектов систем электронной обработки данных (исключение влияния стихийных бедствий, возможностей тайного проникновения в помещения, обеспечение удобств для контроля прохода и перемещения людей и т. п.);

– мероприятия, осуществляемые при подборе и подготовке персонала вычислительных центров (проверка принимаемых на работу, создание условий, при которых персонал не хотел бы лишиться работы, обучение правилам работы с закрытой информацией, ознакомление с мерами ответственности за нарушение правил защиты и т. п.);

– организация надежного пропускного режима, организация хранения и использования документов и носителей: маркировка, определение правил выдачи, использования и возвращения, ведение журналов выдачи и использования и т. п.;

– организация работы в сменах дата-центров: выделение ответственных за защиту информации, организация контроля за работой персонала, ведение журналов работы, уничтожение установленным порядком производственных отходов и т. п.;

– контроль внесения изменений в математическое и программное обеспечения (строгое санкционирование, рассмотрение и утверждение проектов изменений, проверка всех изменений на удовлетворение требованиям защиты, документальное отображение изменений и т. п.);

– организация подготовки и контроля работы пользователей и др.

Одно из важнейших организационных мероприятий — создание в дата-центрах специальной штатной службы защиты

информации, численность и состав которой обеспечивали бы создание надежной системы защиты и регулярное ее функционирование.

Остановимся на законодательных мерах. К ним относятся действующие в стране законы, указы и положения, которые регламентируют правила обращения с информацией ограниченного использования и ответственность за их нарушения, препятствуя тем самым несанкционированному ее использованию. Законодательные меры являются препятствующим и сдерживающим фактором для потенциальных нарушителей.

Основные выводы в научных изданиях о способах использования рассмотренных выше средств, методов и мероприятий защиты сводятся к следующему:

– наибольший эффект достигается, если все используемые средства, методы и мероприятия объединяются в единый, целостный механизм защиты информации;

– механизм защиты должен проектироваться параллельно с созданием систем обработки данных, начиная с момента выработки общего замысла построения системы;

– функционирование механизма защиты должно планироваться и обеспечиваться наряду с планированием и обеспечением основных процессов автоматизированной обработки информации;

– необходимо осуществлять постоянный контроль функционирования механизма защиты.

Существует несколько фундаментальных принципов безопасности, которые необходимо соблюдать независимо от принципов построения архитектуры или проектирования.

**Фундаментальные принципы:**

– реализация функций безопасности должна быть такой, чтобы эти функции невозможно было обойти или подменить ненадежным кодом;

– функции безопасности должны выполняться каждый раз, когда необходимо осуществить реализацию политики безопасности; часть продукта, системы или приложения, реализующая функции безопасности, должна быть достаточно мала, чтобы ее можно было проанализировать на корректность и возможные побочные эффекты, критичные для безопасности;

– несекретность проектирования. Ознакомление опытных и квалифицированных специалистов с уязвимыми точками проекта на стадии разработки будет способствовать корректировке системы до ее реализации и до того, как другие системы начнут полагаться на этот проект с недостатками. Если кто-нибудь сделает попытку удержать в тайне механизм обеспечения безопасности, это может привести к тому, что злоумышленниками будут обнаружены уязвимые места проекта, в результате чего будет возможен несанкционированный доступ. Таким образом, лучше обнаружить уязвимости с помощью заранее приглашенных специалистов. Если описание проекта системы нельзя опубликовать в открытой печати, то ее нельзя считать достаточно надежной с точки зрения обеспечения безопасности, т. е. ею не могут пользоваться с полным доверием;

– удобство для пользователей. Система обеспечения безопасности должна быть удобна пользователям, иначе будут найдены пути ее обхода. Хорошая иллюстрация этого — пренебрежение миллионов недовольных пользователей блокированием автомобильного зажигания и ремнями безопасности. Интерфейс человека с системой должен быть простым, естественным и легким для использования, иначе пользователи будут обходить его, превращая систему обеспечения безопасности в неэффективную;

– полное посредничество. Система защиты должна проверять полномочия при каждом обращении к каждому объекту. Этот принцип требует, чтобы в случае изменений в полномочиях будущие обращения не могли использовать заполненных результатов предыдущих проверок полномочий, но сами могли быть перепроверены;

– минимум привилегий. Каждому пользователю, программе, терминалу или другому ресурсу следует давать только те привилегии, которые необходимы для завершения его задания;

– экономичность механизма. Проект необходимо делать настолько простым и малым, насколько это возможно;

– разделение привилегий. Требование наличия двух ключей для открытия любого механизма защиты более гибко и надежно, чем разрешение в доступе только по одному ключу. Если механизм закрыт, два ключа могут быть физически разнесены,

причем ответственными за них должны быть назначены различные программы, организации или лица;

– минимум общего механизма. Любой механизм, используемый многими пользователями, является потенциальным информационным путем к его раскрытию, поэтому такие пути должны быть минимальными.

## Выводы по главе 1

Определены понятия безопасности, секретности, конфиденциальности, целостности и их влияние на безопасность телекоммуникационных систем.

Рассмотрена последовательность действий в модели управления доступом к данным в информационной системе.

Изучено несколько принципов проектирования систем, обеспечивающих безопасность: несекретность проектирования, приемлемость пользователем, полное посредничество и минимальные привилегии.

Определено понятие целостности данных и важность этого свойства, когда данные в системе не отличаются от данных в исходных документах, т. е. не произошло их случайной или преднамеренной замены или разрушения.

## Вопросы для самоконтроля

1. Какой аргумент можно привести в пользу невынесения системы обеспечения безопасности на общее рассмотрение и критический анализ? Каковы слабые и сильные стороны аргумента?
2. Какая разница между секретностью и безопасностью, безопасностью и конфиденциальностью, конфиденциальностью и целостностью?
3. Каковы действия в модели управления доступом?
4. Каковы задачи обеспечения безопасности информации?
5. Каковы основные фундаментальные принципы безопасности?

# Глава 2

# Полномочия и подлинность пользователей

## 2.1. Идентификация и установление подлинности пользователя

До рассмотрения методов установления подлинности необходимо понять взаимосвязь между идентификацией, установлением подлинности и определением прав (полномочий), которые в совокупности определяют, какой доступ разрешается к защищаемым ресурсам.

**Идентификация** — это присвоение объекту уникального имени или числа.

**Установление подлинности** заключается в проверке, является ли лицо или объект тем, за кого себя выдает.

**Определение полномочий** устанавливает, дано ли лицу или объекту и в какой мере право обращаться к защищаемому ресурсу.

Указанные проверки используются совместно для принятия решения о доступе.

Идентификация пользователя, терминала, файла, программы или другого объекта представляет собой присвоение объекту уникального имени или числа. Идентификация является заявкой на установление идентичности. В идентификатор по возможности следует вводить контрольные цифры (защита информационной части) или использовать другие средства самоконтроля с целью минимизации шансов ошибочной идентификации.

Идентификация необходима не только для опознавания, но и для учета обращений, однако ее нельзя использовать без дополнительного установления подлинности, если в системе требуется определенная степень безопасности.

Установление подлинности заключается в проверке, является ли лицо (объект) тем, за кого себя выдает. Может быть затребована информация различного характера, прежде чем подлинность будет признана установленной. Подлинность устанавливается обычно только один раз, но в установках, обеспечивающих высокую степень безопасности, может потребоваться периодическая перепроверка или перепроверка при определенных условиях. Повторное установление подлинности может быть желательно, например, после всех системных сбоев. Процедура идентификации и установления подлинности показана на рис. 2.1.

Для установления подлинности пользователей ЭВМ употребляются пароли и другие методы диалога. Если пользователь в состоянии правильно представить требуемую информацию, ЭВМ признает его подлинность. Так как эти методы установления подлинности обычно применяются только однажды, требования к используемому при этом рабочему полю запоминающего устройства и машинному времени являются менее важными по сравнению с другими параметрами. С точки зрения пользователя, количество символов, подлежащих печатанию, количество времени, требуемого для обдумывания, и способ исправления ошибок при печати являются весьма важными.

Описываемые ниже сложные методы установления подлинности являются более дорогими с точки зрения машинного времени и удобства пользователя, чем схемы с простым паролем. При большом числе пользователей или при частом изменении их состава требования к памяти, ресурсам машины и программированию могут сделать эти методы практически неприменимыми. Но в то же время они обеспечивают большую степень безопасности. Проектировщику системы необходимо делать соответствующий выбор между противоречивыми требованиями увеличения степени обеспечения безопасности и простотой использования.

*Пароли.* Метод паролей требует, чтобы пользователь ввел строку символов (пароль) для проверки их в ЭВМ. Если пароль соответствует тому, который хранится в ЭВМ, пользователь мо-

жет пользоваться всей информацией, доступ к которой ему разрешен (пароли можно использовать независимо от пользователя для защиты файлов, записей, полей данных внутри записей и т. д.).

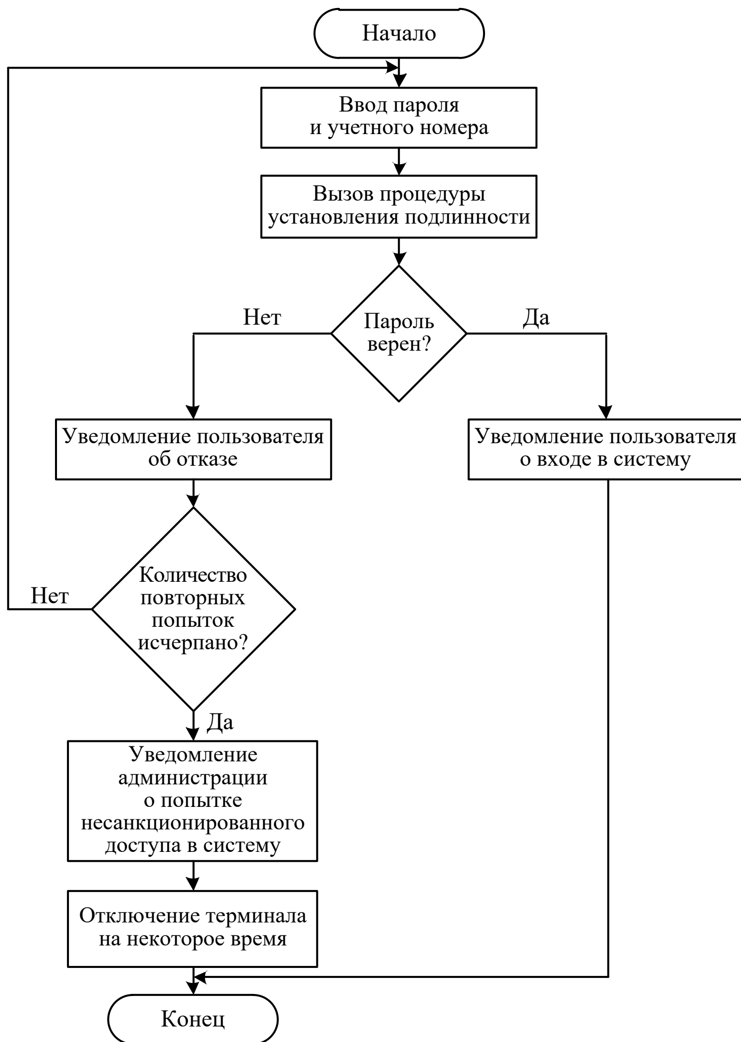


**Рис. 2.1.** Процедура идентификации и установления подлинности

*Простой пароль.* Схема простого пароля очень легка для использования, так как пользователю необходимо только ввести с клавиатуры пароль (рис. 2.2).

**Выполнить вход**

Вам доступен вход по локальной учетной записи

Пароль

Запомнить меня

**Выполнить вход**

**Рис. 2.2.** Форма для ввода пароля доступа

Реализация этого метода обходится дешево. При этом пользователь может самостоятельно выбрать себе пароль.

*Выбор пароля.* В схеме с простым паролем и в некоторых ее вариантах пользователю разрешается самому выбирать пароль, который он легко запомнит. Также следует позаботиться и о том, чтобы пароль не был слишком очевидным. Если зарегистрированный пользователь выбирает сложный пароль, для лучшего запоминания его можно записать. В этом случае возникает риск случайного обнаружения посторонним лицом пароля на выброшенном листе бумаги, например, на использованном протоколе с пульта терминала. Одно из решений этой проблемы заключается в использовании генератора «произносимых» случайных паролей, которые являются довольно легкими для запоминания. Один из нередко используемых методов предполагает наличие пробелов в ряду символов пароля и в конце его. В этом случае незаконно полученный лист бумаги с паролем не позволит автоматически раскрыть его тайну.

Недостатком схемы с простым паролем является то, что пароль может быть легко использован другим лицом без ведома зарегистрированного пользователя (вплоть до конца месяца, когда ему будет представлен счет). В схеме с простым паролем нет необходимости делать какой-либо физический дубликат, и потому тот факт, что другие лица обладают этим паролем, может быть никому не известен (в отличие от обыкновенного ключа, который, например, кто-либо имеет или не имеет).

Одним из путей решения этой проблемы является выдача системой на терминал пользователя каждый раз, когда он пользуется системой, номера, за которым зарегистрировано его обращение к системе в этот день и продолжительность работы. На печать терминала можно вывести также текущий счет на конкретный момент времени. Если за это время кто-либо воспользовался чужим счетом, бдительный пользователь сможет это обнаружить.

Конечно, если нескольким пользователям приписан один и тот же номер пользователя, эта схема становится непрактичной. Но обычно практика обеспечения сохранности требует, чтобы каждому пользователю было задано уникальное идентификационное число для обеспечения безопасности. Если для проведения расчетов (или других целей) пользователей с системой желательно использование номера подразделений или номера проектов, их можно запрашивать в процессе опознавания совместно с идентификаторами пользователя или производить более мелкие расчеты, например, отдельно для каждого пользователя.

Сформировать пароль можно с помощью криптометодов. Простым криптокодом являются подстановочные шифры.

Подстановочным (substitution cipher) называется шифр, в котором каждый символ открытого текста заменяется другим символом в шифротексте. Получатель выполняет обратную подстановку в шифротексте, восстанавливая открытый текст.

В классической криптографии существуют **четыре типа подстановочных шифров**.

1. *Простой подстановочный шифр* (simple substitution cipher), или моноалфавитный шифр (monoalphabetic cipher) — это шифр, который заменяет каждый символ открытого текста соответствующим символом шифротекста. Примером простых подстановочных шифров являются криптограммы в газетах.

2. *Омофонический подстановочный шифр* (homophonic substitution cipher) похож на простую подстановочную криптосистему за исключением того, что один символ открытого текста заменяется несколькими символами шифротекста. Например, букве А может соответствовать набор чисел 5, 13, 25 или 56, букве В — 7, 19, 31, 42 и т. д.

3. *Полиграммный подстановочный шифр* (polygram substitution cipher) — это шифр, который заменяет одни блоки симво-

лов другими. Например, символам ABA могут соответствовать символы RTQ, символам ABB — символы SLL и т. д.

4. *Полиалфавитный подстановочный шифр* (polyalphabetic substitution cipher) состоит из нескольких простых подстановочных шифров. Например, можно использовать пять разных простых подстановочных фильтров так, что каждый символ открытого текста заменяется с использованием одного конкретного шифра.

Из полиалфавитных подстановочных шифров самый простой алгоритм получил название «Шифр Цезаря», в котором каждый символ исходного текста заменяется символом, находящимся на некотором постоянном числе позиций левее или правее него в алфавите. Например, в шифре со сдвигом вправо на два элемента А была бы заменена на В, Б стала Г и т. д. (рис. 2.3). Шифр Цезаря представляет собой простой подстановочный фильтр. На самом деле этот алгоритм еще проще, чем подстановочный, потому что алфавит шифротекста представляет собой результат смещения алфавита открытого текста, а не его случайную перестановку.

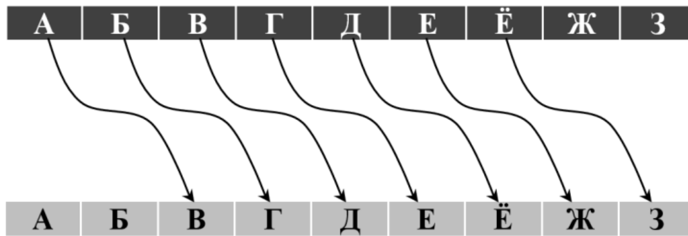


**Рис. 2.3.** Шифр Цезаря

Рассмотрим программу формирования шифротекста Цезаря в языке Python.

В начале программы определяем русский и английский алфавиты.

```
alfavit_EU =
'ABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZ'
alfavit_RU =
'АБВГДЕЁЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯАБВГДЕЁЖЗИЙКЛМНОПРСТУФХЦЧШ
ЩЪЫЬЭЮЯ'
```

```
# Вручную будет определяться шаг сдвига в шифротексте
# и сообщение для шифровки, переменные в программе smeshenie
# и message, а также переменная для шифротекста itog

smeshenie = int(input('Шаг шифровки: '))
message = input("Сообщение для шифровки: ").upper()
itog = ''
```

В программе можно использовать русский или английский текст. Тип языка вводится вручную.

```
lang = input('Выберите язык RU/EU: ') #Добавляем возможность
выбора языка
```

Алгоритм формирования шифротекста и его печать будут следующими:

```
if lang == 'RU':
  for i in message:
    mesto = alfavit_RU.find(i) # Алгоритм для шифрования
сообщения на русском
    new_mesto = mesto + smeshenie
    if i in alfavit_RU:
      itog += alfavit_RU[new_mesto]
    else:
      itog += i
else:
  for i in message:
    mesto = alfavit_EU.find(i) # Алгоритм для шифрования
сообщения на английском
    new_mesto = mesto + smeshenie
    if i in alfavit_EU:
      itog += alfavit_EU[new_mesto]
    else:
      itog += i
print (itog)
```

*Дешифровка шифротекста (сообщения).* Алгоритм дешифровки шифротекста является обратным шифровке. Изменения произойдут только в части величины шага, знак должен указываться « - ».

```
alfavit_EU =
'ABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZ'
alfavit_RU =
'АБВГДЕЁЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯАБВГДЕЁЖЗИЙКЛМНОПРСТУФХЦЧШ
ЩЪЫЬЭЮЯ'
```

```
smeshenie = int(input('Шаг дешифровки (не забудьте про
знак): '))
message = input("Сообщение для Дешифровки: ").upper()
itog = ''
lang = input('Выберите язык RU/EU: ')
if lang == 'RU':
  for i in message:
    mesto = alfavit_RU.find(i)
    new_mesto = mesto + smeshenie
    if i in alfavit_RU:
      itog += alfavit_RU[new_mesto]
    else:
      itog += i
else:
  for i in message:
    mesto = alfavit_EU.find(i)
    new_mesto = mesto + smeshenie
    if i in alfavit_EU:
      itog += alfavit_EU[new_mesto]
    else:
      itog += i
print (itog)
```

В небольших, устойчивых к изменениям сетях, где каждая ЭВМ известна, проверка подлинности происходит децентрализовано, в каждом узле. В больших динамических сетях, где численность пользователей или даже конфигурация сети изменяется относительно быстро, применяются более сложные программные решения. Наилучший вариант Identity в ASP.NET Core рассмотрим подробно далее.

Рассмотрим **меры предосторожности, связанные с использованием паролей**.

1. Пароли никогда не следует хранить в вычислительной системе в явной форме: они всегда должны быть зашифрованы, и их безопасность должна обеспечиваться недорогими и эффективными средствами.

Случалось, что незашифрованные пароли обнаруживали в метках томов, в программах, используемых для вызова других программ, и, конечно, в файле паролей.

Можно использовать простой метод обратимого шифрования или более сложный метод необратимой беспорядочной сборки, когда несколько паролей в явной форме преобразуются в одинаковый зашифрованный пароль. В этом случае не существует никакой схемы для возвращения к оригиналу пароля. Система

шифрует каждый пароль пользователя во время процесса регистрации и сверяет его с зашифрованным паролем, хранящимся в собственном файле пользователя.

Рассмотрим общее полиномиальное необратимое представление:

$$f(x) = (x^n + a_1 x^{n_1} + a_2 x^3 + a_3 x^2 + a_4 x + a_5) \bmod P,$$

где $P = 2^{64} - 59$; $n = 2^{24} + 17$; $n_1 = 2^{24} + 3$; $a_i$ — любое произвольное 19-разрядное число ($i = 1, 2, 3, 4, 5$); $x$ — пароль в явной форме; $f(x)$ — зашифрованный пароль.

Проектировщик системы, вероятно, желал бы выбрать свои собственные значения коэффициентов для общего полиномиального необратимого представления. Это выражение — хорошо подобранное подмножество полиномов $F(x)$:

$$F(x) = (x^n + a_1 x^{n-1} + a_2 x^3 + ... + a_{n-1} x + n) \bmod P,$$

где $P$ — некоторое большое число; $a_i$ — целые числа.

2. Может оказаться целесообразным требование включения пользователем своего пароля в управляющие записи задания.

Эти задания часто считываются на спулинговые файлы (некоторый генерируемый дополнительный объем данных на жестком диске, который используется для выполнения запроса), которые недостаточно защищены. Спулинговые программы должны осуществлять специальные действия для шифрования паролей до того, как они будут записаны в спулинговый файл, или еще лучше, если они сами проверяют пароль сразу же после того, как он будет считан с перфокарты.

3. Пароли не следует показывать в явном виде на мониторе, в том числе и в распечатках. Если это возможно, то во время ввода пароля печатающий механизм должен быть выключен. В системах, где характеристики мониторов делают это невозможным, пароль подпечатывается на маску, представляющую собой некий черный узор. Затем у пользователя запрашивается пароль. Пользователь печатает его поверх черных пятен, создавая трудности для чтения пароля посторонним лицом. В других системах пароль забивается после того, как пользователь его напечатает.

Хотя эта схема обеспечивает некоторую дополнительную надежность, она не является вполне безопасной. Если для маски и пароля используется некоторая известная комбинация букв, можно прочесть его, держа бумагу над источником света. Необходимо проводить эксперименты с терминалами любой системы для определения эффективных символов, обеспечивающих хорошие маски. Однако эта минимальная мера предосторожности может быть преодолена, даже если терминалы будут иметь набор взаимозаменяемых шрифтов.

4. Чем больший период времени используется один и тот же пароль, тем выше вероятность, что он будет раскрыт. Следовательно, чтобы достичь наибольшей эффективности, пароли нужно менять довольно часто.

Иногда желательно задавать два пароля одновременно. Каждый пароль соответствует определенному периоду времени, причем пароли по времени следуют один за другим, а интервалы могут перекрываться. Это позволяет осуществлять автоматическую смену паролей в заранее определенный момент времени. Переход к новому набору паролей может происходить до некоторой степени непрерывно, так как устаревший пароль пользователя является правильным для нескольких первых дней второго временного периода.

5. Система никогда не должна вырабатывать новый пароль в конце сеанса связи. Если постороннее лицо сможет похитить его и немедленно использовать, оно будет иметь доступ к системе, а зарегистрированный пользователь будет от системы изолирован.

Для установления подлинности пользователей широко используется *процедура рукопожатий* (handshaking — согласованный обмен, квитирование), построенная по принципу «вопрос-ответ». Система защиты может потребовать, чтобы пользователь установил свою подлинность с помощью корректной обработки алгоритмов. Это процедура рукопожатий также может быть выполнена как между двумя ЭВМ, так и между пользователем и ЭВМ. Процедуры в этом режиме обеспечивают большую степень безопасности, чем многие другие схемы, но вместе с тем являются более сложными и требующими дополнительных затрат времени для пользователя. Необходимо найти компромисс между требуемой степенью безопасности и простотой использования.

Например, для установления подлинности ЭВМ могла дать пользователю число, выбранное случайным образом, а затем запросить от него ответ. Для подготовки ответа пользователь $p$ применяет собственное, заранее подготовленное преобразование $t_p$. Информацией, на основе которой принимается решение, здесь является не пароль, а преобразование $t_p$. Система защиты посылает значение $x$, а пользователь отвечает значением $t_p(x)$. Любое постороннее лицо для проникновения в систему даже в случае знания значения $x$ и $t_p(x)$ должно отгадать функцию преобразования на основе нескольких вводов и выводов, так как сама функция преобразования никогда не передается по линиям связи, по которым посылается только $x$ и $t_p(x)$.

## 2.2. Установка профилей полномочий пользователя

В сетях ЭВМ применяются технологии защищенного канала, включающие три основные составляющие:

1) взаимную аутентификацию абонентов при установлении соединения;

2) защиту передаваемых по каналу сообщений от несанкционированного доступа;

3) подтверждение целостности поступающих по каналу сообщений.

В сетевых протоколах предусмотрены разные технологии защищенного канала. Например, новая версия протокола IP (Internet protocol) предусматривает все три составляющие технологии на сетевом уровне, а протокол туннелирования PPTP (point-to-point tunneling protocol) защищает данные на канальном уровне.

В зависимости от места расположения программного обеспечения защищенного канала различают две схемы его образования. В схеме с конечными узлами (рис. 2.4) защищенный канал образуется программными средствами, установленными на двух удаленных компьютерах. Компьютеры принадлежат двум разным локальным сетям одной организации и связаны между собой через публичную сеть.

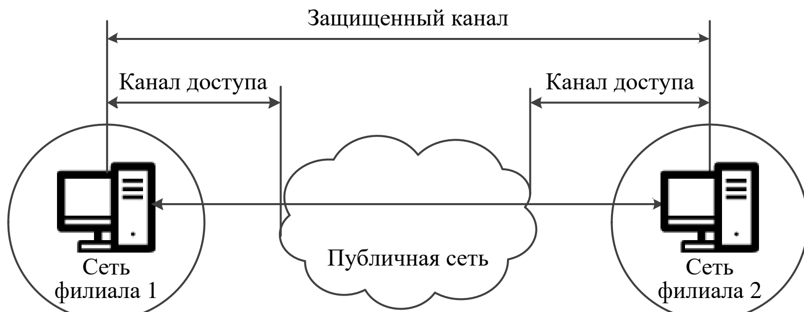


**Рис. 2.4.** Схема с конечными узлами

Достоинство этой схемы — полная защищенность канала вдоль всего пути следования и возможность использования любых протоколов создания защищенных каналов, чтобы на конечных точках канала поддерживался один и тот же протокол.

Недостатки заключаются в уязвимости сетей с коммутацией пакетов, а не каналов телефонной сети или выделенных каналов, через которые локальные сети подключены к территориальной сети. Следовательно, защиту каналов доступа к публичной сети можно считать избыточной. Для предоставления услуг защищенного канала на каждом компьютере требуется отдельно устанавливать, конфигурировать программный пакет коммуникаций.

На рис. 2.5 показана схема с защитой между пограничными устройствами доступа.

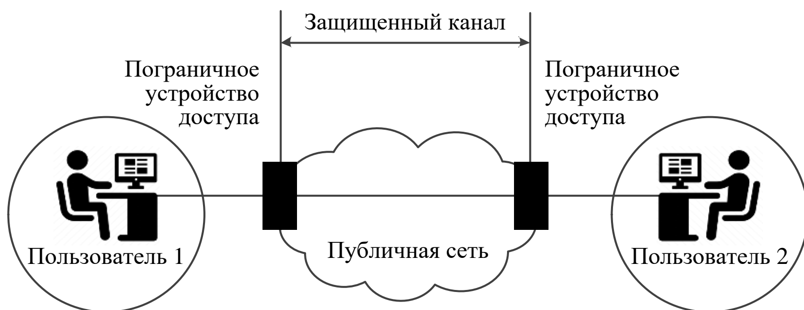


**Рис. 2.5.** Схема с защитой
между пограничными устройствами доступа

В этой схеме защищенный канал прокладывается только внутри публичной сети с коммутацией пакетов, т. е. клиенты и серверы не участвуют в создании защищенного канала. Канал может быть проложен между сервером удаленного доступа поставщика услуг публичной сети и пограничным маршрутизатором корпоративной сети. В этом случае канал управляется централизованно администратором корпоративной сети и администратором сети поставщика услуг. Такой подход позволяет легко образовывать новые каналы защищенного взаимодействия между компьютерами независимо от их места расположения, так как программное обеспечение конечных узлов остается без изменений. Реализация этой схемы сложнее: нужен стандартный протокол образования защищенного канала; требуется установка у всех поставщиков услуг программного обеспечения, поддерживающего такой протокол; необходима поддержка протокола производителями пограничного коммуникационного оборудования. Кроме того, оказываются незащищенными каналы доступа к публичной сети, и потребитель услуг находится в полной зависимости от надежности поставщика услуг. Тем не менее, специалисты прогнозируют, что именно вторая схема в ближайшем будущем станет основной в построении защищенных каналов.

Иногда после осуществления процедуры установления подлинности с использованием одного из методов, рассмотренных ранее, могут быть проверены полномочия запросов, вводимых пользователем, терминалом или другим ресурсом.

Если дается разрешение на выполнение затребованного действия, мы говорим, что объект, осуществляющий запрос, имеет полномочия по отношению к этому элементу данных. Элементом данных может быть файл, запись, поле, отношение или некоторая другая структура. Будет дано разрешение на доступ или нет, зависит от нескольких факторов: прав пользователя на доступ, прав терминала на доступ, требуемого действия, самого элемента данных, значения элемента данных, или, например, времени дня. Обычно рассматриваются не все эти факторы.

Система обеспечения безопасности поддерживает некоторые профили полномочий каждого пользователя, терминала, процедуры или другого ресурса, который осуществляет доступ к эле-

ментам данных. Эти профили устанавливаются в системе с помощью специальной привилегированной программы, и их можно представить в виде матрицы установления полномочий.

Независимо от того, вводятся новые профили или изменяются существующие, необходимо осуществить процедуру регистрации. В зависимости от степени секретности данных дополнительно могут быть произведены другие возможные действия, а именно:

– однодневная задержка действий по установлению (изменению) профиля (период замораживания);

– задержка установления (изменения) профиля до тех пор, пока другой уполномоченный пользователь не предпримет такие же действия («приятельская» система, требующая двух подписей для осуществления больших изменений);

– задержка установления (изменения) профиля до тех пор, пока не будет дан сигнал от специфического пользователя в системе («приказ управляющего»).

Цель этих специфических мер состоит в том, чтобы обеспечить некоторый независимый взгляд, выступающий в роли арбитра, иначе говоря, снимающий «оковы» для выполнения полномочий.

## Выводы по главе 2

В главе даны определения различия между идентификацией и установлением подлинности, представлено (вместе с их преимуществами и недостатками) несколько методов установления подлинности: простой пароль, выборка символов, пароль однократного использования, метод «запрос-ответ» и процедуры в режиме «рукопожатия». Обсуждены некоторые соображения, которых следует придерживаться в процессе реализации различных методов. Рассмотрена технология защищенного канала в сетях ЭВМ.

Такие технологии широко используются, когда требуются дополнительные меры по защите передаваемой информации. Требование конфиденциальности особенно важно, потому что пакеты, передаваемые по публичной сети, уязвимы для перехвата

при прохождении через каждый из узлов (серверов) на пути от источника к получателю. Технология защищенного канала включает три основные составляющие: взаимную аутентификацию абонентов при установлении соединения, защиту передаваемых по каналу сообщений от несанкционированного доступа, подтверждение целостности поступающих по каналу сообщений.

## Вопросы для самоконтроля

1. Какими недостатками обладают все схемы с паролями?
2. Каким образом постороннее лицо может перехватить секретную информацию в сети ЭВМ, даже если установление подлинности производится в обоих направлениях? Что это означает по отношению к обеспечению безопасности, создаваемой только схемами с паролями?
3. Какие дополнительные недостатки имеет схема с паролем однократного использования, применяемая совместно со схемой простого пароля? Обеспечивается ли за счет этого увеличение степени безопасности?
4. Какие преимущества и недостатки имеет алгоритм установления подлинности по сравнению со схемой с паролем?
   Сравните величину дополнительной безопасности, получаемую при использовании выхода на служебную программу пользователя, с системой, использующей пароль, и с процедурой в режиме «рукопожатия»?
5. Допустим, что процедура ввода обращения пользователя состоит из печатания совокупности имени и пароля. При вводе неверного пароля пользователь имеет возможность набрать новый пароль после того, как будет напечатано сообщение об ошибке, что занимает 5 с. После введения трех неправильных паролей происходит отключение пользователя от системы. Если требуется 3 с для набора имени и 1 с для набора каждого символа пароля и если алфавит, из которого составляются пароли, состоит из 100 символов, а процедура отключения занимает 3 с, что обеспечивает большую степень безопасности и почему?
   а) увеличение длины пароля с трех до четырех символов;
   б) введение дополнительной одноминутной задержки к процедуре отключения.

6. Файл имеет внутренний 32-битовый ключ. Вызов и проверка для открытия замка файлов требует 20 мкс. Определите ожидаемое безопасное время при использовании для открытия пароля метода проб и ошибок.
7. Типичные процедуры идентификации и установления подлинности требуют, чтобы удаленный пользователь набрал на терминале имя и пароль. Какие недостатки можно определить?

# Глава 3

## Кибербезопасность средствами микроконтроллеров

С преобладанием постоянных подключений и достижений в технологиях, которые доступны на сегодняшний день, виды киберугроз быстро развиваются, нацеливаясь на различные аспекты информационных технологий. Любое устройство уязвимо для атаки, а с появлением концепции Интернета вещей (Internet of things, IoT) это стало реальностью. В октябре 2021 г. на DNS-серверы была проведена серия DDos-атак, в результате чего перестали работать некоторые основные веб-сервисы. Это стало возможным из-за большого количества небезопасных IoT-устройств по всему миру. В то время как использование IoT для запуска масштабной кибератаки является чем-то новым, наличие уязвимости в этих устройствах таковым не является. На самом деле они были там довольно давно.

Это позволяет сделать вывод, что компании должны уделять особое внимание аутентификации и авторизации пользователей и их прав доступа.

Личность пользователя — это новый параметр, который требует мер безопасности, специально разработанных для аутентификации и авторизации лиц на основании их работы и потребности в конкретных данных сети. Кража учетных данных может быть только первым шагом киберпреступников к системе. Наличие действующей учетной записи пользователя в сети позволит им распространяться дальше и в какой-то момент найдет правильную возможность повысить привилегию до учетной записи администратора домена.

Еще одной растущей тенденцией для защиты личных данных пользователей является применение многофакторной аутентификации. Один из методов, который получил более широкое распространение, — это функция обратного вызова, когда пользователь первоначально аутентифицируется, используя свои учетные данные (имя пользователя и пароль), и получает вызов для ввода своего пин-кода. Если оба фактора аутентификации успешны, им разрешен доступ к системе или сети.

## 3.1. Микроконтроллеры как технология защиты от киберугроз

Технологии четвертой промышленной революции характеризуются более мощным и всеобъемлющим Интернетом, более совершенными и умными подключаемыми к нему компонентами, открытием большого количества областей для их применения. Однако это может привести к критичным последствиям, связанным с нарушением безопасности, кибератаками, что требует принятия соответствующих мер по предотвращению уязвимости.

Основой сенсорных технологий четвертой промышленной революции «Индустрия 4.0» (Industry 4.0) и Интернета вещей являются микроконтроллеры и их программное обеспечение.

Пример микроконтроллеров приведен на рис. 3.1. Это 32-битные микроконтроллеры архитектуры RISC — архитектурный подход к проектированию процессоров, в котором быстродействие увеличивается за счет такого кодирования инструкций, чтобы их декодирование было более простым, а время выполнения — меньшим. В системах команд первых RISC-процессоров даже отсутствовали команды умножения и деления.

**Рис. 3.1.** Микроконтроллеры архитектуры RISC

Именно они в рамках подключенных «умных» фабрик и «умных» домов предлагают огромный потенциал для роста и инноваций в этом бурно развивающемся бизнесе, но в то же время именно они делают системы уязвимыми для внешних атак.

Некоторые семейства микроконтроллеров уже включают множество функций безопасности. Дело в том, что микроконтроллеры являются основными компонентами в среде управления в подключенных системах. Их поставщики уже используют процессы разработки и сертификацию по соответствующим стандартам безопасности. А поставщики полупроводников также гарантируют, что могут предложить своим клиентам безопасное комплексное решение.

С точки зрения безопасности микроконтроллеры можно классифицировать согласно их целевым конечным приложениям.

1. Решения в области аутентификации и доверенные платформенные модули (trusted platform module, TPM), например, для защиты как непосредственно самого пользователя, так и сетей IoT.

2. Банковские и идентификационные решения для классических компаний-производителей и эмитентов смарт-карт, используемых в сфере обработки платежей, в качестве персональных идентификаторов, для оплаты услуг транспорта и в системах доставки платного контента для телевидения.

3. Мобильные решения безопасности для решений на базе SIM-карт в мобильных продуктах и приложениях межмашинного взаимодействия — M2M (machine-to-machine).

4. Автомобильные решения для коммуникации ближнего поля (NFC, eSE) и систем обеспечения безопасного вождения.

*Интегрированные функции защиты данных.* Робототехника, IoT и Industry 4.0 в основном используют стандартные микроконтроллеры, созданные для промышленного и бытового применения (микроконтроллеры общего назначения). Но также уже доступны и модели со встроенными функциями безопасности. Например, семейство микроконтроллеров STM32 (семейство 32-битных микроконтроллеров производства STMicroelectronics), которое имеет множество встроенных функций, обеспечивающих их защиту, в том числе:

– защиту от кражи личных данных (защита от манипуляций, защита целостности, отслеживаемость движения продукта);

– отказ в обслуживании данных (регулирование);

– защиту от отслеживания и манипулирования данными и кодом (защита памяти, управление правами доступа, уровень отладки, защита от манипуляций, защита целостности, безопасные обновления прошивки);

– защиту от физического (механического) вмешательства (защита от манипуляций на кристалле).

Эти функции в основном реализуются их интеграцией непосредственно на кристалле микроконтроллера. Они обеспечивают надежную проверку подлинности (верификацию), целостность платформы и постоянную защиту данных, включая защиту конфиденциальности конечных пользователей, а также комплексную защиту данных, IP-адресов и брендинга и отвечают самым высоким требованиям безопасности данных для стандартных продуктов. Типичные целевые приложения таких микроконтроллеров — это, например, принтеры, компьютеры, шлюзы, конечные точки IoT и различные датчики.

## 3.2. Функции защиты от киберугроз на аппаратной основе

На аппаратной основе используется циклический контроль избыточности кода (cyclic redundancy check calculate), т. е. вычисляется контрольная сумма, которая выявляет ошибки при передаче или хранении данных. Это не только обеспечивает проверку целостности кода, но и означает, что сигнатура программного обеспечения (ПО) может быть рассчитана во время его работы.

Мониторинг питания — еще один метод с высокой степенью защиты. Для определения причины сброса и обеспечения сброса только с помощью аутентифицированного доступа используется система управления статусом флага POR (power on RESET — «включение питания RESET»), PDR (power down RESET — «отключение питания RESET»), BOR (brown out RESET — «снижение питания RESET»), PVD (programmable voltage detector — «программируемый детектор напряжения»). Для эффективного обнаружения манипуляций и ведения журнала все это дополняется функцией «Read while Write» (буквально:

«читай во время записи», т. е. считывание одного слова во время записи другого слова).

Функциональность CSS (clock security system — «система безопасности тактирования») основана на том, что при использовании внешнего генератора (в микроконтроллерах серии ST32 он обозначен как HSE) в качестве источника тактового сигнала тактовой частоты система не останется в неопределенном состоянии, а сможет выполнить какие-то действия, SYSCLK или PLL (система ФАПЧ). Если произойдет срыв генерации, CSS автоматически переключит всю систему на работу от встроенного RC-генератора (в микроконтроллерах серии ST32 — HSI). Таким образом, если что-то случится с тактовыми сигналами, то можно перевести объект управления микроконтроллером в безопасное состояние. Кроме того, сторожевой таймер (watchdog) и оконный сторожевой таймер (window watchdog) также контролируют временные окна независимо друг от друга.

Целостность и достоверность содержимого памяти обеспечиваются проверкой и исправлением ошибок кода (error correction code, ECC) и, как уже было сказано, проверкой четности. Здесь тоже обеспечивается дополнительная защита от атак, направленных на недопущение заражения систем ошибками кода. Кроме того, датчик температуры непрерывно измеряет температуру среды, окружающей микроконтроллер. Это необходимо для того, чтобы убедиться, что она остается в указанном диапазоне, и таким образом избежать риска его повреждения при специальном длительном нагреве.

*Правильно выполненное шифрование.* Методы шифрования защищают путем кодирования оригинальный исходный текст от несанкционированного доступа к нему.

*Предотвращение манипуляций.* Защита от манипуляций включает защитные механизмы для предотвращения преднамеренных или непреднамеренных физических атак на аппаратную систему вне микроконтроллера.

Резервный домен, связанный с различными источниками пробуждения, гарантирует, что защита также поддерживается в режиме низкого энергопотребления.

Часы реального времени (real time clock, RTC) назначают метку времени каждому событию манипуляции.

## 3.3. Дополнительные средства защиты от кибератак

Блокировка отладки предотвращает несанкционированный доступ к микроконтроллеру через интерфейс отладки. Уровень безопасности может быть выбран в зависимости от приложения и требований, однако, выбрав однажды, его впоследствии нельзя уменьшить.

Права доступа позволяют пользователям или группам пользователей выполнять определенные действия. Для этого встроенный модуль защиты памяти (memory protection unit, MPU) делит память на области с различными правами и правилами доступа.

Когда выполняется передача данных, брандмауэр защищает код или часть данных флеш-памяти или SRAM от кода (или фрагментов), выполняющегося за пределами защищенного сектора. Брандмауэр более строг, чем MPU, но он интегрирован только в микроконтроллеры STM32 серий L0 и L4.

Функция защиты от чтения используется для управления контролем доступа к памяти. В общем, это предотвращает сброс данных оперативной памяти, например, резервное копирование пользовательских IP-адресов.

Защита от записи оберегает каждый сектор от нежелательных операций записи. Собственная защита кода позволяет настраивать каждый сектор памяти как execute only (только для выполнения), т. е. код в нем может только выполняться, но не записываться.

Функции полного и безопасного стирания позволяют безопасно удалять IP-адреса и конфиденциальные данные. Это действие полностью сбрасывает память к заводским настройкам по умолчанию.

Для обеспечения отслеживаемости конечного продукта многие серии микроконтроллеров STM32 имеют 96-битный уникальный идентификатор. Это также может быть использовано для диверсификации ключей безопасности.

Многие серии рассматриваемых микроконтроллеров содержат дополнительные функции безопасного обновления прошивки. Таким образом, аппаратные функции безопасности могут быть

в любой момент еще больше расширены с помощью тех или иных программных мер.

Защита конечного продукта от манипуляций со стороны третьих лиц основана на интегрированных программных решениях и используемых компонентах электронного оборудования. Микроконтроллеры и микросхемы памяти, а где это уместно и в сочетании с датчиками и интегральными микросхемами, являются ключевыми для приложений IoT и Industry 4.0. В связи с General data protection regulation, или сокращенно GDPR (Общий регламент ЕС о защите данных), который вступил в силу 25 мая 2018 г., компания Rutronik для семейств микроконтроллеров подготовила информацию об интегрированных в них функциях безопасности. В этот комплект документации включены таблицы с перечнем систем защиты от тех или иных манипуляций, модули шифрования, управление разрешениями, уровни блокировки отладки, данные по защите памяти, а также вопросы гарантирования целостности и функциональной безопасности.

Оценка относящихся к безопасности функций, перечисленных в таблицах, в отношении интегрированной защиты данных в портфеле микроконтроллеров Rutronik дает клиентам компании полное информативное понимание по безопасности. В них, кроме различных семейств микроконтроллеров STM32, присутствует и ряд продуктов компании Renesas, а именно CISC-микроконтроллеров RX и семейства Synergy S1/S3, которые в отношении функций безопасности предлагают степень защиты выше средней. При этом отдельные микроконтроллеры в категории Synergy S5/S7 компании Renesas полностью соответствуют всем требованиям по защите.

Кроме того, здесь следует подчеркнуть полностью интегрированную поддержку как симметричных, так и асимметричных методов шифрования, в том числе интегрированную генерацию ключей на основе AES (128/192/256), 3DES/ARC4, RSA/DAS или DLP.

Микроконтроллеры семейства RX можно рассматривать как первые в плане полного охвата различных функций безопасности, а также поддержки интегрированных механизмов для симметричного и асимметричного шифрования.

# Выводы по главе 3

Рассмотрены методы, модели, методики, алгоритмы и архитектуры, выбранные в соответствии с обобщенной архитектурой перспективной системы управления инцидентами безопасности критически важных объектов, которые могут быть успешно реализованы для управления инцидентами безопасности в рамках комплексной защиты информационных систем Индустрии 4.0. При этом комплексность защиты достигается за счет применения микроконтроллеров.

Отмечено, что микропроцессоры образуют прототип системы управления инцидентами безопасности для комплексной защиты элементов информационных систем и технологий Индустрии 4.0.

# Вопросы для самоконтроля

1. Образуют ли микроконтроллеры множество функций безопасности?
2. Каково влияние микропроцессоров на интегрированные функции защиты данных?
3. Каковы функции микроконтроллеров при защите на аппаратной основе?
4. Как взаимосвязаны микроконтроллеры и шифрование данных информационной системы?
5. Какие существуют микроконтроллеры и средства защиты от манипуляций данными?
6. Как влияет технология микроконтроллеров на симметричный метод шифрования?
7. Какие дополнительные средства защиты от нападения обеспечивают микроконтроллеры?

# Глава 4

# Криптографические методы защиты информации

## 4.1. Элементы криптосистемы

Сегодня понятие криптографии значительно расширилось и включило в себя аутентификацию, цифровые подписи и множество других элементарных функций безопасности.

Исследованием и разработкой методов преобразования информации с целью ее защиты занимается криптография.

Современный период в ее развитии характеризуется разработкой большого количества различных методов шифрования, созданием теоретических и практических основ их применения. Возросшее внимание к криптографическим методам обусловлено потребностями защиты корпоративных каналов, прокладываемых через публичную сеть Интернет, защитой Big Data, блокчейн и других технологий четвертой промышленной революции.

Рассмотрим **основные элементы криптосистемы**.

1. **Отправитель и получатель.** Отправитель посылает сообщение получателю. Более того, отправитель хочет послать свое сообщение безопасно: он должен быть уверен, что в случае перехвата это сообщение не будет прочитано.

2. **Сообщения и шифрование.** Само сообщение называется открытым текстом. Изменение вида сообщения с целью утаивания его сути называется шифрованием. Шифрованное сообщение называется шифротекстом. Процесс преобразования шифротекста в открытый текст называется дешифрированием. Эта последовательность показана на рис. 4.1.

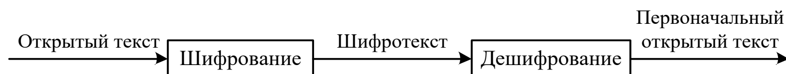


**Рис. 4.1.** Шифрование и расшифровка

Обозначим открытый текст как $M$ (от message, сообщение). Это может быть поток битов, текстовый файл, битовое изображение, оцифрованный звук, цифровое видеоизображение и др. Для компьютера $M$ — это двоичные данные.

Обозначим шифротекст как $C$ (от ciphertext). Это двоичные данные того же размера, что и $M$, иногда больше (если шифрование сопровождается сжатием, $C$ может быть меньше, чем $M$, однако само шифрование не обеспечивает сжатие информации). Функция шифрования $E$ действует на $M$, создавая $C$. В математической записи это выглядит так:

$$E(M) = C.$$

В обратном процессе функция дешифрирования $D$ действует на $C$, восстанавливая $M$:

$$D(C) = M.$$

Поскольку смыслом шифрования и последующего дешифрирования сообщения является восстановление первоначального открытого текста, должно выполняться следующее равенство:

$$D(E(M)) = M.$$

Кроме обеспечения конфиденциальности, криптография часто используется для других функций:

1) *аутентификация.* Получатель сообщения может проверить его источник, злоумышленник не сможет замаскироваться под кого-либо;

2) *целостность.* Получатель сообщения может проверить, не было ли сообщение изменено в процессе доставки, злоумышленник не сможет подменить правильное сообщение ложным;

3) *неотрицание авторства.* Отправитель не сможет ложно отрицать отправку сообщения.

Существуют жизненно важные требования к общению при помощи компьютеров так же, как существуют аналогичные требования при общении лицом к лицу. Эти функции обеспечивают уверенность в том, что личность является тем, за кого себя выдает, и документы, предъявляемые ей, настоящие.

3. **Алгоритмы и ключи.** Криптографический алгоритм, также называемый шифром, представляет собой математическую функцию, используемую для шифрования и дешифрирования. Обычно это две связанные функции: одна для шифрования, а другая для дешифрирования.

Ограниченным называется алгоритм, безопасность которого основана на сохранении самого алгоритма в тайне. Ограниченные алгоритмы представляют только исторический интерес, но они совершенно не соответствуют сегодняшним стандартам. Большая или изменяющаяся группа пользователей не может использовать такие алгоритмы, так как всякий раз, когда пользователь покидает группу, ее члены должны переходить на другой алгоритм. Алгоритм должен быть заменен и в том случае, если кто-нибудь извне случайно узнает секрет.

Также ограниченные алгоритмы не допускают качественного контроля или стандартизации. У каждой группы пользователей должен быть свой уникальный алгоритм. Такие группы не могут использовать открытые аппаратные или программные продукты, так как злоумышленник может купить такой же продукт и раскрыть алгоритм. Из-за этого необходимо разрабатывать и реализовывать собственные алгоритмы.

Несмотря на эти недостатки ограниченные алгоритмы необычайно популярны для приложений с низким уровнем безопасности. Пользователи либо не понимают проблем, связанных с безопасностью своих систем, либо не заботятся о них.

Современная криптография решает эти проблемы с помощью ключа. Множество возможных ключей называют пространством ключей. И шифрование, и дешифрирование зависят от ключа, что обозначается индексом $k$. Обозначим эти функции так:

$$E_k(M) = C,$$

$$D_k(C) = M.$$

При этом выполняется следующее равенство:

$$D_k(E_k(M)) = M.$$

Для некоторых алгоритмов при шифровании и дешифрировании используются различные ключи. То есть ключ шифрования $k_1$, отличается от соответствующего ключа дешифрирования $k_2$. В этом случае:

$$E_{k_1}(M) = C,$$

$$D_{k_2}(C) = M,$$

$$D_{k_2}(E_{k_1}(M)) = M.$$

Безопасность этих алгоритмов полностью основана на ключах, а не на деталях алгоритмов. Это значит, что алгоритм может быть опубликован и проанализирован и продукты, использующие этот алгоритм, могут широко тиражироваться. Даже в случае знания злоумышленником вашего алгоритма, он не сможет прочесть сообщения, так как ему неизвестен конкретный ключ.

Криптосистема представляет собой алгоритм и все возможные открытые тексты, шифротексты и ключи. Существует два основных типа алгоритмов, основанных на ключах: симметричные и с открытым ключом.

Симметричные алгоритмы, иногда называемые условными, представляют собой алгоритмы, в которых ключ шифрования может быть рассчитан по ключу дешифрирования и наоборот.

В большинстве симметричных алгоритмов ключи шифрования и дешифрирования одни и те же. Эти алгоритмы, также называемые алгоритмами с секретным ключом или алгоритмами с одним ключом, требуют, чтобы отправитель и получатель согласовали используемый ключ перед началом безопасной передачи сообщений.

Безопасность симметричного алгоритма определяется ключом, раскрытие которого означает, что любой человек сможет шифровать и дешифрировать сообщения. Пока передаваемые сообщения должны быть тайными, ключ должен храниться в секрете.

Шифрование и дешифрирование с использованием симметричного алгоритма обозначается как:

$$E_k(M) = C,$$

$$D_k(C) = M.$$

Симметричные алгоритмы делятся на две категории.

1. Потоковые алгоритмы, или потоковые шифры, которые обрабатывают открытый текст побитно (иногда побайтно).

2. Алгоритмы, работающие с группами битов открытого текста.

Группы битов называются блоками, а алгоритмы — блочными алгоритмами или блочными шифрами. Для алгоритмов, используемых в компьютерных модемах, типичный размер блока составляет 64 бита — достаточно большое значение, чтобы помешать анализу, и достаточно небольшое и удобное для работы. Такой вариант может рассматриваться как потоковый алгоритм, обрабатывающий поток символов.

Алгоритмы с открытым ключом (асимметричные алгоритмы) разработаны таким образом, что ключ, используемый для шифрования, отличается от ключа дешифрирования. Более того, ключ дешифрирования не может быть (в течение разумного количества времени) рассчитан по ключу шифрования.

В алгоритмах с открытым ключом ключ шифрования может быть открытым: кто угодно может использовать ключ для шифрования сообщения, но только конкретный человек с соответствующим ключом дешифрирования сможет его расшифровать. В этих системах ключ шифрования часто называется открытым ключом, а ключ дешифрирования — закрытым. Закрытый ключ иногда называется секретным ключом. Шифрование с открытым ключом $k$ обозначается как:

$$E_k(M) = C.$$

Хотя открытый и закрытый ключи различны, дешифрирование с соответствующим закрытым ключом обозначается как:

$$D_k(C) = M.$$

Иногда сообщения шифруются закрытым ключом, а дешифрируются открытым, что используется для цифровой подписи.

Несмотря на возможную путаницу, эти операции обозначают:

$$E_k(M) = C,$$

$$D_k(C) = M.$$

## 4.2. Криптоаналитические атаки

Смысл криптографии — в сохранении открытого текста и (или) ключа в тайне от злоумышленников (взломщиков, соперников, врагов, перехватчиков). Предполагается, что злоумышленники полностью контролируют линии связи между отправителем и получателем.

Криптоанализ — это наука получения открытого текста без ключа. Успешно проведенный криптоанализ может раскрыть открытый текст или ключ. Он также может обнаружить слабые места в криптосистемах, что в конце концов приведет к предыдущему результату. Раскрытие ключа некриптологическими способами называется компрометацией. Попытка криптоанализа называется атакой.

Существует четыре основных типа криптоаналитической атаки. Для каждого из них предполагается обладание криптоаналитиком всей полнотой знаний об используемом алгоритме шифрования.

1. **Атака с использованием только шифротекста.** У криптоаналитика есть шифротексты нескольких сообщений, зашифрованных одним и тем же алгоритмом шифрования. Его задача состоит в раскрытии открытого текста как можно большего числа сообщений или, что лучше, получении ключа (ключей), использованного для шифрования сообщений, для дешифрирования других сообщений, зашифрованных теми же ключами.

Дано: $C_1 = E_k(P_1)$, $C_2 = E_k(P_2)$, …, $C_i = E_k(P_i)$.

Получить: либо $P_1$, $P_2$, …, $P_i$; $K$; либо алгоритм получения $P_{i+1}$ из $C_{i+1} = E_k(P_{i+1})$.

2. **Атака с использованием открытого текста.** У криптоаналитика есть доступ не только к шифротекстам нескольких сообщений, но и к открытому тексту этих сообщений. Его задача состоит в получении ключа (ключей), использованного для шифрования сообщений, для дешифрирования других сообщений, зашифрованных тем же ключом (ключами).

Дано: $P_1, C_1 = E_k(P_1), P_2, C_2 = E_k(P_2), \ldots, P_i, C_i = E_k(P_i)$.

Получить: либо $k$; либо алгоритм получения $P_{i+1}$ из $C_{i+1} = E_k(p_{i+1})$.

3. **Атака с использованием выбранного открытого текста.** У криптоаналитика есть не только доступ к шифротекстам и открытым текстам нескольких сообщений, но и возможность выбирать открытый текст для шифрования. Это предоставляет больше вариантов, чем атака с использованием открытого текста, так как криптоаналитик может выбирать шифруемые блоки открытого текста, что может дать больше информации о ключе. Его задача состоит в получении ключа (или ключей), использованного для шифрования сообщений, или алгоритма, позволяющего дешифрировать новые сообщения, зашифрованные тем же ключом (ключами).

Дано: $P_1, C_1 = E_k(P_1), P_2, C_2 = E_k(P_2), \ldots, P_i, C_i = E_k(P_i)$, где криптоаналитик может выбирать $P_1, P_2, \ldots, P_i$.

Получить: либо $k$; либо алгоритм получения $P_{i+1}$ из $C_{i+1} = E_k(P_{i+1})$.

4. **Адаптивная атака с использованием открытого текста.** Это частный случай атаки с использованием выбранного открытого текста. Криптоаналитик может не только выбирать шифруемый текст, но и строить свой последующий выбор на базе полученных результатов шифрования. При атаке с использованием выбранного открытого текста криптоаналитик может выбрать для шифрования только один большой блок открытого текста, при адаптивной атаке с использованием выбранного открытого текста он может выбрать меньший блок открытого текста, затем выбрать следующий блок, используя результаты первого выбора и т. д.

5. **Атака с использованием выбранного шифротекста.** Криптоаналитик может выбрать различные шифротексты для дешифрирования и имеет доступ к дешифрированным открытым текстам. Например, у криптоаналитика есть доступ к черному

ящику, который выполняет автоматическое дешифрирование. Его задача состоит в получении ключа.

Дано: $C_1, P_1 = D_k(C_1)$, $C_2, P_2 = D_k(C_2)$, …, $C_i, P_i = D_k(C_i)$.

Получить: $k$.

Такой тип атаки обычно применим к алгоритмам с открытым ключом. Атака с использованием выбранного шифротекста иногда также эффективно против симметричных алгоритмов. Иногда атака с использованием выбранного открытого текста и атака с использованием выбранного шифротекста вместе называются атакой с использованием выбранного текста.

6. **Атака с использованием выбранного ключа.** Такой тип атаки означает не то, что криптоаналитик может выбирать ключ, а что у него есть некоторая информация о связи между различными ключами.

7. **Бандитский криптоанализ.** Криптоаналитик угрожает или шантажирует кого-нибудь, пока не получит ключ. Взяточничество иногда называется атакой с покупкой ключа. Это очень мощные способы атаки, часто являющиеся наилучшим путем взломать алгоритм.

Алгоритмы атак делятся по следующим категориям, приведенным в порядке убывания значимости:

1) *полный взлом.* Криптоаналитик получил ключ $k$, такой, что $D_k(C) = P$;

2) *глобальная дедукция.* Криптоаналитик получил альтернативный алгоритм $A$, эквивалентный $D_k(C)$ без знания $k$;

3) *местная (или локальная) дедукция.* Криптоаналитик получил открытый текст для перехваченного шифротекста;

4) *информационная дедукция.* Криптоаналитик получил некоторую информацию о ключе или открытом тексте. Такой информацией могут быть несколько бит ключа, сведения о форме открытого текста и т. д.

## 4.3. Безопасность алгоритмов

Различные алгоритмы предоставляют разные степени безопасности в зависимости от того, насколько трудно их взломать. Чтобы быть в безопасности, стоимость взлома алгоритма должна

быть выше стоимости зашифрованных данных; время взлома алгоритма больше времени, в течение которого зашифрованные данные должны сохраняться в секрете; объем данных, зашифрованных одним ключом, меньше объема данных, необходимого для взлома алгоритма.

Однако невозможно всегда находиться в безопасности, так как существует вероятность новых прорывов в криптоанализе, также значимость большинства данных со временем падает. Важно, чтобы значимость данных всегда оставалась меньше, чем стоимость взлома системы безопасности, защищающей данные.

Алгоритм является безусловно безопасным, если независимо от объема шифротекстов у криптоаналитика информации для получения открытого текста недостаточно. По сути только шифрование одноразовыми блокнотами невозможно атаковать при бесконечных ресурсах. Все остальные криптосистемы подвержены атаке с использованием только шифротекста простым перебором возможных ключей и проверкой осмысленности полученного открытого текста.

Криптография больше интересуется криптосистемами, которые тяжело взломать вычислительным способом. Алгоритм считается вычислительно безопасным (или, как иногда называют, сильным), если он не может быть взломан с использованием доступных ресурсов сейчас или в будущем. Сложность можно измерить различными способами:

1) сложность в объемах данных, используемых на входе операции вскрытия;

2) сложность обработки заключается во времени, необходимом для проведения атаки, часто называется коэффициентом работы;

3) требования к памяти. Объем памяти, необходимый для атаки.

В качестве эмпирического метода сложность определяется по максимальному из перечисленных трех коэффициентов. Ряд операций атаки предполагает взаимосвязь коэффициентов: более быстрая атака возможна за счет увеличения требований к памяти.

Сложность выражается порядком величины. Если сложность обработки для алгоритма составляет $2^{128}$, то $2^{128}$ операций требуется для вскрытия алгоритма. Эти операции могут быть

сложными и длительными. Так, если предполагается, что вычислительные мощности способны выполнять миллион операций в секунду, и для решения задачи используется миллион параллельных процессоров, получение ключа займет свыше $10^{19}$ лет, что в миллиард раз превышает время существования Вселенной.

В то время, как сложность атаки остается постоянной, мощь компьютеров растет. За последние полвека вычислительные мощности феноменально выросли, и нет никаких причин подозревать, что эта тенденция не будет продолжена. Многие криптографические взломы пригодны для параллельных компьютеров: задача разбивается на миллиарды маленьких кусочков, решение которых не требует межпроцессорного взаимодействия. Объявление алгоритма безопасным просто потому, что его нелегко взломать, используя современную технику, в лучшем случае ненадежно. Хорошие криптосистемы проектируются устойчивыми к взлому с учетом развития вычислительных средств на много лет вперед.

Стеганография служит для передачи секретов в сообщениях так, что скрыто само существование секрета. Как правило, отправитель пишет какое-нибудь неприметное сообщение, а затем прячет секретное на том же листе бумаги. Исторические приемы включают невидимые чернила; невидимые простому глазу пометки; плохо заметные отличия в написании букв; пометки карандашом машинописных символов; решетки, покрывающие большую часть сообщения, кроме нескольких символов и т. п.

Ближе к нашему времени люди начали прятать секреты в графических изображениях, заменяя младший значащий бит изображения битом сообщения. Графическое изображение при этом менялось совсем незаметно — большинство графических стандартов определяет больше цветовых градаций, чем способен различить человеческий глаз — и сообщение извлекалось на противоположном конце. Так, в черно-белой картинке 1024×1024 пиксела можно спрятать сообщение в 64 Кбайт. Многие общедоступные программы могут проделывать подобный фокус.

До появления компьютеров криптография состояла из алгоритмов на символьной основе. Различные криптографические алгоритмы либо заменяли одни символы другими, либо переставляли их. Лучшие алгоритмы делали и то, и другое несколько раз.

Сегодня все значительно сложнее, но философия остается прежней. Первое изменение заключается в том, что алгоритмы стали работать с битами, а не символами. Это важно с точки зрения размера алфавита: с 32 элементов до двух. Большинство хороших криптографических алгоритмов до сих пор комбинирует подстановки и перестановки.

Подстановочным шифром называется шифр, который каждый символ открытого текста в шифротексте заменяет другим символом. Получатель инвертирует подстановку шифротекста, восстанавливая открытый текст.

**ROTI3** — это простая шифровальная программа, обычно поставляемая с системами UNIX. Она также является простым подстановочным шифром. В этом шифре `A` заменяется на `N`, `B` — на `O` и т. д. Каждая буква смещается на 13 мест. Шифрование файла программой ROTI3 дважды восстанавливает первоначальный файл.

Простые подстановочные шифры легко раскрываются, так как они не прячут частоты использования различных символов в открытом тексте. Чтобы восстановить открытый текст, хорошему криптоаналитику требуется только знать все символы алфавита.

Шифр с бегущим ключом (иногда называемый книжным шифром), использующий один текст для шифрования другого текста, представляет собой другой пример подобного шифра. И хотя период этого шифра равен длине текста, он также может быть легко взломан.

В перестановочном шифре меняется не открытый текст, а порядок символов. В простом столбцовом перестановочном шифре открытый текст пишется горизонтально на разграфленном листе бумаги фиксированной ширины, а шифротекст считывается по вертикали. Дешифрирование представляет собой запись шифротекста вертикально на листе разграфленной бумаги фиксированной ширины и затем считывание открытого текста горизонтально.

Так как символы шифротекста совпадают с шифрами в открытом тексте, частотный анализ шифротекста показывает, что каждая буква встречается приблизительно с той же частотой,

что и обычно. Это даст криптоаналитику возможность применить различные методы, определяя правильный порядок символов для получения открытого текста. Применение к шифротексту второго перестановочного фильтра значительно повышает безопасность. Существуют и еще более сложные перестановочные фильтры, но компьютеры могут раскрыть почти все из них.

Хотя многие современные алгоритмы используют перестановку, с этим связана проблема использования большого объема памяти, также иногда требуется работа с сообщениями определенного размера.

XOR представляет собой операцию «исключающее или» (^) в языке C или $\oplus$ в математической нотации. Это обычная операция над битами:

$0 \oplus 0 = 0,$

$0 \oplus 1 = 1,$

$1 \oplus 0 = 1,$

$1 \oplus 1 = 0.$

Также заметим, что:

$a \oplus a = 0,$

$a \oplus b \oplus b = a.$

Запутанный алгоритм простого XOR является полиалфавитным шифром Виженера. Здесь он упоминается из-за распространенности в коммерческих программных продуктах, а именно в мире MS и Macintosh. К сожалению, если программу компьютерной безопасности заявляют как «патентованный» алгоритм шифрования, значительно более быстрый, чем DES, то в ней скорее всего используется вариант симметричного алгоритма. Открытый текст подвергается операции «исключающее или» вместе с ключевым текстом для получения шифротекста. Так как повторное применение операции XOR восстанавливает оригинал для шифрования и дешифрирования, используется одна и та же программа:

$P \oplus K = C,$

$C \oplus K = P.$

О настоящей безопасности здесь не может идти речи, так как этот тип шифрования легко вскрывается: его взлом на компьютере занимает несколько секунд.

Предположим, что открытый текст использует английский язык. Более того, пусть длина ключа составляет любое небольшое число байт. Разберем, как взломать этот шифр.

1. Определим длину ключа с помощью процедуры, известной как подсчет совпадений. Применим операцию XOR к шифротексту, используя в качестве ключа сам шифротекст с различными смещениями, и подсчитаем совпадающие байты. Если величина смещения кратна длине ключа, то совпадет свыше 6 % байтов. Если нет, будет совпадать меньше, чем 0,4 % (считая, что обычный ASCII-текст кодируется случайным ключом, для других типов открытых текстов числа будут другими). Это называется показателем совпадений. Минимальное смещение от одного значения, кратного длине ключа, к другому и является длиной ключа.

2. Сместим шифротекст на эту длину и проведем операцию XOR для смещенного и оригинального шифротекстов. Результатом операции будет удаление ключа и получение открытого текста, подвергнутого операции XOR с самим собой, смещенным на длину ключа. Так как в английском языке на 1 байт приходится 1,3 бита действительной информации, существующая значительная избыточность позволяет определить способ шифрования.

*Одноразовые блокноты.* Идеальный способ шифрования называется одноразовым блокнотом (разработан в 1917 г.). В классическом понимании одноразовый блокнот является большой неповторяющейся последовательностью символов ключа, распределенных случайным образом, написанных на кусочках бумаги и приклеенных к листу блокнота. Первоначально это была одноразовая лента для телетайпов. Отправитель использовал каждый символ ключа блокнота для шифрования только одного символа открытого текста. Шифрование представляет собой сложение по модулю 26 символов открытого текста и символа ключа из одноразового блокнота. Каждый символ ключа используется единожды и для одного сообщения. Отправитель шифрует сообщения и уничтожает использованные страницы блокнота или использованную часть ленты. Получатель, используя точно такой же блокнот, дешифрирует каждый символ шифротекста. Расшифровав сообщение, он уничтожает соответствующие страницы блокнота или часть ленты. Новое сообщение — это новые символы ключа. Например, если сообщением является `ONETIMEPAD`,

а ключевая последовательность в блокноте TBFRGFARFM, то шифротекст будет выглядеть как IPKLPSFHGQ, так как O + T mod 26 = I; N + B mod 26 = P; E + F mod 26 = K и т. д.

Эта схема совершенно безопасна, так как хакер не сможет получить доступ к одноразовому блокноту, использованному для шифрования сообщения. Это шифрованное сообщение на вид соответствует любому открытому сообщению того же размера.

Так как все ключевые последовательности совершенно одинаковы (символы ключа генерируются случайным образом), у противника отсутствует информация, позволяющая подвергнуть шифротекст криптоанализу. Кусочек шифротекста может быть похож на POYYAEAAZX, что дешифрируется как SALMONEGGS, или на BXEGBMTMXM, что дешифрируется как GREENFLUID.

В связи с тем, что все открытые тексты равновероятны, у криптоаналитика нет возможности определить, какой из открытых текстов является правильным. Случайная ключевая последовательность, сложенная с неслучайным открытым текстом, дает совершенно случайный шифротекст, и никакие вычислительные мощности не смогут это изменить.

Необходимо помнить, что символы ключа должны генерироваться случайным образом. Любые попытки вскрыть такую схему сталкиваются со способом, которым создается последовательность символов ключа. Использование генераторов псевдослучайных чисел не считается: у них всегда неслучайные свойства.

Ключевую последовательность никогда нельзя использовать второй раз. Даже если используется блокнот размером в несколько гигабайт, криптоаналитик получит несколько текстов с перекрывающимися ключами и сможет восстановить открытый текст. Он сдвинет каждую пару шифротекстов относительно друг друга и подсчитает число совпадений в каждой позиции. Если шифротексты смещены правильно, соотношение совпадений резко возрастет — точное значение зависит от языка открытого текста. С этой точки зрения криптоанализ не представляет труда. Это похоже на показатель совпадений, но сравниваются два различных периода.

Идея одноразового блокнота легко расширяется на двоичные данные. Вместо одноразового блокнота, состоящего из букв,

используется одноразовый блокнот из битов. Вместо сложения открытого текста с ключом одноразового блокнота используется XOR. Для дешифрирования XOR применяется к шифротексту с тем же одноразовым блокнотом. Все остальное не меняется, безопасность остается такой же совершенной.

Несмотря на это, существует несколько проблем. В связи с тем, что ключевые биты должны быть случайными и не могут использоваться снова, длина ключевой последовательности должна равняться длине сообщения. Одноразовый блокнот удобен для нескольких небольших сообщений, но его нельзя использовать для работы по каналу связи с пропускной способностью 1,5 Мбит/с. Можно хранить 650 Мбайт случайных данных на флешке, но возникнут проблемы. Во-первых, флешки экономичны только при больших тиражах. И, во-вторых, необходимо уничтожать использованные биты. Для флеш-накопителя нет другой возможности удалить информацию, кроме как физически ее разрушить (плохое стирание сохраняет информацию).

Даже если проблемы распределения и хранения ключей решены, необходимо точно синхронизировать работу отправителя и получателя. Если получатель пропустит бит (или несколько бит пропадут при передаче), сообщение потеряет всякий смысл. С другой стороны, если несколько бит изменятся при передаче (и ни один не будет удален или добавлен, что гораздо больше похоже на влияние случайного шума), то лишь эти биты будут расшифрованы неправильно. Но одноразовый блокнот не обеспечивает проверку подлинности.

Одноразовые блокноты используются и сегодня, главным образом для сверхсекретных каналов связи с низкой пропускной способностью. По слухам, горячая линия между Соединенными Штатами Америки и Россией шифруется с помощью одноразового блокнота.

Также существует множество компьютерных алгоритмов. Следующие три используются чаще всего.

1. DES (data encryption standard, стандарт шифрования данных) — самый популярный компьютерный алгоритм шифрования, является американским и международным стандартом. Это симметричный алгоритм, при котором один и тот же ключ используется для шифрования и дешифрирования.

2. RSA (назван в честь создателей — Ривеста (Rivest), Шамира (Shamir) и Эдлмана (Adleman)) — самый популярный алгоритм с открытым ключом. Используется и для шифрования, и для цифровой подписи.

3. DSA (digital signature algorithm, алгоритм цифровой подписи, используется как часть стандарта цифровой подписи) — алгоритм с открытым ключом, использующийся только для цифровой подписи. Он не может быть использован для шифрования.

## Выводы по главе 4

В телекоммуникационных системах информация по каналам связи передается в зашифрованном виде в интересах информационной безопасности. Следовательно, актуальной является прикладная криптография.

Сегодня понятие криптографии значительно расширилось и включает в себя аутентификацию, цифровые подписи и множество других элементарных функций безопасности.

Возросшее внимание к криптографическим методам обусловлено потребностями защиты корпоративных каналов, прокладываемых через Интернет, защитой Big Data, блокчейн и других технологий четвертой промышленной революции.

Рассмотрены базовые понятия — отправитель и получатель. Показаны их интересы в безопасных коммуникациях, проблемы которых решаются шифрованием передаваемых сообщений.

Определены математические модели шифрования и последующего дешифрирования сообщения, их преимущества и недостатки, переход к криптосистемам, в которых существует два основных типа алгоритмов, основанных на ключах: симметричные и с открытым ключом.

Применительно к четырем основным типам криптоаналитических атак сформулированы алгоритмы информационной безопасности и применение подстановочных и перестановочных шифров. Рассмотрен алгоритм одностраничного блокнота, не поддающийся взлому.

# Вопросы для самоконтроля

1. Каковы роль и значение криптографии для информационной безопасности в телекоммуникационных системах?
2. В чем отличия и сходства отправителя и получателя?
3. Что такое дешифрование и общая схема функционирования?
4. Назовите математические модели функций шифрования и дешифрования.
5. Что представляют алгоритмы и ключ в криптографическом алгоритме?
6. Каковы симметричные алгоритмы, их возможности, достоинства и недостатки?
7. Каковы алгоритмы с открытым ключом, их возможности, достоинства и недостатки?
8. Что подразумевает атака с использованием только шифротекста?
9. Что подразумевает атака с использованием открытого текста?
10. Что подразумевает атака с использованием выбранного открытого текста?
11. Что подразумевает адаптивная атака с использованием открытого текста?
12. Что подразумевает атака с использованием выбранного шифротекста?
13. Что подразумевает атака с использованием выбранного ключа?
14. Как определяется безопасность алгоритмов?
15. Перечислите алгоритмы атак.
16. Каковы подстановочные и перестановочные шифры?
17. Что такое простое XOR и где оно применяется?
18. Каковы одноразовые блокноты, их возможности и применение?

# Глава 5

# Криптографическая безопасность

## 5.1. Информационная и вычислительная безопасность

В криптографии безопасность определяется не так, как в общей информатике. Основное различие между безопасностью ПО и криптографической безопасностью заключается в том, что последняя поддается количественному измерению. В отличие от мира ПО, где приложение обычно считается либо безопасным, либо небезопасным, в мире криптографии часто можно оценить, сколько усилий необходимо приложить для взлома криптографического алгоритма. Кроме того, если основная цель безопасности ПО — помешать противнику использовать код программы для нанесения ущерба, то цель криптографической безопасности — сделать невозможным решение четко поставленной задачи.

В криптографии есть два определения понятия невозможного: информационная безопасность и вычислительная безопасность. Проще говоря, **информационная безопасность** — это теоретическая невозможность, а **вычислительная безопасность** — практическая невозможность. Для информационной безопасности не нужно количественно измерять безопасность, потому что шифр рассматривается как или безопасный, или небезопасный. Поэтому на практике это понятие бесполезно, хотя играет важную роль в теоретической криптографии. Вычислительная безопасность — более практически полезная мера стойкости шифра.

Информационная безопасность характеризует не трудность взлома шифра, а возможность взлома. Шифр считается информа-

ционно безопасным, только если его нельзя взломать даже при наличии неограниченного времени и памяти. Если успешная атака на шифр возможна, хотя и потребует триллион лет, такой шифр информационно небезопасен.

Например, одноразовый блокнот, описанный ранее, является безопасным. Он преобразует открытый текст $P$ в шифротекст $C = P \oplus K$, где $K$ — случайная битовая строка, уникальная для каждого открытого текста. Этот шифр информационно безопасен, потому что, имея шифротекст и неограниченное время для перебора всех возможных ключей $K$ и вычисления соответствующего открытого текста $P$, мы не смогли бы найти правильный $K$, так как возможных $P$ столько же, сколько $K$.

В отличие от информационной безопасности, при оценке вычислительной безопасности шифр считается безопасным, если его нельзя взломать за разумное время и при наличии ресурсов: памяти, оборудования, бюджета, энергии и т. д. Вычислительная безопасность — это способ количественного измерения безопасности шифра или вообще любого криптографического алгоритма.

Например, рассмотрим шифр **E**, для которого известна пара «открытый текст — шифротекст» ($P$, $C$), но неизвестен 128-битовый ключ $K$, с помощью которого вычислен $C = \mathbf{E}(K, P)$. Этот шифр не является информационно безопасным, потому что его можно было бы взломать, перебрав все $2^{128}$ возможных 128-битовых ключа $K$, пока не будет найден такой, для которого $\mathbf{E}(K, P) = C$. Но на практике, даже если мы будем проверять по 100 млрд ключей в секунду, для решения задачи потребуется более 100 квинтиллионов лет. Иными словами, разумно считать, что этот шифр вычислительно безопасен, потому что взломать его практически невозможно.

Вычислительную безопасность иногда характеризуют двумя величинами:

1) $t$ — предельное число операций, которое может выполнить противник;

2) ε (эпсилон) — предельная вероятность успешной атаки.

Говорят, что криптографическая схема является ($t$, ε)-безопасной, если, выполнив не более $t$ операций, противник не сможет добиться вероятности успеха более ε, где ε — число от 0

до 1. Вычислительная безопасность определяет предел трудности взлома криптографического алгоритма.

Важно отметить, что $t$ и ε — всего лишь предельные значения: если шифр является ($t$, ε)-безопасным, никакой противник, выполнивший менее $t$ операций, не сможет добиться успеха (с вероятностью ε), но это не значит, что противник, выполнивший ровно $t$ операций, добьется успеха. Также неизвестно сколько операций необходимо, а их число может быть гораздо больше $t$. Имеется ввиду, что $t$ — нижняя граница потребных вычислительных усилий, поскольку для компрометации системы необходимо по меньшей мере $t$ операций.

Иногда точно известно, сколько усилий нужно приложить для взлома шифра; в таких случаях говорят, что ($t$, ε)-безопасность дает точную границу, если существует атака, которая приводит к взлому шифра с вероятностью ε ровно за $t$ операций.

Например, рассмотрим симметричный шифр со 128-битовым ключом. В идеале этот шифр должен быть ($t$, $t/2^{128}$)-безопасным для любого значения $t$ между 1 и $2^{128}$. Наилучшей атакой должен быть полный перебор (испытание всех ключей, пока не будет найден правильный). Любая более эффективная атака должна была бы использовать какое-то упущение в шифре, поэтому мы стремимся создавать шифры, для которых полный перебор является наилучшей из возможных атак.

В предположении ($t$, $t/2^{128}$)-безопасности рассмотрим вероятность успеха трех возможных атак.

В первом случае $t = 1$, т. е. противник проверяет один ключ и достигает успеха с вероятностью $\varepsilon = 1/2^{128}$.

Во втором случае $t = 2^{128}$, т. е. противник проверяет все $2^{128}$ ключей, и один из них точно правилен. Таким образом, вероятность $\varepsilon = 1$ (если противник перепробует все ключи, то, очевидно, найдет правильный).

В третьем случае противник пробует только $t = 2^{64}$ ключей и добивается успеха с вероятностью $\varepsilon = 2^{64}/2^{128} = 2^{-64}$. Если перебирается лишь часть всего множества ключей, вероятность успеха пропорциональна количеству проверенных ключей. Мы можем заключить, что шифр с $n$-битовым ключом в лучшем случае является ($t$, $t/2^n$)-безопасным для любого $t$ от 1 до $2^n$, потому что вне зависимости от стойкости шифра атака полным перебором всегда

завершится успешно. Следовательно, ключ должен быть достаточно длинным, чтобы сделать атаки полным перебором практически неосуществимыми.

## 5.2. Количественное измерение безопасности

Обнаружив атаку, необходимо узнать, насколько она эффективна теоретически и осуществима ли она на практике. Аналогично для предположительно безопасного шифра нужно знать, насколько настойчивым усилиям он может противостоять. Чтобы ответить на эти вопросы, разберем, как можно измерить криптографическую безопасность в битах (теоретический взгляд на вещи) и какие факторы влияют на фактическую стоимость атаки.

### Измерение безопасности в битах

В контексте вычислительной безопасности шифр называется $t$-безопасным, если для успешной атаки необходимо по меньшей мере $t$ операций. Таким образом, мы избегаем интуитивно неочевидной нотации $(t, \varepsilon)$, предполагая, что вероятность успеха $\varepsilon$ близка к 1, ведь именно этот случай интересует нас на практике. Далее безопасность выражается в битах; выражение «$n$-битовая безопасность» означает, что для компрометации некоторого аспекта безопасности необходимо приблизительно $2^n$ операций.

Если примерно известно, сколько операций требуется для взлома шифра, его уровень безопасности в битах можно определить, вычислив двоичный логарифм числа операций: если требуется 1 000 000 операций, то уровень безопасности равен $\log_2(1\,000\,000)$, или около 20 бит ($1\,000\,000 \approx 2^{20}$). Напомним, что $n$-битовый ключ обеспечивает не более чем $n$-битовую безопасность, потому что атака с полным перебором всех $2^n$ возможных ключей всегда будет успешной. Но размер ключа не всегда равен уровню безопасности — это лишь верхняя граница, или максимально возможный уровень безопасности.

Уровень безопасности может быть меньше размера ключа по двум причинам:

1) для взлома шифра достаточно меньшего числа операций, чем ожидалось. Например, если для нахождения ключа нужно просмотреть не все $2^n$ вариантов, а только их часть;

2) уровень безопасности шифра намеренно сделан меньшим, чем размер ключа, как в большинстве алгоритмов с открытым ключом. Например, алгоритм RSA с 2048-битовым закрытым ключом обеспечивает безопасность меньше 100 бит.

Битовая безопасность полезна при сравнении уровней безопасности шифров, но не дает достаточно информации о фактической стоимости атаки. Иногда эта абстракция оказывается слишком простой, потому что предполагается, что для взлома шифра с уровнем безопасности $n$ бит хватает $2^n$ операций, но вид этих операций не конкретизируется. Поэтому реальные уровни безопасности двух шифров с одинаковой битовой безопасностью могут сильно различаться по реальной стоимости атаки для противника.

Предположим, имеется два шифра со 128-битовой безопасностью и 128-битовым ключом. Чтобы взломать их, необходимо вычислить шифр $2^{128}$ раз, но вычисление второго шифра занимает в 100 раз больше времени, чем первого. Тогда на $2^{128}$ вычислений второго шифра уйдет примерно столько же времени, сколько на $100 \times 2^{128} \approx 2^{134,64}$ вычислений первого. Если подсчитывать в терминах более быстрого первого шифра, то для взлома более медленного требуется $2^{134,64}$ операций, а если в терминах второго шифра — только $2^{128}$ операций. Правильно будет сказать, что второй шифр более стойкий, чем первый, но стократное различие в производительности широко используемых шифров встречается редко. Неточность определения операции создает много трудностей при сравнении эффективности атак. В описании некоторых атак утверждается, что им удалось снизить безопасность шифра, потому что выполняется $2^{120}$ вычислений некоторой операции, а не $2^{128}$ вычислений шифра, но при этом упускается из виду скорость каждого типа атак. Не всегда атака с $2^{120}$ операциями работает быстрее, чем атаки с полным перебором $2^{128}$ вариантов. Тем не менее битовая безопасность остается полезным понятием, если только операция занимает примерно столько же времени, сколько вычисление шифра. В конце концов, в реальной жизни уровень безопасности достаточно определить с точностью до порядка величины.

## Полная стоимость атаки

Битовая безопасность отражает стоимость самой быстрой атаки на шифр путем оценивания количества необходимых для успеха операций по порядку величины. Но на стоимость атаки влияют и другие факторы, которые нужно принимать во внимание, оценивая фактический уровень безопасности. Выделим четыре главных фактора: параллелизм, память, предварительные вычисления и количество мишеней.

**Параллелизм**

Прежде всего нужно учитывать вычислительный параллелизм. Рассмотрим, к примеру, две атаки, в каждой из которых $2^{56}$ операций. Разница между ними в том, что вторую атаку можно распараллелить, а первую — нет. В первой атаке нужно выполнить $2^{56}$ последовательно зависимых операций типа $x_{i+1} = f_i(x_i)$ для некоторого $x_0$ и некоторых функций $f_i$ (где $i$ изменяется от 1 до $2^{56}$), а во второй $2^{56}$ независимых операций типа $x_i = f_i(x)$ для некоторого $x$ и $i$ от 1 до $2^{56}$, которые могут производиться параллельно. Параллельная обработка может быть на несколько порядков быстрее последовательной. Например, если бы нам было доступно $2^{16}$ = 65 536 процессоров, то рабочую нагрузку параллельной атаки можно было бы разбить на $2^{16}$ независимых задач, каждая из которых выполняет $2^{56} / 2^{16} = 2^{40}$ операций. Но первая атака ничего не выигрывает от наличия нескольких ядер, потому что каждая операция зависит от результата предыдущей. Поэтому параллельная атака завершится в 65 536 раз быстрее последовательной, хотя количество выполненных операций одно и то же.

**Память**

Второй фактор, который следует учитывать при определении стоимости атаки, — располагаемая память. При оценивании криптоаналитических атак нужно принимать во внимание время и память: сколько операций они производят в единицу времени, сколько потребляют памяти, как используется эта память и каково быстродействие доступной памяти? Битовая безопасность отражает только время, потребное для выполнения атаки. Важно учитывать количество обращений к памяти, необходимых для совершения атаки, скорость обращения к памяти (помня, что скорости записи и чтения могут различаться), размер читаемых или запи-

сываемых данных, характер доступа (последовательный или произвольный) и организацию данных в памяти. Например, на одном из современных CPU общего назначения чтение из регистра занимает один такт, чтение из процессорного кеша (кеша L3) — примерно 20 тактов, а чтение из DRAM — по меньшей мере 100 тактов. Коэффициент 100 — это разница между одним днем и тремя месяцами.

**Предварительные вычисления**

Предварительные вычисления выполняются один раз, но их результатами можно пользоваться многократно при последующих атаках. Иногда это называется офлайновой стадией атаки.

Например, рассмотрим атаку с компромиссом между временем и памятью. В этом случае противник предварительно выполняет очень большое вычисление гигантских справочных таблиц, которые сохраняются и используются в ходе фактической атаки. Так, в одной атаке на шифрование в мобильной сети 2G пришлось потратить два месяца на построение таблиц общим объемом 2 Тбайта, которые затем использовались для взлома шифра и нахождения секретного сеансового ключа за несколько секунд.

**Количество мишеней**

Чем больше количество мишеней, тем шире поверхность атаки и тем больше информации противник может получить о ключах, за которыми охотится.

Например, рассмотрим поиск ключа полным перебором: если мы ищем всего один $n$-битовый ключ, то для гарантированного его нахождения потребуется $2^n$ попыток. Если мы ищем несколько $n$-битовых ключей, скажем $M$, и если для одного $P$ имеется $M$ различных шифр-текстов, где $C = \mathbf{E}(K, P)$ для каждого из $M$ искомых ключей ($K$), то для нахождения всех ключей понадобится $2^n$ попыток. Но если нас интересует хотя бы один из $M$ ключей, а не все сразу, то для успеха в среднем достаточно $2^n / M$ попыток.

Например, чтобы взломать один 128-битовый ключ из $2^{16} = 65\,536$ ключей-мишеней, в среднем нужно произвести $2^{128-16} = 2^{112}$ вычислений шифра. Следовательно, стоимость (и время проведения) атаки уменьшается при возрастании числа мишеней.

## Выбор и вычисление уровней безопасности

Выбор уровня безопасности часто сводится к выбору между 128-битовой и 256-битовой безопасностью, потому что большинство стандартных криптографических алгоритмов и их реализаций доступны для одного из этих двух уровней. По-прежнему существуют схемы с 64- или 80-битовой безопасностью, но в современном мире они уже не считаются достаточно безопасными. На верхнем уровне 128-битовая безопасность означает, что для взлома криптосистемы нужно выполнить приблизительно $2^{128}$ операций. Для понимания величины этого числа приведем в пример возраст Вселенной, равный примерно $2^{88}$ наносекундам (в одной секунде миллиард наносекунд). Поскольку при современных технологиях для проверки одного ключа нужно не менее одной наносекунды, атака займет в $2^{40}$ раз больше времени, чем существует Вселенная. При этом параллелизм и выбор нескольких мишеней не могут существенно уменьшить время атаки. Допустим, нас интересует взлом любой из миллиона мишеней, и в нашем распоряжении имеется миллион параллельных ядер. Тогда время поиска уменьшится с $2^{128}$ до $(2^{128} / 2^{20}) / 2^{20} = 2^{88}$, что эквивалентно одному времени жизни Вселенной.

При оценке уровня безопасности следует также помнить о развитии технологий. Закон Мура утверждает, что эффективность вычислений удваивается примерно каждые два года. Можно интерпретировать этот факт как потерю одного бита безопасности в два года: если сегодня бюджет в 1 000 долл. позволяет взломать 40-битовый ключ за час, то по закону Мура через два года при том же бюджете можно будет взломать 41-битовый ключ (намеренное упрощение). Экстраполируя, можно сказать, что через 80 лет безопасность станет на 40 бит меньше. Иными словами, через 80 лет выполнение $2^{128}$ операций может стоить столько же, сколько сегодня стоит выполнение $2^{88}$ операций. С учетом параллелизма и нескольких мишеней время вычислений уменьшается до $2^{48}$ наносекунд, или примерно трех дней. Но такая экстраполяция в высшей степени неточна, потому что закон Мура не может и не будет масштабироваться настолько прямолинейно. Тем не менее то, что кажется неосуществимым сегодня, вполне может стать реальным через 100 лет.

Иногда уровень безопасности ниже 128 бит вполне оправдан. Например, если данные должны оставаться в безопасности лишь короткое время, а затраты на реализацию более высокого уровня негативно отразятся на стоимости или удобстве работы с системой. Реальный пример — платное телевидение, в котором ключи шифрования имеют длину 48 или 64 бита, и такой уровень безопасности достаточен, потому что ключ обновляется каждые 5 или 10 секунд.

Тем не менее для обеспечения долговременной безопасности следует выбирать 256-битовый или чуть меньший уровень. Даже в худшем случае — при появлении квантовых компьютеров — схема с 256-битовой безопасностью вряд ли будет взломана в обозримом будущем. В более 256 бит практически нет надобности, это всего лишь маркетинговый ход.

## 5.3. Достижение безопасности

После того как уровень безопасности выбран, важно, чтобы криптографическая схема отвечала ему. Для обретения уверенности в безопасности криптографического алгоритма можно опираться на математические доказательства — это называется доказуемой безопасностью, или на свидетельства, основанные на безуспешных попытках взломать алгоритм, — эвристическая или предположительная безопасность. Эти два подхода дополняют друг друга, ни один из них не является более предпочтительным.

### Доказуемая безопасность

Существует возможность строго доказать, что взломать криптографическую схему так же трудно, как решить другую заведомо трудную задачу. Такое доказательство безопасности гарантирует, что схема останется безопасной до тех пор, пока трудная задача остается трудной. Доказательство такого типа называется сведением, оно уходит корнями в теорию сложности: взлом некоторого шифра сводится к задаче $X$, если любой метод решения $X$ дает также метод взлома шифра. Доказательства безопасности бывают двух видов в зависимости от характера предположительно трудной задачи: относительно математической задачи и относительно криптографической задачи.

**Доказательства относительно математической задачи**

Многие доказательства безопасности (например, для криптографии с открытым ключом) показывают, что взлом криптографической схемы по меньшей мере так же труден, как решение некоторой трудной математической задачи. Речь идет о задачах, для которых решение заведомо существует и легко проверить, что предъявленное решение действительно является таковым, но найти решение трудно с вычислительной точки зрения.

Как пример рассмотрим решение задачи факторизации, одной из самых известных математических задач в криптографии: дано число, про которое известно, что оно является произведением двух простых чисел ($n = pq$), требуется найти эти числа. Например, если $n = 15$, то множителями будут 3 и 5. Для небольших чисел это легко, но с ростом числа трудность задачи экспоненциально возрастает. Например, для числа $n$ длиной 3 000 бит (около 900 десятичных цифр) или более задача факторизации считается практически неразрешимой.

Одной из самых известных схем, опирающихся на задачу факторизации, является RSA. В этом случае открытый текст $P$, рассматриваемый как большое число, шифруется путем вычисления $C = P_e \bmod n$, где число $e$ и $n = pq$ составляют открытый ключ. При дешифрировании открытый текст восстанавливается по шифротексту путем вычисления $P = C^d \bmod n$, где $d$ — закрытый ключ, ассоциированный с $e$ и $n$. Если мы сможем факторизовать $n$, то взломаем схему RSA, поскольку сможем найти закрытый ключ, зная открытый. И наоборот, зная закрытый ключ, мы сможем факторизовать $n$. Таким образом, нахождение закрытого ключа RSA и факторизация $n$ — эквивалентные трудные задачи. Именно такого рода сведение интересует нас при доказательстве безопасности. Однако нет никакой гарантии, что реконструкция открытого текста в RSA — задача столь же трудная, как факторизация $n$, поскольку знания открытого текста недостаточно для нахождения закрытого ключа.

**Доказательства относительно другой криптографической задачи**

Вместо сравнения криптографической схемы с математической задачей, можно сравнить ее с другой криптографической схемой и доказать, что вторую можно взломать, только если воз-

можно взломать первую. Доказательства безопасности симметричных шифров обычно строятся именно таким образом.

Например, имея всего один перестановочный алгоритм, можно построить симметричные шифры, генераторы случайных битов и другие криптографические объекты, например, функции хеширования. Для этого нужно только сочетать перестановки с различными типами входных данных. Тогда доказательства показывают, что все созданные схемы безопасны, если безопасен перестановочный алгоритм. Такие доказательства обычно строятся путем конструирования атаки на меньший компонент, если известна атака на больший, т. е. посредством сведения.

При доказательстве того, что один криптоалгоритм не слабее другого, главным преимуществом является уменьшенная поверхность атаки: вместо анализа базового алгоритма и комбинации можно рассмотреть базовый алгоритм нового шифра. Разрабатывая шифр, в котором используется новый перестановочный алгоритм и новая комбинация, можно доказать, что комбинация не уменьшает безопасность по сравнению с базовым алгоритмом, поэтому чтобы взломать комбинацию, нужно взломать новый перестановочный алгоритм.

## Возможные проблемы

Криптографы часто прибегают к доказательствам безопасности — как относительно математических схем, так и относительно других криптографических схем. Но существование доказательства безопасности еще не гарантирует идеальность криптографической схемы и не может служить основанием для пренебрежения более практическими аспектами реализации. Как считают специалисты, то, что доказуемо безопасно, вероятно, таковым не является. Имеется в виду, что доказательство безопасности не следует воспринимать как абсолютную гарантию безопасности. Существует несколько причин, из-за которых доказуемо безопасная схема может приводить к утрате безопасности.

Одна из таких причин кроется в самой фразе «доказательство безопасности». В математике доказательством считается демонстрация абсолютной истины, тогда как в криптографии доказательство демонстрирует лишь относительную истину. Например, доказательство того, что взломать шифр так же трудно, как

вычислить дискретный логарифм — найти число $x$, если известны $g$ и $g^x \bmod n$, — гарантирует, что если шифр взламывается, то взламывается и много других шифров.

Еще один подводный камень заключается в том, что безопасность обычно доказывается в отношении какого-то одного аспекта. Например, можно было бы доказать, что найти закрытый ключ шифра так же трудно, как решить задачу факторизации. Но если можно восстановить открытый текст по шифротексту, не зная ключа, то это доказательство бесполезно, поскольку находить ключ не нужно.

Кроме того, доказательства не всегда корректны, и может оказаться, что взломать алгоритм проще, чем предполагалось.

Иногда трудная математическая задача решается гораздо легче. Например, при некоторых слабых параметрах взломать алгоритм RSA нетрудно. Так часто бывает, когда задача новая и еще плохо изучена.

Даже если доказательство безопасности алгоритма безупречно, его реализация может оказаться криптографически нестойкой. Например, воспользовавшись утечкой информации по побочному каналу, противник может что-то узнать о внутреннем устройстве алгоритма и взломать его, обойдя доказательство. Имеется также опасность неправильного применения криптографической схемы: если алгоритм слишком сложен и имеет много конфигурационных параметров, велики шансы, что пользователь или разработчик сконфигурирует его неправильно, что может привести к полной утрате безопасности.

## 5.4. Пути повышения криптобезопасности

### Эвристическая безопасность

Доказуемая безопасность — прекрасный способ получить уверенность в качестве криптографической схемы, но не для всех видов алгоритмов такие доказательства существуют. На самом деле для большинства симметричных шифров нет доказательства безопасности.

Например, мы ежедневно полагаемся на алгоритм Advanced encryption standard (AES), чтобы безопасно шифровать дан-

ные, которыми обмениваются мобильные телефоны, ноутбуки и настольные компьютеры, но AES не является доказуемо безопасным; никто не доказал, что его так же трудно взломать, как решить какую-то хорошо известную задачу. AES не сводится к математической задаче или к другому алгоритму, потому что сам является трудной задачей.

В тех случаях, когда доказать безопасность невозможно, остается только одна причина доверять шифру — многие компетентные специалисты пытались взломать его и потерпели неудачу. Иногда это называют эвристической безопасностью.

Нельзя быть точно уверенным, что шифр безопасен, но есть некоторая уверенность, что алгоритм не будет взломан, если сотни опытных криптоаналитиков потратили сотни часов, пытаясь взломать его, и опубликовали полученные результаты. Обычно атаки организуются на упрощенные версии шифра (с меньшим числом операций или меньшим числом раундов — коротких последовательностей операций, которые повторяются для лучшего перемешивания битов).

В ходе анализа нового шифра криптоаналитики сначала пытаются взломать один раунд, затем два, три — столько, сколько смогут. Запасом безопасности называется разность между общим числом раундов и числом успешно атакованных раундов. Если спустя несколько лет изучения запас безопасности все еще велик, то появляется уверенность, что шифр, вероятно, безопасен.

### Генерирование ключей

Если планируется что-то зашифровать, необходимо сгенерировать временные «сеансовые» ключи (наподобие тех, что генерируются при посещении HTTPS-сайта) или открытые на длительный срок. Напомним, что секретные ключи лежат в основе криптографической безопасности и должны генерироваться случайным образом, чтобы их нельзя было предсказать.

Например, при заходе на HTTPS-сайт ваш браузер получает открытый ключ сайта и с его помощью генерирует симметричный ключ, действующий только на время текущего сеанса, тогда как открытый ключ сайта и ассоциированный с ним закрытый ключ могут действовать на протяжении многих лет. Поэтому было бы лучше, если бы противник не смог узнать закрытый ключ, но ге-

нерирование секретного ключа не всегда сводится к простому получению достаточного числа псевдослучайных битов.

Криптографические ключи могут генерироваться одним из трех способов:

1) случайно, с помощью генератора псевдослучайных чисел (PRNG) и при необходимости алгоритма генерирования ключей;

2) из пароля, с помощью функции формирования ключа (KDF), которая преобразует указанный пользователем пароль в ключ;

3) посредством протокола совместной выработки ключа, состоящего из серии обмена сообщениями между двумя или более сторонами, которая завершается выработкой общего ключа.

**Генерирование симметричных ключей**

Симметричный ключ — это секретный ключ, известный обеим сторонам. Такие ключи генерировать проще всего. Обычно они имеют такую же длину, как обеспечиваемый ими уровень безопасности: 128-битовый ключ обеспечивает 128-битовую безопасность, и любой из $2^{128}$ возможных ключей ничем не хуже любого другого.

Чтобы сгенерировать симметричный $n$-битовый ключ с помощью криптографически стойкого PRNG, нужно запросить $n$ псевдослучайных бит и использовать их в качестве ключа. Например, для генерирования случайного симметричного ключа можно воспользоваться пакетом OpenSSL. Следующая команда выводит псевдослучайные биты.

**Генерирование асимметричных ключей**

Асимметричные ключи обычно длиннее обеспечиваемого ими уровня безопасности, но их труднее генерировать, потому что недостаточно получить $n$ бит от PRNG. Асимметричный ключ — это не просто последовательность битов, а объект определенного типа, например большое число, обладающее нужными свойствами (в RSA это произведение двух простых чисел). Маловероятно, что случайная битовая строка будет обладать свойствами, необходимыми, чтобы считаться допустимым ключом.

Для генерирования асимметричного ключа псевдослучайные биты нужно подать на вход алгоритма генерирования ключа. По этому начальному значению, длина которого должна быть не меньше требуемого уровня безопасности, алгоритм строит за-

крытый ключ и соответствующий ему открытый ключ так, чтобы эта пара удовлетворяла всем обязательным критериям. Например, наивный алгоритм генерирования ключей для RSA сгенерировал бы число $n = pq$, воспользовавшись алгоритмом нахождения двух простых чисел примерно одинаковой длины. Этот алгоритм пробует случайные числа, пока какое-нибудь не окажется простым, поэтому понадобится алгоритм для проверки простоты числа.

Чтобы не заниматься самостоятельной реализацией алгоритма генерирования ключей, можно воспользоваться пакетом OpenSSL и сгенерировать, например, 4096-битовый закрытый ключ RSA.

## Защита ключей

Ключ необходимо хранить в тайне, но так, чтобы он был доступен в любой момент, когда понадобится. Есть три способа решения этой проблемы.

**Обертывание ключа (шифрование ключа другим ключом)**

Сложность заключается в том, что второй ключ должен быть доступен, когда понадобится дешифрировать защищенный ключ. На практике второй ключ часто генерируется из пароля, указанного пользователем в момент, когда ему нужно использовать защищенный ключ. Именно так закрытые ключи обычно защищаются в протоколе Secure Shell (SSH).

**Динамическое генерирование из пароля**

В этом случае никакой зашифрованный файл хранить не нужно, потому что ключ генерируется непосредственно из пароля. Этот метод используется в современных системах типа miniLock. Несмотря на сравнительную простоту, он менее распространен, чем обертывание ключа, так как уязвим к слабым паролям. Предположим, что противник перехватил какое-то зашифрованное сообщение; если использовалось обертывание ключа, противнику нужно сначала получить файл с защищенным ключом, который обычно хранится в локальной файловой системе пользователя, так что добраться до него нелегко. Но если применялось динамическое генерирование, то противник может подобрать правильный пароль из числа кандидатов и попытаться таким образом де-

шифрировать сообщение. Если пароль слабый, то ключ будет скомпрометирован.

**Хранение ключа в аппаратном устройстве (на смарт-карте или на электронном USB-ключе)**

При таком подходе ключ хранится в защищенной памяти и останется в секрете, даже если компьютер будет скомпрометирован. Это самый безопасный способ хранения, но и самый дорогой и неудобный, потому что аппаратное устройство нужно носить с собой, рискуя потерять. Смарт-карты и электронные ключи обычно не требуют ввода пароля, чтобы извлечь ключ из памяти.

Чтобы протестировать обертывание ключа, необходимо выполнить команду OpenSSL с аргументом -aes128, который означает, что ключ нужно зашифровать шифром AES-128 (AES со 128-битовым ключом).

Запрошенная парольная фраза будет использоваться для шифрования вновь созданного ключа.

## Возможные проблемы

Криптографическая безопасность может быть нарушена многими способами. Самый серьезный риск — ложное чувство безопасности, возникающее благодаря имеющимся доказательствам безопасности или использованию хорошо изученных протоколов.

**Ложное чувство безопасности**

Даже доказательства безопасности, предложенные известными учеными, могут содержать ошибки. Один из самых ярких примеров чудовищной ошибки в доказательстве — метод оптимального асимметричного шифрования с дополнением Optimal asymmetric encryption padding (OAEP), который использовал RSA и был реализован во многих приложениях. Некорректное доказательство безопасности OAEP против атаки с подобранным шифротекстом считалось правильным в течение семи лет, пока другой исследователь не обнаружил в нем дефект в 2001 г. Доказательство содержало ошибку, и сам результат был неверен. Найденное впоследствии новое доказательство продемонстрировало, что OAEP не совсем безопасен против атаки с подобранным шифротекстом.

**Короткие ключи для поддержки унаследованных приложений**

Было обнаружено, что некоторые HTTPS-сайты и SSH-серверы поддерживают криптографию с открытым ключом меньшей длины, чем ожидалось, а именно 512 бит вместо 2048. Напомним, что в схемах с открытым ключом уровень безопасности не равен длине ключа, а в случае HTTPS ключи длиной 512 бит обеспечивают уровень безопасности примерно 60 бит. Для взлома таких ключей достаточно примерно двух недель вычислений на кластере из 72 процессоров. Проблема была обнаружена на многих сайтах, в том числе на сайте ФБР. Хотя в конечном итоге дефект был устранен (благодаря исправлениям в OpenSSL и других программах).

## Выводы по главе 5

Основное различие между безопасностью программного обеспечения и криптографической безопасностью заключается в том, что последняя поддается количественному измерению. В отличие от мира ПО, где приложение обычно считается либо безопасным, либо небезопасным, в мире криптографии часто можно оценить, сколько усилий придется приложить для взлома криптографического алгоритма. Кроме того, если основная цель безопасности ПО — помешать противнику использовать код программы для нанесения ущерба, то цель криптографической безопасности — сделать невозможным решение четко поставленной задачи.

Рассмотрено два определения понятия невозможного: информационная безопасность и вычислительная безопасность. Информационная безопасность — теоретическая невозможность, а вычислительная безопасность — практическая невозможность.

Сформулировано количественное измерение безопасности. Обнаружив атаку, первым делом необходимо узнать, насколько она эффективна теоретически и осуществима ли она на практике.

Показано, что на стоимость атаки влияют и другие факторы, которые нужно принимать во внимание, оценивая фактический уровень безопасности. Пояснено четыре главных фактора: парал-

лелизм, память, предварительные вычисления и количество мишеней.

Рассмотрена технология генерирования асимметричных ключей, которые обычно длиннее обеспечиваемого ими уровня безопасности.

Определено новшество в информационной защите — короткие ключи для поддержки унаследованных приложений. Было обнаружено, что некоторые HTTPS-сайты и SSH-серверы поддерживают криптографию с открытым ключом меньшей длины, чем ожидалось, а именно 512 бит вместо 2048.

## Вопросы для самоконтроля

1. В чем различие между безопасностью программного обеспечения и криптографической безопасностью?
2. Чем характеризуется вычислительная безопасность?
3. Каково количественное измерение безопасности?
4. Каково измерение безопасности в битах?
5. В чем заключается полная стоимость атаки?
6. Каково влияние параллелизма на стоимость атаки?
7. Каково влияние памяти на стоимость атаки?
8. Какое влияние предварительные вычисления оказывают на стоимость атаки?
9. Какое влияние количество мишеней оказывает на стоимость атаки?
10. В чем заключается достижение безопасности?
11. В чем заключается доказуемая безопасность?
12. В чем заключается эвристическая безопасность?
13. В чем заключается генерирование ключей?
14. В чем заключается генерирование симметричных ключей?
15. В чем заключается генерирование асимметричных ключей?
16. В чем заключается защита ключей?
17. В чем заключается обертывание ключа?
18. В чем заключается динамическое генерирование из пароля?
19. В чем заключается хранение ключа в аппаратном устройстве?
20. В чем заключаются короткие ключи для поддержки унаследованных приложений?

# Глава 6

## Авторизация и аутентификация[1]

Телекоммуникационные системы представляют собой технические средства, предназначенные для передачи больших объемов информации через различные каналы линий связи. Как правило, они предназначены для обслуживания большого количества пользователей: от нескольких десятков тысяч до миллионов. Использование такой системы предполагает наличие постоянно развивающейся системы информационной безопасности, которая предполагает защиту локальных информационных систем и их баз данных эффективными системами авторизации и аутентификации.

Рассмотрим одну из самых эффективных систем ASP.NET Identity Core MVC 6, встроенную в фреймворк ASP.NET. Эта система позволяет пользователям создавать учетные записи, аутентифицироваться, управлять учетными записями или использовать для входа на сайт учетные записи внешних провайдеров.

**Авторизация** — возможность для пользователя информационной системы (сайт, кабинет локальной информационной системы, доступ к любому защищенному объекту или ресурсу любой цифровой среды) получить доступ к интересующей информации, указав два значения: логин и пароль.

**Аутентификация** — технология процесса авторизации, позволяющая определить и проверить пользователя.

---

[1] Материал главы изложен по: *Freeman A.* Pro ASP.NET Core 6: Develop Cloud-Ready Web Applications Using MVC, Blazor, and Razor Pages. — 9th ed. — New York: Apress, 2022. — 1286 p.

ASP.NET Identity Core MVC является уникальной по возможностям и встроенной в фреймворк ASP.NET технологией создания и управления удостоверениями аутентификации и авторизации. Пользователи получают возможность создавать учетную запись с информацией для входа, хранящуюся в зашифрованном виде в базе данных сервера, или использовать данные внешнего поставщика входа.

Хранение удостоверений по умолчанию организуется в среде СУБД SQL или в облачной платформе для разработки сервисов и приложений Azure.

При использовании Microsoft Visual Studio 2019–2022, представляющей набор технологий интегрированной среды разработки (IDE) программного обеспечения может использоваться внутренний модуль безопасности базовых возможностей по защите для пользователя. Этот модуль указывается в общих параметрах создаваемого проекта «Создание веб-приложения с проверкой подлинности без сертификата или с ним».

## 6.1. Создание веб-приложения с проверкой подлинности

Система ASP.NET Core Identity представляет собой API-интерфейс от Microsoft, предназначенный для управления пользователями в приложениях ASP.NET. В этом параграфе демонстрируется процесс настройки ASP.NET Core Identity и создания простого инструмента администрирования пользователей, который управляет индивидуальными пользовательскими учетными записями, хранящимися в базе данных.

Система ASP.NET Core Identity поддерживает другие виды пользовательских учетных записей, такие как записи, хранящиеся с использованием Active Directory, но они редко применяются вне корпораций (где реализации Active Directive оказываются настолько замысловатыми, что очень трудно отыскать полезные общие примеры).

Для выполнения приведенных примеров на локальном компьютере необходимо наличие установленного средства SQL Server LocalDB для Visual Studio.

Также рассмотрим, как выполнять аутентификацию и авторизацию с помощью пользовательских учетных записей и каким образом выйти за рамки основ и применять ряд более сложных приемов.

В табл. 6.1 приведена сводка, позволяющая поместить систему ASP.NET Core Identity в контекст.

Таблица 6.1

**Помещение системы ASP.NET Core ldentity в контекст**

| Вопрос | Ответ |
|---|---|
| Что это такое? | Система ASP.NET Core ldentity — это APl-интерфейс для управления пользователями и запоминания пользовательских данных в хранилищах, таких как реляционные базы данных, посредством Entity Framework Core |
| Чем она полезна? | Управление пользователями является важной возможностью для большинства приложений. ASP.NET Core ldentity предлагает готовую хорошо протестированную платформу, которая не требует создания специальных версий распространенных функций |
| Как она используется? | Система ldentity используется через службы и промежуточное ПО, добавляемое в класс Start up, и посредством классов, которые действуют в качестве шлюзов между приложением и функциональностью Identity |

Система ASP.NET Core Identity работает с инфраструктурой ASP.NET Core аналогично тому, как было в предшествующих версиях, хотя она обновлена для согласования с системой служб и промежуточного ПО, а также имеет большее число компонентов, доступных через внедрение зависимостей. Сложную авторизацию можно выполнять с помощью проверок, основанных на политиках и ресурсах.

Подготовим проект для примера.

Создадим новый проект типа Empty (Пустой) по имени Users с использованием шаблона ASP.NET Core Web Application (.NET Core) (веб-приложение ASP.NET Core (.NET Core)). Добавим требуемые пакеты NuGet в раздел dependencies файла project, json и настроим инструментарий Razor в разделе tools, как показано в примере 6.1.

Пакеты, требующиеся для Identity, будут добавляться отдельно, чтобы подчеркнуть разницу между пакетами, необходимыми для общей разработки приложений MVC, и пакетами, предназначенными для аутентификации и авторизации.

**Пример 6.1.** Добавление пакетов в файле project.json

```
{
  "dependencies": {
    "Microsoft.NETCore.App": {
      "version": "1.0.0",
      "type": "platform"
    }
    "Microsoft.AspNetCore.Diagnostics": "1.0.0",
    "Microsoft.AspNetCore.Server.IISIntegration": "1.0.0",
    "Microsoft.AspNetCore.Server.Kestrel": ″1.0.0",
    "Microsoft.Extensions.Logging.Console": "1.0.0",
    "Microsoft.AspNetCore.Mvc": "1.0.0",
    "Microsoft.AspNetCore.StaticFiles": "1.0.0",
    "Microsoft.AspNetCore.Razor.Tools": {
      "version": "1.0.0-preview2-final",
      "type": "build"
    }
  },
"tools": {
  "Microsoft.AspNetCore.Razor.Tools": "1.0.0-preview2-final",
  "Microsoft.AspNetCore.Server.IlSIntegration,Tools":
"1.0.0-preview2-final"
},
"frameworks": {
  "netcoreappl, 0": {
    "imports": ["dotnet5.6","portable-net45+win8"]
  }
},
"buildOptions": {
  "emitEntryPoint": true, "preserveCompilationContext": true
},
"runtimeoptions": {
  "configProperties": {"System.GC.Server": true}
  }
}
```

В примере 6.2 показан код класса Startup, который конфигурирует средства, предоставляемые пакетами NuGet.

**Пример 6.2.** Содержимое файла Startup.cs

```
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.DependencyInjection;
```

```
namespace Users
{
  public class Startup
  {
    public void ConfigureServices(IServiceCollection ser-
vices)
    {
      services.AddMvc();
    }
    public void Configure(IApplicationBuilder app)
    {
      app.UseStatusCodePages();
      app.UseDeveloperExceptionPage();
      app.UseStaticFiles();
      app.UseMvcWithDefaultRoute();
    }
  }
}
```

### Создание контроллера и представления

Создадим папку Controllers, добавим файл класса по имени HomeController.cs и поместим в него определение контроллера из примера 6.3. Этот контроллер будет применяться для описания деталей пользовательских учетных записей и данных, а его метод действия Index() передает словарь значений стандартному представлению через метод View().

**Пример 6.3.** Содержимое файла Homecontroller.cs из папки Controllers

```
using System.Collections.Generic;
using Microsoft.AspNetCore.Mvc;

namespace Users.Controllers
{
  public class HomeController : Controller
  {
    public ViewResult Index() =>
      View(new Dictionary<string, object>
        {["Placeholder"] = "Placeholder" });
  }
}
```

Чтобы снабдить контроллер представлением, создадим папку Views/Home и добавим в нее файл представления по имени Index.cshtml с разметкой, приведенной в примере 6.4.

**Пример 6.4.** Содержимое файла Index.cshtml из папки Views/Home

```
@model Dictionary<string, object>

<div class="bg-primary panel-body"><h4>User Details</h4></div>

<table class="table table-condensed table-bordered">
    @foreach (var kvp in Model)
    {
        <tr><th>@kvp.Key</th><td>@kvp.Value</td></tr>
    }
</table>
```

Представление отображает содержимое словаря модели в таблице. Для поддержки представления создадим папку Views/ Shared и добавим в нее файл по имени Layout.cshtml с разметкой, показанной в примере 6.5.

**Пример 6.5.** Содержимое файла Layout.cshtml из папки Views/Shared

```
<!DOCTYPE html>
<html>
<head>
  <title>Users</title>
  <meta name="viewport" content="width=device-width" />
    <link href="/lib/bootstrap/dist/css/bootstrap.css"
rel="stylesheet" />
</head>
<body class="panel-body">
  @RenderBody()
</body>
</html>
```

При стилизации HTML-элементов представление полагается на CSS-пакет Bootstrap. Создадим в корневой папке проекта файл bower, json с использованием шаблона элемента Bower Configuration File (Файл конфигурации Bower) и добавим пакет Bootstrap в раздел dependencies (пример 6.6).

**Пример 6.6.** Добавление пакета Bootstrap в файле bower.json

```
{
"name": "asp.net",
"private": true,
```

```
"dependencies": {
  "bootstrap": "3.3.6"
  }
}
```

Последний подготовительный шаг связан с созданием файла Viewimports. cshtml в папке Views, в котором настраиваются встроенные дескрипторные вспомогательные классы для применения в представлениях (пример 6.7).

**Пример 6.7.** Содержимое файла ViewImports.cshtml из папки Views

```
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

Создадим в папке Views файл запуска представления по имени ViewStart.cshtml с содержимым из примера 6.8. Он обеспечит использование компоновки, созданной в примере 6.5, всеми представлениями в приложении.

**Пример 6.8.** Содержимое файла ViewStart.cshtml из папки Views

```
@{
  Layout = "_Layout";
}
```

Запустим приложение, увидим вывод, приведенный на рис. 6.1.

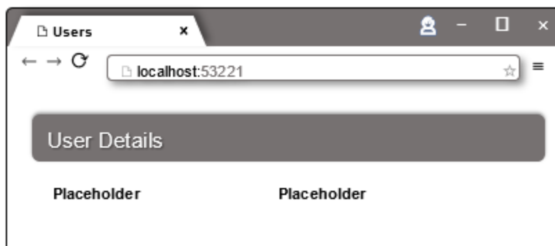


**Рис. 6.1.** Запуск примера приложения

## Настройка ASP.NET Core Identity

Процесс настройки системы Identity затрагивает почти каждую часть приложения, требуя новые классы моделей, изменения

конфигурации, а также контроллеров и действий для поддержки операций аутентификации и авторизации. Далее рассмотрим процесс настройки Identity в базовой конфигурации, чтобы продемонстрировать разнообразные шаги, которые с ним связаны. Задействовать систему Identity в приложении можно многими разными способами, но конфигурация, применяемая в этом параграфе, предусматривает использование самых простых и ходовых параметров.

## Добавление пакета Identity в приложение

В случае применения шаблона Empty среда Visual Studio не добавляет пакеты ASP.NET Core Identity или Entity Framework Core в создаваемые проекты, поэтому они должны быть добавлены вручную. Добавим требуемые пакеты в файл project.json (пример 6.9).

**Пример 6.9.** Добавление пакетов Identity в файле project.json

```
{
  "dependencies": {
    "Microsoft.NETCore.App": {
      "version": "1.0.0",
      "type": "platform"
    },
    "Microsoft.AspNetCore.Diagnostics": "1.0.0",
    "Microsoft.AspNetCore.Server.IlSIntegration": "1.0.0",
    "Microsoft.AspNetCore.Server.Kestrel": "1.0.0",
    "Microsoft.Extensions.Logging.Console": "1.0.0",
    "Microsoft.AspNetCore.Mvc": "1.0.0",
    "Microsoft.AspNetCore.StaticFiles": "1.0.0",
    "Microsoft.AspNetCore.Razor.Tools" : {
      "version": "1.0.0-preview2-final", "type": "build"
    }
    "Microsoft.Extensions.Configuration": "1.0.0",
    "Microsoft.Extensions.Configuration.Json": "1.0.0",
    "Microsoft.AspNetCore.Identity.EntityFrameworkCore":
"1.0.0",
    "Microsoft.EntityFrameworkCore.SqlServer": "1.0.0",
    "Microsoft.EntityFrameworkCore.Tools": "1.0.0-preview2-
final"
  },
  "tools": {
  "Microsoft.AspNetCore.Razor.Tools": "1.0.0-preview2-final",
```

```
  "Microsoft.AspNetCore.Server.IISIntegration.Tools":
"1.0.0-preview2-final",
  "Microsoft.EntityFrameworkCore.Tools": "1.0.0-preview2-final"
  },
  "frameworks": {
    "netcoreappl.O": {
      "imports": [ "dotnet5.6", "portable-net45+win8" ]
    }
  },
  "buildOptions": {
    "emitEntryPoint": true,
    "preserveCompilationContext": true
  },
  "runtimeoptions": {
    "configProperties": { "System.GC.Server" : true }
  }
}
```

Указанные пакеты добавим в проект ASP.NET Core Identity и Entity Framework Core. Добавление в разделе tools устанавливает инструменты командной строки, которые позволяют настраивать базу данных, используемую для хранения данных Identity, что вскоре будет сделано.

### Создание класса пользователя

Следующий шаг заключается в определении класса, предназначенного для представления пользователя в приложении, который называется классом пользователя. Класс пользователя наследуется от класса IdentityUser, определенного в пространстве имен Microsoft.AspNetCore.Identity.EntityFrameworkCore. Класс IdentityUser обеспечивает базовое представление пользователя, которое можно расширять, добавляя свойства к производному классу.

Индивидуальные свойства в настоящий момент неважны. Важно то, что класс IdentityUser предоставляет доступ к базовой информации о пользователе: имя пользователя, адрес электронной почты, телефонный номер, тип пароля, членство в ролях и т. д. (табл. 6.2). При желании хранить дополнительные сведения о пользователе необходимо добавить свойства в класс, унаследованный от IdentityUser, который будет использоваться для представления пользователей в приложении.

Таблица 6.2

**Свойства класса IdentityUser**

| Имя | Описание |
|---|---|
| Id | Содержит уникальный идентификатор пользователя |
| UserName | Возвращает имя пользователя |
| Claims | Возвращает коллекцию заявок (claim) пользователя |
| Email | Содержит адрес электронной почты пользователя |
| Logins | Возвращает коллекцию входов пользователя, используемую для сторонней аутентификации |
| PasswordHash | Возвращает хешированную форму пароля пользователя |
| Roles | Возвращает коллекцию ролей, к которым принадлежит пользователь |
| PhoneNumber | Возвращает телефонный номер пользователя |
| Securitystamp | Возвращает значение, которое изменяется, когда меняется удостоверение пользователя, например, из-за смены пароля |

Чтобы создать класс пользователя для приложения, создадим папку Models и добавим в нее файл класса по имени AppUser.cs с определением класса AppUser, приведенным в примере 6.10.

**Пример 6.10.** Содержимое файла AppUser.cs из папки Models

```
using.Microsoft.AspNetCore.Identity.EntityFrameworkCore;
namespace Users.Models }
  public class AppUser : IdentityUser {
  }
}
```

### Конфигурирование импортирования представлений

Чтобы упростить написание представлений, добавим пространство имен Users.Models в файл импортирования представлений (пример 6.11).

**Пример 6.11.** Добавление пространства имен в файле _Viewlmports.cshtml

```
using Users.Models
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

## Создание класса контекста базы данных

Следующий шаг предусматривает создание класса контекста базы данных Entity Framework Core, который оперирует с классом AppUser. Класс контекста унаследован от IdentityDbContext<T>, где T — класс пользователя (AppUser в текущем примере). Добавим в папку Models файл класса по имени AppIdentityDbContext.cs и определим в нем класс, как показано в примере 6.12.

**Пример 6.12.** Содержимое файла AppIdentityDbContext.cs из папки Models

```
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore;
namespace Users.Models {
  public class AppIdentityDbContext : IdentityDbContext<App-
User> {
     public AppIdentityDbContext(DbContextOptions<AppIden-
tityDbContext> options) : base(options) { }
  }
}
```

Класс контекста базы данных может быть расширен для изменения способа, которым база данных настраивается и используется, но в случае элементарного приложения ASP.NET Core Identity простого определения класса вполне достаточно, чтобы начать и получить заполнитель для любой настройки в будущем.

## Конфигурирование настройки строки подключения к базе данных

Первый шаг по конфигурированию ASP.NET Core Identity заключается в определении строки подключения к базе данных. По соглашению строка подключения помещается в файл appsettings.json, который затем загружается в классе Startup. Создадим в корневой папке проекта файл appsettings.json с использованием шаблона элемента ASP.NET Configuration File (Файл конфигурации ASP.NET) и добавим в него содержимое примера 6.13.

**Пример 6.13.** Содержимое файла appsettings.json

```
{
  "Data": {
    "SportStoreldentity": {
```

```
      "Connectionstring": "Server=(localdb)\\MSSQLLocalDB;
Database= Identityusers;Trusted_Connection=True;
MuitipleActiveResultSets=true"
    }
  }
}
```

В строке подключения указан параметр localdb, который предоставляет удобную поддержку баз данных для разработчиков. Кроме того, в качестве имени базы данных указывается IdentityUsers.

Имея строку подключения к базе данных, можно обновить класс Startup для чтения конфигурационного файла и обеспечения доступности настроек (пример 6.14).

**Пример 6.14.** Чтение настроек приложения в файле startup.cs

```
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Configuration;
using Microsoft.AspNetCore.Hosting;
namespace Users {
  public class Startup {
    IConfigurationRoot Configuration;
    public Startup(IHostingEnvironment env) {
      Configuration = new ConfigurationBuilder()
        .SetBasePath(env.ContentRootPath)
        .AddJsonFile("appsettings.json"),Build();
    }
    public void ConfigureServices(IServiceCollection services) {
      services.AddMvc();
      }
      public void Configure(lApplicationBuilder app) {
      app.UseStatusCodePages ();
      app.UseDeveloperExceptionPage();
      app.UseStaticFiles();
      app.UseMvcWithDefaultRoute();
    }
  }
}
```

Внесенные в код изменения загружают файл appsettings.json и представляют содержимое через свойство Configuration, которое применяется в следующем разделе при настройке служб и промежуточного ПО ASP.NET Core Identity.

## Конфигурирование служб и промежуточного программного обеспечения Identity

Финальный шаг настройки системы Identity предусматривает добавление служб и промежуточного ПО в класс Startup, чтобы внедрить Identity в конвейер обработки запросов и предоставить средства, которые используются для управления пользователями где-то в других местах приложения. Необходимые изменения показаны в примере 6.15.

**Пример 6.15.** Конфигурирование служб и промежуточного ПО ASP.NET Core Identity в файле Startup.cs

```
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Configuration;
using Microsoft.AspNetCore.Hosting;
using Microsoft.EntityFrameworkCore;
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Users.Models;
namespace Users
{
  public class Startup {
    IConfigurationRoot Configuration;
    puЬlic Startup(IHostingEnvironment env) {
      Configuration = new ConfigurationBuilder()
        .SetBasePath(env.ContentRootPath)
        .AddJsonFile(" appsettings.json").Build();
    }
    public void ConfigureServices(IServiceCollection services) {
      services.AddDbContext<AppIdentityDbContext>(options =>
        options.UseSqlServer(
          Configuration["Data:SportStoreIdentity:Connec-
tionString"]));
      services.AddIdentity<AppUser, IdentityRole>()
        .AddEntityFrameworkStores<AppIdentityDbContext>();
        .AddDefaultTokenProviders();
      services.AddMvc();
    }
    public void Configure(IApplicationBuilder app) {
      app.UseStatusCodePages();
      app.UseDeveloperExceptionPage();
      app.UseStaticFiles();
      app.UseIdentity();
      app.UseMvcWithDefaultRoute();
    }
  }
}
```

Для создания базовой установки ASP.NET Core Identity требуются три набора изменений. Сначала настраивается инфраструктура Entity Framework (EF) Core, которая предоставляет приложениям MVC службы доступ к данным:

```
services.AddDbContext<AppIdentityDbContext>(options =>
  options.UseSqlServer(Configuration[
    "Data:Sportstoreidentity:ConnectionString"]));
```

Метод AddDbContext () добавляет службы, требующиеся для EF, а метод UseSqlServer () настраивает поддержку, необходимую для хранения данных с применением Microsoft SQL Server. Метод AddDbContext () позволяет использовать ранее созданный класс контекста базы данных и указать, что он будет копироваться с базой данных SQL Server, строка подключения для которой получается из конфигурации приложения (в примере приложения это файл appsettings.json).

Понадобится также настроить службы для ASP.NET Core Identity, что делается следующим образом:

```
services.Addldentity<AppUser, IdentityRole>()
  .AddEntityFrameworkStores<AppIdentityDbContext>();
```

Метод Addidentity () имеет параметры типов, которые указывают класс, применяемый для представления пользователей, и класс, используемый для представления ролей. Здесь задается класс AppUser для пользователей и класс IdentityRole для ролей. Метод AddEntityFrameworkStores () указывает, что система Identity должна использовать инфраструктуру Entity Framework Core для сохранения и извлечения своих данных с применением созданного ранее класса контекста базы данных. Последнее изменение в классе Startup касается добавления системы ASP.NET Core Identity в конвейер обработки запросов. Это позволяет ассоциировать пользовательские учетные данные с запросами на основе cookie-наборов или переписывания URL. Детали пользовательских учетных записей не включаются напрямую в HTTP-запросы, отправляемые приложению, или в ответы, которые оно генерирует:

```
app.UseIdentity();
```

## Создание базы данных Identity

Осталось фактически создать базу данных, которая будет использоваться для хранения данных Identity. Откроем окно консоли диспетчера пакетов, выбрав пункт меню Tools ⇒ NuGet Package Manager (Сервис ⇒ Диспетчер пакетов NuGet) в Visual Studio, и введем следующую команду:

```
Add-Migration Initial
```

Как объяснялось при настройке базы данных для приложения SportsStore, инфраструктура Entity Framework Core управляет изменениями в схемах баз данных через средство, которое называется миграции. В случае модификации классов модели, применяемых для генерации схемы, можно сгенерировать файл миграции, который содержит команды SQL, предназначенные для обновления базы данных. Приведенная выше команда создает файлы миграции, которые будут настраивать базу данных Identity.

Когда команда завершит выполнение, в окне Solution Explorer появится папка Migrations. Просмотрев содержимое файлов в этой папке, обнаружим команды SQL, которые будут использоваться для создания начальной базы данных. Чтобы задействовать файлы миграции для создания базы данных, введем следующую команду:

```
Update-Database
```

Выполнение команды может занять некоторое время. После ее завершения база данных будет создана и готова к применению.

## Использование ASP.NET Core Identity

Выполнив базовую настройку, можно запустить и оценить применение ASP.NET Core Identity для добавления поддержки управления пользователями в пример приложения.

Инструменты централизованного администрирования пользователей полезны практически во всех приложениях, даже в тех, которые позволяют пользователям создавать и управлять собственными учетными записями. Часто необходимо пакетное создание учетных записей и поддержка задач, предполагающих проверку и изменение пользовательских данных.

## Оценка пользовательских учетных записей

Важной является оценка (просмотр) всех пользовательских учетных записей в базе данных, что позволит увидеть эффект от кода, который будет добавлен в приложение позже.

Сначала добавим в папку Controllers файл класса по имени AdminController.cs и определим в нем контроллер, как показано в примере 6.16, который будет применяться для реализации функциональности администрирования пользователей.

**Пример 6.16.** Содержимое файла AdminController.cs из папки Controllers

```
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using Users.Models;
namespace Users.Controllers {
  public class AdminController : Controller {
    private UserManager<AppUser> userManager;
    public AdminController(UserManager<AppUser> usrMgr) {
      userManager = usrMgr;
    }
    public ViewResult Index() => View(userManager.Users);
  }
}
```

Метод действия Index() показывает пользователей, управляемых системой Identity: разумеется, в текущий момент пользователи отсутствуют, но вскоре они появятся.

Доступ к пользовательским данным осуществляется через объект UserManager<AppUser>, который конструктор контроллера получает посредством внедрения зависимостей.

С помощью объекта UserManager<AppUser> можно запрашивать хранилище данных. Свойство Users возвращает перечисление объектов пользователей (экземпляров класса AppUser в рассматриваемом приложении), с которым удобно работать, используя LINQ (язык управления базой данных). Внутри метода действия значение свойства Users, которое будет перечислять всех пользователей в базе данных, передается методу View(), так что он сможет отобразить детали учетных записей. Чтобы снабдить метод действия представлением, создадим папку Views/Admin, добавим файл по имени Index.cshtml и поместим в него разметку из примера 6.17.

**Пример 6.17.** Содержимое файла Index.cshtml из папки Views/Admin

```
@model IEnumerable<AppUser>
<div class="bg-primary panel-body"><h4>User Accounts</h4>
</div>
<table class="table table-condensed table-bordered">
  <tr><th>ID</th><th>Name</th><th>Email</th></tr>
  @if (Model.Count () == 0) {
    <tr><td colspan="3" class="text-center">No User Ac-
counts</td></tr> )
    } else {
      foreach (AppUser user in Model) {
        <tr>
          <td>@user.Id</td><td>@user.UserName</td><td>
@user.Email</td></tr>
      }
    }
</table>
<a class="btn btn-primary" asp-action="Create">Create</a>
```

Представление содержит таблицу, в которой для каждого пользователя предусмотрена строка с колонками, отображающими уникальный идентификатор, имя пользователя и адрес электронной почты. Если пользователи в базе данных отсутствуют, выводится соответствующее сообщение, как показано на рис. 6.2, для чего понадобится запустить приложение и запросить URL вида /Admin.

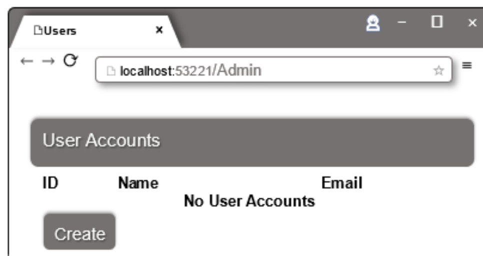


**Рис. 6.2.** Визуализация (пустого) списка пользователей

В представление включена ссылка Create (Создать), стилизованная под кнопку, которая нацелена на действие Create контроллера Admin. Это действие будет поддерживать добавление пользователей.

## Переустановка базы данных

Запустив приложение и перейдя на URL вида /Admin, мы заметим небольшую паузу перед отображением содержимого, визуализируемого из представления. Причина в том, что инфраструктура Entity Framework Core должна создать и подготовить базу данных к ее первому применению.

Посмотреть созданную базу данных можно, открыв окно SQL Server Object Explorer (Проводник объектов SQL Server) в Visual Studio. Если впервые используется окно SQL Server Object Explorer, нужно выбрать в меню Tools (Сервис) пункт Connect to Database (Подключиться к базе данных), чтобы сообщить среде Visual Studio о базе данных, с которой нужно работать. В качестве источника данных выберем Microsoft SQL Server, для имени сервера укажем (localdb) \mssqllocaldb, оставим отмеченным флажок Use Windows Authentication (Использовать аутентификацию Windows) и щелкнем на стрелке, раскрывающей поле Select Or Enter a Database Name (Выберите или введите имя базы данных). Спустя несколько секунд отобразится список доступных баз данных LocalDB, в котором должна быть возможность выбора базы данных IdentityUsers, относящейся к примеру приложения. Щелкнем на кнопку ОК, после чего в окне SQL Server Object Explorer появится новая запись. Среда Visual Studio запомнит базу данных, так что описанный процесс необходимо выполнить только один раз.

Для просмотра базы данных понадобится раскрыть элемент (localdb) \mssqllocaldb ⇒ Databases ⇒ IdentityUsers в окне SQL Server Object Explorer. Получим возможность увидеть таблицы, которые были созданы файлами миграции, с именами вроде AspNetUsers и AspNetRoles. После добавления пользователей отправлять запросы для просмотра содержимого таблиц.

Чтобы удалить базу данных, щелкнем правой кнопкой мыши на элементе IdentityUsers в окне SQL Server Object Explorer и выберем в контекстном меню пункт Delete (Удалить). В диалоговом окне Delete Database (Удаление базы данных) отметим оба флажка и щелкнем на кнопке ОК для удаления базы данных.

Для создания базы данных откроем окно консоли диспетчера пакетов и введем следующую команду:

```
Update-Database
```

База данных будет воссоздана и готова к применению, когда снова будет запущено приложение.

## Создание пользователей

В отношении входных данных, получаемых приложением, будет использоваться проверка достоверности моделей MVC, и легче всего это сделать за счет создания простых моделей представлений для каждой операции, поддерживаемой контроллером.

Создадим в папке Models файл класса по имени UserViewModels.cs с содержимым, приведенным в примере 6.18.

**Пример 6.18.** Содержимое файла UserViewModels.cs из папки Models

```
using System.ComponentModel.DataAnnotations;
namespace Users.Models
{
  public class CreateModel
  {
    [Required]
    public string Name { get; set; }
    [Required]
    public string Email { get; set; }
    [Required)
    public string Password { get; set; }
  }
}
```

Начальная модель называется CreateModel и определяет основные свойства, которые требуются для создания пользовательской учетной записи: имя пользователя, адрес электронной почты и пароль. Атрибут Required из пространства имен System.ComponentModel.DataAnnotations применяется для указания на обязательность значений всех трех свойств, определяемых моделью.

В примере 6.19 к контроллеру Admin добавлена пара методов действий Create (), на которые нацелена ссылка в представлении Index из предыдущего раздела: они используют стандартный прием для отображения пользователю представления в случае запроса GET и обработки данных формы при поступлении запроса POST.

**Пример 6.19.** Определение методов действий Create() в файле AdminController.cs

```
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using Users.Models;
using System.Threading.Tasks;
namespace Users.Controllers
{
  public class AdminController : Controller
  {
    private UserManager<AppUser> userManager;
    public AdminController(UserManager<AppUser> usrMgr)
    {
      userManager = usrMgr;
    }
    public ViewResult Index() => View(userManager.Users);
    public ViewResult Create() => View();
    [HttpPost]
    public async Task<IActionResult> Create(CreateModel model)
    {
      if (ModelState.IsValid)
      {
        AppUser user = new AppUser
        {
          UserName = model.Name,
          Email = model.Email
        };
        IdentityResult result
          = await userManager.CreateAsync(user,
model.Password);
        if (result.Succeeded)
        {
          return RedirectToAction("Index");
        }
        else
        {
          foreach (IdentityError error in result.Errors)
          {
            ModelState.AddModelError("", error.Description);
          }
        }
      }
      return View(model);
    }
  }
}
```

Важной частью листинга является метод действия Create(), принимающий аргумент CreateModel, который будет вызываться, когда администратор отправляет данные формы. Свойство Modelstate.IsValid применяется для проверки того, что данные содержат обязательные значения, тогда создается новый экземпляр класса AppUser, который передается асинхронному методу UserManager.CreateAsync():

```
AppUser user = new AppUser {UserName = model.Name, Email =
model.Email);
IdentityResult result = await userManager.CreateAsync(user,
model.Password);
```

В качестве результата метод CreateAsync() возвращает объект IdentityResult, который описывает исход операции посредством свойств, перечисленных в табл. 6.3.

Таблица 6.3

**Свойства класса IdentityResult**

| Имя | Описание |
|---|---|
| Succeeded | Возвращает true, если операция выполнилась успешно |
| Errors | Возвращает последовательность объектов IdentityError, описывающих ошибки, которые возникли при выполнении операции. Класс IdentityError предлагает свойство Description со сводкой по проблеме |

В методе действия Created с помощью свойства Succeeded выясняется, была ли создана новая запись о пользователе в базе данных. Если свойство Succeeded возвращает true, тогда клиент перенаправляется на действие Index, так что отобразится список пользователей:

```
if (result.Succeeded) {
  return RedirectToAction("Index");
} else {
  foreach (IdentityError error in result.Errors) {
    ModelState.AddModelError("", error.Description);
  }
}
```

Если свойство Succeeded возвращает false, тогда производится перечисление последовательности объектов IdentityError, которую предоставляет свойство Errors. Свойство Description ис-

пользуется для создания ошибки проверки достоверности на уровне модели с помощью метода Modelstate.AddModelError().

Чтобы наделить новые методы действий представлением, создадим в папке Views/Admin файл представления Create.cshtml и добавим в него разметку, показанную в примере 6.20.

**Пример 6.20.** Содержимое файла Create.cshtml из папки Views/Admin

```
@model CreateModel
<div class="bg-primary panel-body"><h4>Create
User</h4></div>
<div asp-validation-summary=" All" class="text-danger"></div>
<form asp-action="Create" method="post">
  <div class="form-group">
    <label asp-for="Name"></label>
    <input asp-for="Name" class="form-control" />
  </div>
  <div class="form-group">
    <label asp-for="Email"></label>
    <input asp-for="Email" class="form-control" />
  </div>
  <div class="form-group">
    <label asp-for="Password"></label>
    <input asp-for="Password" class="form-control" />
  </div>
  <button type="submit" class="btn btn-primary">Create
</button>
  <a asp-action="Index" class="btn btn-secondary">Cancel</a>
</form>
```

В этом представлении присутствует простая форма для сбора значений, которые инфраструктура MVC привяжет к свойствам объекта модели, переданного методу действия Create(), а также сводка по возможным ошибкам проверки достоверности.

### Тестирование функциональности создания

Чтобы протестировать возможность создания новой пользовательской учетной записи, запустим приложение, перейдем на URL вида /Admin и щелкнем на кнопку Create (Создать).

Заполним форму значениями из табл. 6.4.

После ввода значений щелкнем на кнопке Create. Система ASP.NET Core Identity создаст пользовательскую учетную запись, которая будет отображаться после перенаправления браузера на

метод действия Index(), что видно на рис. 6.3. Получим другой идентификатор, так как идентификаторы для пользовательских учетных записей генерируются случайным образом.

Таблица 6.4

**Значения для создания примера пользователя**

| Имя | Описание |
|---|---|
| Name | Joe |
| Email | joe@example.com |
| Password | Secretl23 |

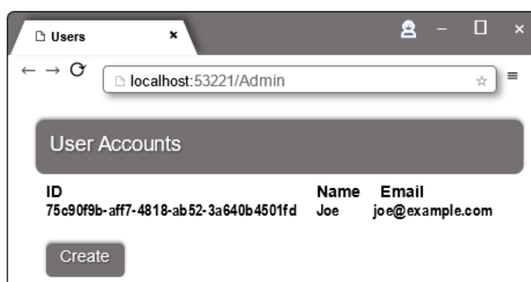


**Рис. 6.3.** Добавление новой пользовательской учетной записи

Щелкнем на кнопку Create еще раз и введем в элементах формы значения из табл. 6.4.

На этот раз после отправки формы получим ошибку, о которой сообщается в сводке по проверке достоверности модели (рис. 6.4).

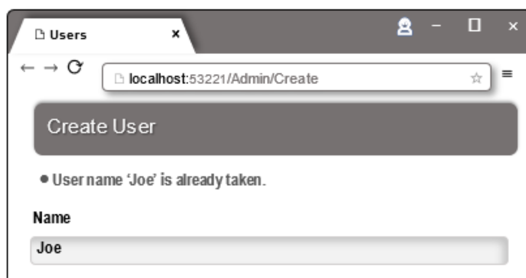


**Рис. 6.4.** Ошибка при попытке создать нового пользователя

## Проверка паролей

Одним из наиболее распространенных требований, особенно в корпоративных приложениях, является принудительное применение политики проверки паролей. Чтобы взглянуть на стандартную политику, запустим приложение, запросим URL вида /Admin/Create и заполним форму данными, приведенными в табл. 6.5; важное отличие их от данных из предыдущего раздела связано со значением, вводимым в поле пароля.

Таблица 6.5

**Значения для создания примера пользователя**

| Имя | Описание |
| --- | --- |
| Name | Alice |
| Email | alice@example.com |
| Password | secret |

Когда форма отправляется серверу, система Identity проверяет пароль-кандидат и генерирует ошибки, если он не удовлетворяет требованиям (рис. 6.5).

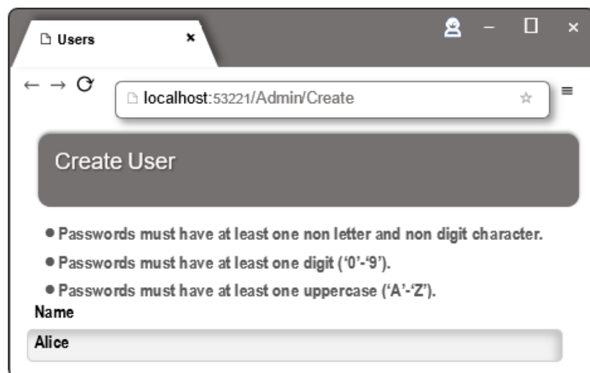


**Рис. 6.5.** Ошибки в результате проверки пароля

Правила проверки паролей можно сконфигурировать в классе Startup, как показано в примере 6.21.

**Пример 6.21.** Конфигурирование правил проверки паролей в файле Startup.cs

```
public void ConfigureServices(IServiceCollection services)
     {
       services.AddDbContext<AppIdentityDbContext>(options =>
         options.UseSqlServer(
           Configuration["Data:SportStoreIdentity:Connection-
String"]));
       services.AddIdentity<AppUser, IdentityRole>(opts => {
         opts.Password.RequiredLength = 6;
         opts.Password.RequireNonAlphanumeric = false;
         opts.Password.RequireLowercase = false;
         opts.Password.RequireUppercase = false;
         opts.Password.RequireDigit = false;
       }).AddEntityFrameworkStores<AppIdentityDbContext>()
       services.AddMvc();
}
```

Метод Addidentity() может использоваться с функцией, которая принимает объект IdentityOptions, чье свойство Password возвращает экземпляр класса Passwordoptions. Класс Passwordoptions предоставляет свойства для управления политикой проверки паролей, описанные в табл. 6.6.

Таблица 6.6

**Свойства класса Passwordoptions**

| Имя | Описание |
|---|---|
| RequiredLength | Это свойство типа int применяется для указания минимальной длины паролей |
| RequireNonAlphanumeric | Установка этого свойства типа bool в true требует, чтобы пароли содержали хотя бы один символ, не являющийся буквой или цифрой |
| RequireLowercase | Установка этого свойства типа bool в true требует, чтобы пароли содержали хотя бы один символ нижнего регистра |
| RequireUppercase | Установка этого свойства типа bool в true требует, чтобы пароли содержали хотя бы один символ верхнего регистра |

В примере 6.21 указано, что пароли должны иметь минимальную длину в шесть символов, а другие ограничения отключены. Это позволяет получить эффективную демонстрацию, но

не должно использоваться в реальном проекте. Запустив приложение, перейдя на URL вида /Admin/Create и повторив отправку формы, обнаружим, что пароль secret теперь принимается, и новая пользовательская учетная запись успешно создается (рис. 6.6).

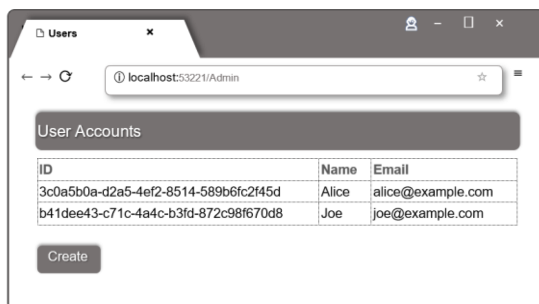


**Рис. 6.6.** Изменение политики проверки паролей

## Реализация специального класса проверки паролей

Встроенной проверки паролей вполне достаточно для большинства приложений, но может возникнуть необходимость в реализации специальной политики, особенно при разработке корпоративного производственного приложения, в котором сложные политики проверки паролей обычное явление. Функциональность проверки паролей определяется интерфейсом IPasswordValidator<T> из пространства имен Microsoft.AspNetCore.Identity, где T — класс пользователя, специфичный для приложения (AppUser в рассматриваемом примере):

```
using System.Threading.Tasks;
namespace Microsoft.AspNetCore.Identity {
  public interface IPasswordValidator<TUser> where TUser :
class { Task<IdentityResult> ValidateAsync(UserManager<TUser>
manager, TUser user, string password);
  }
}
```

Для проверки пароля вызывается метод ValidateAsync(), которому передаются данные контекста через объект UserManager (позволяющий выполнять запросы к базе данных Identity), представляющий объект пользователя и пароль-кандидат. В резуль-

тате возвращается объект IdentityResult, создаваемый с использованием статического свойства Success, если проблемы отсутствуют, или вызывается статический метод Failed(), которому передается массив объектов IdentityError, описывающих возникшие во время проверки проблемы.

Чтобы продемонстрировать применение специальной политики проверки, создадим папку Infrastructure и добавим в нее файл класса по имени CustomPasswordValidator.cs с определением из примера 6.22.

**Пример 6.22.** Содержимое файла CustomPassword-Validator.cs из папки Infrastructure

```
using System.Collections.Generic;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Identity;
using Users.Models;
namespace Users.Infrestructure (
  public class CustomPasswordValidator : IPasswordValida-
tor<AppUser> {
  public Task<IdentityResult> ValidateAsync(UserManager
<AppUser> manager, AppUser user, string password) {
    List<IdentityError> errors = new List<IdentityError> ();
    if (password.ToLower().Contains(user.UserName.ToLower())) {
      errors.Add(new IdentityError {
        Code = "PasswordContainsUserName",
        Description = "Password cannot contain username”
      });
    )
    if (password.Contains("12345")) {
      errors.Add(new IdentityError (
        Code = "PasswordContainsSequence",
        Description - "Password cannot contain numeric
sequence"
      });
    )
    return Task.FromResult(errors.Count == 0 ?
      IdentityResult.Success : IdentityResult.Failed(er-
rors.To-Array(());
      }
    }
}
```

Класс CustomPasswordValidator проверяет, не содержит ли пароль имя пользователя или последовательность 12345.

В примере 6.23 класс CustomPasswordValidator регистрируется как средство проверки паролей для объектов AppUser.

**Пример 6.23.** Регистрация специального класса проверки паролей в файле Startup.cs

```
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Configuration;
using Microsoft.AspNetCore.Hosting;
using Microsoft.EntityFrameworkCore;
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Users.Models;
using Users.Infrastructure;
using Microsoft.AspNetCore.Identity;
namespace Users {
  public class Startup {
    IConfigurationRoot Configuration;
    public Startup(IHostingEnvironment env) (
      Configuration = new ConfigurationBuilder()
        .SetBasePath(env.ContentRootPath)
        .AddJsonFile("appsettings.json").Build();
  }
  public void ConfigureServices(IServiceCollection services)
  {
    services.AddTransient<IPasswordValidator<AppUser>,
      CustomPasswordValidator>();
    services.AddDbContext<AppIdentityDbContext>(options => (
      options.UseSqlServer(
        Configuration["Data:Sportstoreidentity:Connection-
String"]));
    services.AddIdentity<AppUser, IdentityRole>(opts => {
      opts.Password.RequiredLength = 6;
      opts.Password.RequireNonAlphanumeric = false;
      opts.Password.RequireLowercase = false;
      opts.Password.RequireUppercase = false;
      opts.Password.RequireDigit = false;
      }).AddEntityFrameworkStores<AppIdentityDbContext>();
      services.AddMvc();
  }
  public void Configure(lApplicationBuilder app) {
    app.UseStatusCodePages();
    app.UseDeveloperExceptionPage() ;
    app.UseStaticFiles() ;
    app.Useidentity();
    app.UseMvcWithDefaultRoute() ;
    }
  }
}
```

Чтобы протестировать специальную политику, запустим приложение, запросим URL вида /Admin/Create и заполним форму значениями, приведенными в табл. 6.7.

Таблица 6.7

**Значения для создания примера пользователя**

| Имя | Описание |
|---|---|
| Name | Bob |
| Email | Bob@example.com |
| Password | bob12345 |

Пароль в табл. 6.7 нарушает оба правила проверки, навязываемые специальным классом, и вызывает отображение сообщений об ошибках, как показано на рис. 6.7.

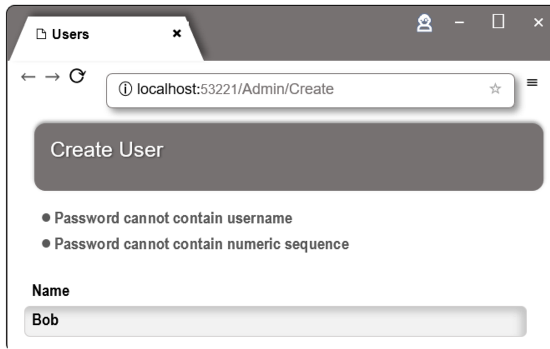


**Рис. 6.7.** Использование специального класса проверки паролей

Можно также реализовать специальную политику проверки, построенную на основе встроенного класса, который применяется по умолчанию. Стандартный класс называется PasswordValidator и определен в пространстве имен Microsoft.AspNetCore.Identity.

В примере 6.24 специальный класс проверки изменен так, что теперь он унаследован от класса PasswordValidator и основывается на поддерживаемых им базовых проверках.

**Пример 6.24.** Наследование от встроенного класса проверки в файле CustomPasswordValidator.cs

```
using System.Collections.Generic;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Identity;
using Users.Models;
using System.Linq;
namespace Users.Infrastructure
{
  public class CustomPasswordValidator : PasswordValida-
tor<AppUser>
  {
    public override async Task<IdentityResult> Validate-
Async(
      UserManager<AppUser> manager, AppUser user, string
password)
    {
      IdentityResult result = await base.ValidateAsync(manager,
        user, password);
      List<IdentityError> errors = result.Succeeded ?
        new List<IdentityError>() : result.Errors.ToList();
      if (password.ToLower().Contains(user.UserName.ToLower()))
      {
        errors.Add(new IdentityError
        {
          Code = "PasswordContainsUserName",
          Description = "Password cannot contain username"
        });
      }
      if (password.Contains("12345"))
      {
        errors.Add(new IdentityError
        {
          Code = "PasswordContainsSequence",
          Description = "Password cannot contain numeric
sequence"
        });
      }
      return errors.Count == 0 ? IdentityResult.Success
        : IdentityResult.Failed(errors.ToArray());
    }
  }
}
```

Для тестирования объединенной проверки запустим приложение и заполним данными из табл. 6.8 форму, возвращаемую для URL вида /Admin/Create.

Таблица 6.8

**Значения для создания примера пользователя**

| Имя | Описание |
|---|---|
| Name | Bob |
| Email | bob@example.com |
| Password | 12345 |

После отправки формы увидим сочетание сообщений об ошибках специальной и встроенной проверки (рис. 6.8).

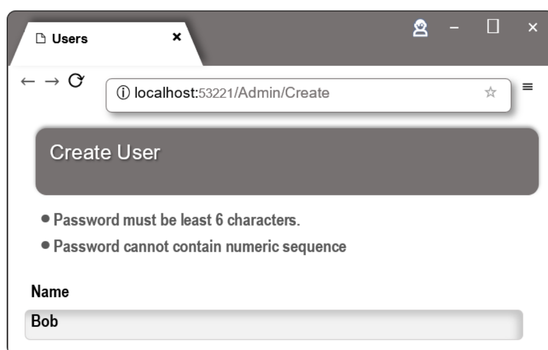


**Рис. 6.8.** Объединение специальной и встроенной проверки паролей

## Проверка деталей, связанных с пользователем

При создании учетной записи проверка выполняется также в отношении имени пользователя и адреса электронной почты. Чтобы оценить работу встроенной проверки, запустим приложение, запросим URL вида /Admin/Create и заполним форму данными из табл. 6.9.

Таблица 6.9

**Значения для создания примера пользователя**

| Имя | Описание |
|---|---|
| Name | Bob! |
| Email | alice@example.com |
| Password | secret |

Отправив форму, получим сообщение об ошибке (рис. 6.9).

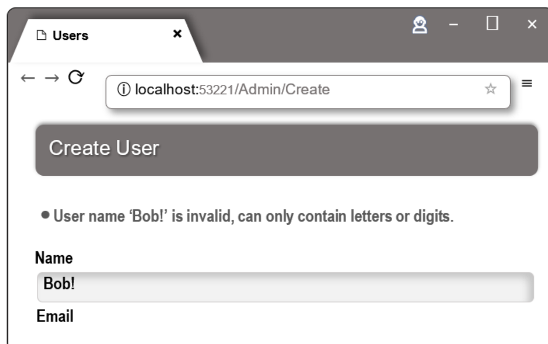


**Рис. 6.9.** Ошибка при проверке пользовательской учетной записи

Проверку можно конфигурировать в классе Startup с использованием свойства IdentityOptions.User, которое возвращает экземпляр класса UserOptions. Свойства UserOptions описаны в табл. 6.10.

Таблица 6.10

**Свойства класса UserOptions**

| Имя | Описание |
|---|---|
| AllowedUserNameCharacters | Это свойство типа string содержит все разрешенные символы, которые могут применяться в имени пользователя. В стандартном значении указаны символы a–z, A–Z, 0–9, переноса, точки, подчеркивания и @. Это свойство не является регулярным выражением, и каждый разрешенный символ должен задаваться явно в строке |
| RequireUniqueEmail | Установка этого свойства типа bool в true требует, чтобы для новых учетных записей указывались адреса электронной почты, которые не использовались ранее |

В примере 6.25 конфигурация приложения изменена так, чтобы уникальные адреса электронной почты были обязательными и в именах пользователей допускались только алфавитные символы нижнего регистра.

**Пример 6.25.** Изменение настроек проверки пользовательских учетных записей в файле Startup.cs

```
public void ConfigureServices(IServiceCollection services)
    {
      services.AddTransient<IPasswordValidator<AppUser>,
        CustomPasswordValidator>();
      services.AddDbContext<AppIdentityDbContext>(options =>
        options.UseSqlServer(
          Configuration["Data:SportStoreIdentity:
            ConnectionString"]));
      services.AddIdentity<AppUser, IdentityRole>(opts => {
        opts.User.RequireUniqueEmail = true;
        opts.User.AllowedUserNameCharacters =
          "abcdefghijklmnopqrstuvwxyz";
        opts.Password.RequiredLength = 6;
        opts.Password.RequireNonAlphanumeric = false;
        opts.Password.RequireLowercase = false;
        opts.Password.RequireUppercase = false;
        opts.Password.RequireDigit = false;
      }).AddEntityFrameworkStores<AppIdentityDbContext>()
      services.AddMvc();
    }
```

Повторно отправив данные из предыдущего теста, заметим, что адрес электронной почты сейчас приводит к ошибке, а символы в имени пользователя по-прежнему отклоняются (рис. 6.10).

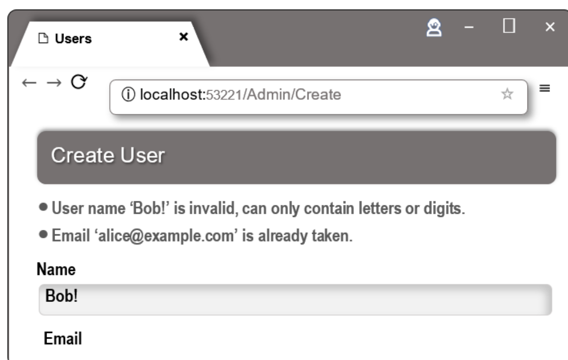


**Рис. 6.10.** Изменение настроек проверки пользовательских учетных записей

## Реализация специальной проверки пользователей

Функциональность проверки указывается с помощью интерфейса IUserValidator<T>, который определен в пространстве имен Microsoft.AspNetCore.Identity:

```
using System.Threading.Tasks;
namespace Microsoft.AspNetCore.Identity
{
  public interface IUserValidator<TUser> where TUser : class
  {
    Task<IdentityResult> ValidateAsync(UserManager<TUser>
manager, TUser user);
  }
}
```

Метод ValidateAsync() вызывается для выполнения проверки. В результате возвращается объект того же самого класса IdentityResult, который применялся при проверке паролей. Чтобы продемонстрировать работу специального класса проверки, добавим в папку Infrastructure файл класса по имени CustomUserValidator.cs с определением, представленным в примере 6.26.

**Пример 6.26.** Содержимое файла CustomUserValidator.cs из папки Infrastructure

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Identity;
using Users.Models;
namespace Users.Infrastructure
{
  public class CustomUserValidator : IUserValidator<AppUser>
  {
    public Task<IdentityResult> ValidateAsync(UserManager
<AppUser> manager, AppUser user)
    {
      if (user.Email.ToLower().EndsWith("@example.com"))
      {
        return Task.FromResult(IdentityResult.Success);
      }
      else
      {
        return Task.FromResult(IdentityResult.Failed(new
IdentityError
        {
          Code = "EmailDomainError",
```

```
            Description = "Only example.com email addresses
are allowed"
          }));
        }
      }
    }
}
```

Класс CustomUserValidator проверяет домен адреса электронной почты, чтобы удостовериться в том, что он является частью домена example.com. В примере 6.27 специальный класс регистрируется как средство проверки для объектов AppUser.

**Пример 6.27.** Регистрация специального класса проверки пользователей в файле Startup.cs

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddTransient<IPasswordValidator<AppUser>,
    CustomPasswordValidator>();
  services.AddTransient<IUserValidator<AppUser>,
    CustomUserValidator>();
  services.AddDbContext<AppIdentityDbContext>(options =>
    options.UseSqlServer(
      Configuration["Data:SportStoreIdentity:Connection-
String"]));
  services.AddIdentity<AppUser, IdentityRole>(opts => {
    opts.User.RequireUniqueEmail = true;
    opts.User.AllowedUserNameCharacters = "abcdefghijklm-
nopqrstuvwxyz";
    opts.Password.RequiredLength = 6;
    opts.Password.RequireNonAlphanumeric = false;
    opts.Password.RequireLowercase = false;
    opts.Password.RequireUppercase = false;
    opts.Password.RequireDigit = false;
  }).AddEntityFrameworkStores<AppIdentityDbContext>()
    .AddDefaultTokenProviders();
  services.AddMvc();
}
```

Для тестирования специального класса проверки запустим приложение и заполним форму, возвращаемую для URL вида /Admin/Create данными из табл. 6.11.

Имя пользователя и пароль успешно проходят проверку, но адрес электронной почты не относится к корректному домену. В результате отправки формы отобразится сообщение об ошибке проверки (рис. 6.11).

Таблица 6.11

**Значения для создания примера пользователя**

| Имя | Описание |
|---|---|
| Name | Bob |
| Email | bob@invalid.com |
| Password | secret |

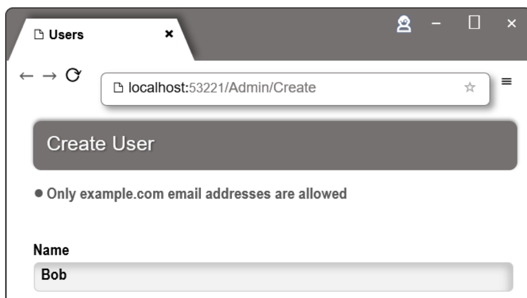


**Рис. 6.11.** Выполнение специальной проверки пользователей

Процесс сочетания встроенной проверки, обеспечиваемой классом UserValidator<T> и специальной проверки аналогичен такому процессу для проверки паролей (пример 6.28).

**Пример 6.28.** Расширение встроенного класса проверки пользователей в файле CustomUserValidator.cs

```
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Identity;
using Users.Models;
namespace Users.Infrastructure
{
  public class CustomUserValidator : UserValidator<AppUser>
  {
    public override async Task<IdentityResult> ValidateAsync(
    UserManager<AppUser> manager,AppUser user)
    {
      IdentityResult result = await base.ValidateAsync(manager,
user);
      List<IdentityError> errors = result.Succeeded ?
        new List<IdentityError>() : result.Errors.ToList();
      if (!user.Email.ToLower().EndsWith("@example.com"))
      {
```

```
          errors.Add(new IdentityError
          {
            Code = "EmailDomainError",
            Description = "Only example.com email addresses
are allowed"
          });
        }
        return errors.Count == 0 ? IdentityResult.Success
          : IdentityResult.Failed(errors.ToArray());
      }
    }
}
```

## Завершение средств администрирования

Для завершения инструмента администрирования осталось только реализовать средства для редактирования и удаления пользователей. В примере 6.29 показаны изменения, внесенные в файл Views/Admin/Index.cshtml для нацеливания на действия Edit и Delete контроллера Admin.

**Пример 6.29.** Добавление кнопок редактирования и удаления в файле Index.cshtml из папки Views/Admin

```
@model IEnumerable<AppUser>
<div class="bg-primary m-1 p-1 text-white"><h4>User Accounts
</h4></div>
<div class="text-danger" asp-validation-summary="ModelOnly"></div>
<table class="table table-sm table-bordered">
  <tr><th>ID</th><th>Name</th><th>Email</th></tr>
  @if (Model.Count() == 0)
  {
    <tr><td colspan="3" class="text-center">No User Accounts</td>
</tr>
  }
  else
  {
    foreach (AppUser user in Model)
    {
      <tr>
        <td>@user.Id</td>
        <td>@user.UserName</td>
        <td>@user.Email</td>
        <td>
          <form asp-action="Delete" asp-route-id="@user.Id"
method="post">
            <a class="btn btn-sm btn-primary" asp-action="Edit"
               asp-route-id="@user.Id">Edit</a>
            <button type="submit"
```

```
                 class="btn btn-sm btn-danger">
               Delete
             </button>
           </form>
         </td>
       </tr>
     }
   }
</table>
<a class="btn btn-primary" asp-action="Create">Create</a>
```

Кнопка Delete (Удалить) отправляет форму действию Delete контроллера Admin, что важно, поскольку при изменении состояния приложения требуется запрос POST. Кнопка Edit (Редактировать) является якорным элементом, который будет отправлять запрос GET, так как первый шаг в процессе редактирования предусматривает отображение текущих данных. Кнопка Edit содержится в элементе form, поэтому CSS-стили Bootstrap не укладывают ее вертикально.

Кроме того, в представление добавлена сводка по проверке достоверности модели, так что можно легко отображать сообщения об ошибках, которые поступают из остальных средств администрирования.

## Реализация средства удаления

В классе UserManager<T> определен метод DeleteAsync (), который принимает экземпляр класса пользователя и удаляет его из базы данных. В примере 6.30 метод DeleteAsync () используется для реализации средства удаления в контроллере Admin.

**Пример 6.30.** Удаление пользователей в файле AdminController.cs

```
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using Users.Models;
using System.Threading.Tasks;
namespace Users.Controllers
{
  public class AdminController : Controller
  {
    private UserManager<AppUser> userManager;
    public AdminController(UserManager<AppUser> usrMgr)
    {
      userManager = usrMgr;
```

```
    }
    // ... для краткости другие действия не показаны ...
    [HttpPost]
    public async Task<IActionResult> Delete(string id)
    {
      AppUser user = await userManager.FindByIdAsync(id);
      if (user != null)
      {
        IdentityResult result = await userManager.Delete-
Async(user);
        if (result.Succeeded)
        {
          return RedirectToAction("Index");
        }
        else
        {
          AddErrorsFromResult(result);
        }
      }
      else
      {
        ModelState.AddModelError("", "User Not Found");
      }
      return View("Index", userManager.Users);
    }
    private void AddErrorsFromResult(IdentityResult result)
    {
      foreach (IdentityError error in result.Errors)
      {
        ModelState.AddModelError("", error.Description);
      }
    }
  }
}
```

Метод действия Delete () получает в своем аргументе уникальный идентификатор пользователя и применяет метод FindByIdAsync() для нахождения соответствующего объекта пользователя, который можно передать методу DeleteAsync(). В качестве результата метод DeleteAsync() возвращает объект IdentityResult, который обрабатывается таким же образом, как в предшествующих примерах, чтобы обеспечить отображение пользователю сообщений о любых ошибках. Для тестирования функциональности удаления создадим нового пользователя и затем щелкнем на кнопке Delete, расположенной рядом с отображаемыми деталями об этом пользователе в представлении Index (рис. 6.12).

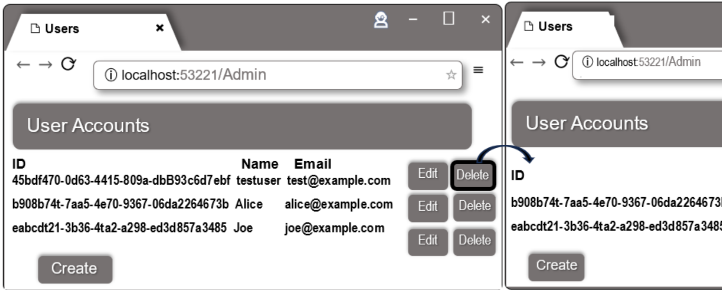


**Рис. 6.12.** Удаление пользовательской учетной записи

## Создание представления

Финальным компонентом является представление, которое будет отображать текущие значения для пользователя и позволит отправлять контроллеру новые значения. Добавим в папку Views/Admin файл по имени Edit.cshtml с содержимым, приведенным в примере 6.31.

**Пример 6.31.** Содержимое файла Edit.cshtml из папки Views/Admin

```
@model AppUser
<div class="bg-primary panel-body"><h4>Edit User</h4></div>
<div asp-validation-summary="All" class="text-danger"></div>
<form asp-action="Edit" method="post">
  <div class="form-group">
    <label asp-for="Id"></label>
    <input asp-for="Id" class="form-control" disabled />
  </div>
  <div class="form-group">
    <label asp-for="Email"></label>
    <input asp-for="Email" class="form-control" />
  </div>
  <div class="form-group">
    <label for="password">Password</label>
    <input name="password" class="form-control" />
  </div>
  <button type="submit" class="btn btn-primary">Save</button>
  <a asp-action="Index" class="btn btn-secondary">Cancel</a>
</form>
```

Представление отображает идентификатор пользователя, который не может быть изменен как статический текст и предла-

гает форму редактирования адреса электронной почты и пароля (рис. 6.13).

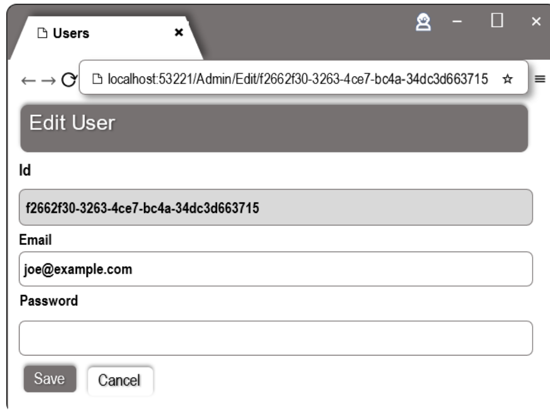


**Рис. 6.13.** Редактирование пользовательской учетной записи

Последнее изменение связано с помещением в комментарий настроек проверки пользователей в классе Startup, чтобы для имен пользователей использовались стандартные символы (пример 6.32). Ввиду того, что некоторые учетные записи в базе данных были созданы до изменения настроек проверки, отредактировать их не удастся, так как имена пользователей не пройдут проверку. А поскольку проверка применяется ко всему объекту пользователя, когда проверяется адрес электронной почты, результатом является пользовательская учетная запись, которую невозможно изменить.

**Пример 6.32.** Комментирование настроек проверки пользователей в файле Startup.cs

```
services.Addldentity<AppUser, IdentityRole>(opts => {
  opts.Password.RequiredLength = 6;
  opts.Password.RequireNonLetterOrDigit = false;
  opts.Password.RequireLowercase = false;
  opts.Password.RequireUppercase = false;
  opts.Password.RequireDigit = false;
  opts.User. RequireUniqueEmail = true;
  // opts.User.AllowedUserNameCharacters = "aЬcdefghijklm-
nopqrstuvwxyz";
}).AddEntityFrameworkStores<AppIdentityDbContext>();
```

Чтобы протестировать средство редактирования, запустим приложение, запросим URL вида /Admin и щелкнем на одной из кнопок Edit. Изменим адрес электронной почты или введем новый пароль (либо сделаем то и другое), потом щелкнем на кнопке Save для обновления БД и возвращения к URL вида /Admin.

## 6.2. Применение ASP.NET Core Identity

Продолжим рассмотрение проекта Users для дальнейшего изучения. В качестве примера создадим учетные записи пользователей, запустим наш проект, приложение, перейдем на URL вида /Admin и, щелкнув на кнопку Create (Создать), добавим в базу данных пользовательские учетные записи из табл. 6.12.

Таблица 6.12

**Пользовательские учетные записи**

| Имя пользователя | Адрес электронной почты | Пароль |
|---|---|---|
| Joe | joe@example.com | Secret123 |
| Alice | alice@example.com | Secret123 |
| Bob | bob@example.com | Secret123 |

По завершении запроса URL вида /Admin должен привести к отображению списка пользователей, включая описанные в табл. 6.12 (не имеет значения, если будут созданы дополнительные пользователи; важно, чтобы присутствовали пользователи, указанные в таблице), как показано на рис. 6.14.

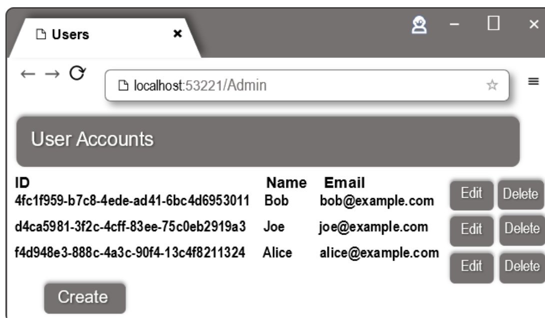


**Рис. 6.14.** Запуск примера приложения

## Аутентификация пользователей

Наиболее существенной работой для системы ASP.NET Core Identity является аутентификация пользователей. Основной инструмент для ограничения доступа к методам действий — атрибут Authorize, который сообщает инфраструктуре MVC о том, что обрабатываться должны только запросы от аутентифицированных пользователей.

В примере 6.33 атрибут Authorize применяется к действию Index контроллера Home.

**Пример 6.33.** Ограничение доступа в файле HomeController.cs

```
using System.Collections.Generic;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Authorization;
namespace Users.Controllers
{
  public class HomeController : Controller
  {
    [Authorize]
    public ViewResult Index() =>
      View(new Dictionary<string, object>
        { ["Placeholder"] = "Placeholder" });
  }
}
```

После запуска приложения браузер отправит запрос на стандартный URL, который будет нацелен на метод действия, декорированный атрибутом Authorize. Пока что у пользователей нет никакой возможности аутентифицировать себя, поэтому в результате возникает ошибка (рис. 6.15).

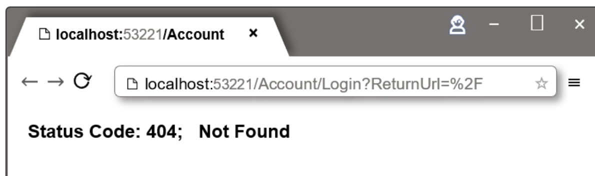


**Рис. 6.15.** Нацеливание на защищенный метод действия

Атрибут Authorize не указывает, как пользователь должен быть аутентифицирован и не имеет прямой ссылки на ASP.NET

Core Identity. Службы Identity и промежуточное ПО охватывают всю платформу ASP.NET, что делает их интеграцию в приложения MVC простой и бесшовной. Работа производится путем модификации объектов контекста, которые описывают HTTP-запросы, и снабжения MVC результатом процесса аутентификации без необходимости в предоставлении любых других деталей.

Платформа ASP.NET предоставляет информацию о пользователе через объект HttpContext, который используется атрибутом Authorize для проверки состояния текущего запроса и выяснения, был ли пользователь аутентифицирован. Свойство HttpContext.User возвращает реализацию интерфейса IPrincipal, который определен в пространстве имен System.Security.Principal. Интерфейс IPrincipal определяет свойство и метод, показанные в табл. 6.13.

Таблица 6.13

**Избранные члены, определяемые интерфейсом IPrincipal**

| Имя | Описание |
|---|---|
| Identity | Возвращает реализацию интерфейса Identity, который описывает пользователя, ассоциированного с запросом |
| IsInRole(role) | Возвращает true, если пользователь является членом указанной роли |

Реализация интерфейса Identity, возвращаемая свойством IPrincipal.Identity, предлагает базовую, но полезную информацию о текущем пользователе через свойства, описанные в табл. 6.14.

Таблица 6.14

**Базовая информация о текущем пользователе**

| Имя | Описание |
|---|---|
| AuthenticationType | Возвращает строку, которая описывает механизм, используемый для аутентификации пользователя |
| IsAuthenticated | Возвращает true, если пользователь был аутентифицирован |
| Name | Возвращает имя текущего пользователя |

Промежуточное ПО ASP.NET Core Identity применяет cookie-наборы, посылаемые браузером, для выяснения, был ли пользователь аутентифицирован. Если пользователь прошел

аутентификацию, тогда свойство Identity.IsAuthenticated устанавливается в true.

Браузер запрашивает URL вида /Account/Login, но из-за того, что в проекте он не соответствует какому-либо контроллеру или действию, сервер возвращает ответ 404 — Not Found (404 — не найдено), давая сообщение об ошибке, показанное на рис. 6.15.

### Изменение URL для входа

Хотя /Account/Login — это стандартный URL, на который клиенты перенаправляются, когда требуется авторизация, в методе ConfigureServices() класса Startup можно указать собственный URL, изменив параметр конфигурации при настройке служб ASP.NET Core Identity:

```
services.AddIdentity<AppUser, IdentityRole> {opts => {
  opts.Cookies.ApplicationCookie.LoginPath = "/Users/Login";
})
.AddEntityFrameworkStores <AppIdentityDbContext>();
```

Система Identity не может полагаться на систему маршрутизации при генерации своих URL, так что цель перенаправления должна указываться буквально. В случае изменения схемы маршрутизации, используемой приложением, потребуется также обеспечить изменение настройки Identity, чтобы URL по-прежнему достигал целевого контроллера.

### Добавление аутентификации пользователей

Запросы к защищенным методам действий корректно перенаправляются контроллеру Account, но учетные данные, предоставляемые пользователем, пока еще не используются для аутентификации.

В примере 6.34 завершается реализация действия Login за счет применения служб ASP.NET Core Identity для аутентификации пользователя с участием деталей, хранящихся в базе данных.

**Пример 6.34.** Добавление аутентификации в файле AccountController.cs

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
```

```
using Users.Models;
namespace Users.Controllers
{
  [Authorize]
  public class AccountController : Controller {
    private UserManager<AppUser> userManager;
    private SignInManager<AppUser> signInManager;
    puЬlic AccountController(Userмanager<AppUser> userMgr,
        SignInManager<AppUser> signinMgr) {
      userManager = userMgr;
      signInManager = signinMgr;
    }
    [AllowAnonymous]
    public IActionResult Login(string returnUrl)
    {
      ViewBag.returnUrl = returnUrl;
      return View();
    }
    [HttpPost]
    [AllowAnonymous]
    [ValidateAntiForgeryToken]
    public async Task<IActionResult> Login(LoginModel
      details, string returnUrl) {
      if (ModelState.IsValid) {
        AppUser user = await userManager.FindByEmail-
          Async(details.Email);
        if (user != null) {
          await signInManager.SignOutAsync();
          Microsoft.AspNetCore.Identity.SigninResult result=
            await signInManager.PasswordSignInAsync(
              user, details.Password, false, false);
          if (result.Succeeded) {
            return Redirect (returnUrl ? ? "/ ") ;
          }
        }
        ModelState.AddModelError(nameof(LoginModel.Email),
          "Invalid user or password");
      }
      return View(details);
    }
  }
}
```

Простейшей частью является получение объекта AppUser, который представляет пользователя, что делается посредством метода FindByEmailAsync() класса UserManager<AppUser>:

```
AppUser user = await userManager.FindByEmailAsync(details.Email);
```

Метод FindByEmailAsync() находит пользовательскую учетную запись, используя адрес электронной почты, который применялся при ее создании. Имеются также альтернативные методы поиска по идентификатору, по имени и по входу. Адрес электронной почты используется для входа из-за того, что такой подход принят в большинстве веб-приложений, доступных через Интернет, и он набирает популярность также в корпоративных приложениях.

В случае если учетная запись с указанным пользователем адресом электронной почты существует, тогда производится аутентификация с применением класса SignInManager<AppUser>, для которого добавляется аргумент конструктора, распознаваемый с помощью внедрения зависимостей. Класс SignInManager используется для выполнения двух шагов аутентификации:

```
await signInManager.SignOutAsync));
Microsoft.AspNetCore.Identity.SignInResult result =
  await signInManager.PasswordSignInAsync(user, details.Password,
false, false);
```

Метод SignOutAsync() аннулирует любой имеющийся у пользователя сеанс, а метод PasswordSignln() проводит саму аутентификацию. В качестве аргументов метод PasswordSignInAsync() получает объект пользователя, предоставленный пользователем пароль, булевское значение, управляющее постоянством cookie-набора аутентификации (отключено), и признак, должна ли учетная запись блокироваться в случае некорректного пароля (отключено).

Результатом метода PasswordSignInAsync() будет объект SignInResult, в котором определено булевское свойство Succeeded, указывающее на успешность аутентификации.

В рассматриваемом примере проверяется свойство Succeeded: если оно равно true, пользователь перенаправляется на местоположение returnUrl, а если false, тогда добавляется ошибка проверки достоверности и затем представление Login отображается заново, чтобы пользователь смог повторить попытку.

Как часть процесса аутентификации система Identity добавляет к ответу cookie-набор, который браузер затем включает в любые последующие запросы, чтобы идентифицировать сеанс поль-

зователя и ассоциированную с ним учетную запись. Мы не обязаны создавать или управлять этим cookie-набором напрямую, так как он поддерживается автоматически промежуточным ПО Identity.

## Редактирование членства в роли

Большая часть кода в версии метода действия Edit() для запросов GET отвечает за генерацию наборов членов и не членов выбранной роли. После того как все пользователи категоризированы, новый экземпляр класса RoleEditModel передается методу View(), так что данные могут быть отображены с применением стандартного представления. Версия метода действия Edit() для запросов POST отвечает за добавление и удаление пользователей в и из ролей. Класс UserManager<T> предлагает методы для работы с ролями, которые описаны в табл. 6.15.

Таблица 6.15

**Методы, связанные с ролями, которые определяет класс UserManager<T>**

| Имя | Описание |
|---|---|
| AddToRoleAsync(user, name) | Добавляет идентификатор пользователя к роли с указанным именем |
| GetRolesAsync(user) | Возвращает список имен ролей, членом которых является пользователь |
| IsInRoleAsync(user, name) | Возвращает true, если пользователь имеет членство в роли с указанным именем |
| RemoveFromRoleAsync(user, name) | Удаляет пользователя как члена из роли с указанным именем |

Причудливость методов, относящихся к ролям, связана с тем, что они оперируют с *именами* ролей, хотя роли имеют также и уникальные идентификаторы. По этой причине класс модели представления RoleModificationModel имеет свойство RoleName.

В примере 6.35 приведено содержимое файла Edit.cshtml, добавленного в папку Views/RoleAdmin, который позволяет пользователю редактировать членство в роли.

**Пример 6.35.** Содержимое файла Edit.cshtml из папки Views/RoleAdmin

```
@model RoleEditModel
<div class= "bg-primary panel-body"><h4>Edit Role</h4></div>
<div asp-validation-summary="All" class="text-danger"></div>
<form asp-action="Edit" method="post">
  <input type="hidden" name="roleName"
value="@Model.Role.Name" />
  <input type="hidden" name="roleId" value="@Model.Role.Id" />
  <h6 class="bg-info p-1 text-white">Add To
@Model.Role.Name</h6>
  <table class="table table-bordered table-sm">
    @if (Model.NonMembers.Count() == 0)
    {
      <tr><td colspan="2">All Users Are Members</td></tr>
    }
    else
    {
      @foreach (AppUser user in Model.NonMembers)
      {
        <tr>
          <td>@user.UserName</td>
          <td>
            <input type="checkbox" name="IdsToAdd"
value="@user.Id">
          </td>
        </tr>
      }
    }
  </table>
  <h6 class="bg-info p-1 text-white">Remove From
@Model.Role.Name</h6>
  <table class="table table-bordered table-sm">
    @if (Model.Members.Count() == 0)
    {
      <tr><td colspan="2">No Users Are Members</td></tr>
    }
    else
    {
      @foreach (AppUser user in Model.Members)
      {
        <tr>
          <td>@user.UserName</td>
          <td>
            <input type="checkbox" name="IdsToDelete"
value="@user.Id">
          </td>
        </tr>
```

```
        }
      }
   </table>
   <button type="submit" class="btn btn-primary">Save</button>
   <a asp-action="Index" class="btn btn-secondary">Cancel</a>
</form>
```

Представление содержит две таблицы: одну для пользователей, не являющихся членами выбранной роли; другую для пользователей, принадлежащих роли. Имя каждого пользователя отображается вместе с флажком, который позволяет изменять членство. Таблицы находятся внутри формы, которая отправляется методу действия Edit() и привязана к классу модели RoleModificationModel, обеспечивая легкий доступ к списку изменений членства в роли.

### Тестирование редактирования членства в роли

Чтобы протестировать поддержку членства в ролях, запустим приложение, перейдем на URL вида /RoleAdmin и создадим новую роль по имени Users. Щелкнем на кнопке Edit (Редактировать); все пользователи в приложении отобразятся внутри списка не членов (рис. 6.16).

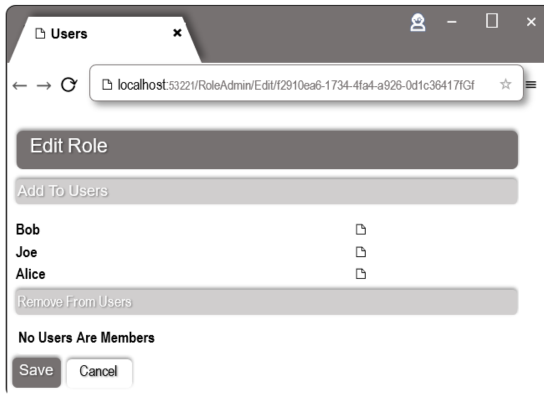


**Рис. 6.16.** Просмотр и редактирование членства в ролях

Отметим флажки для пользователей Alice и Joe (две из учетных записей, добавленных в систему Identity в начале главы)

и щелкнем на кнопке Save (Сохранить). В списке членов роли Users появятся пользователи Alice и Joe, как показано на рис. 6.17.

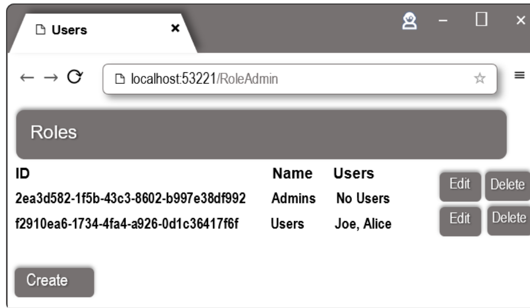


**Рис. 6.17.** Управление членством в ролях

## Использование ролей для авторизации

Теперь, когда в приложении имеются роли, их можно применять в качестве основы для авторизации посредством атрибута Authorize. Чтобы облегчить тестирование авторизации на основе ролей, добавим в контроллер Account метод Logout(), который сделает возможным выход и последующий вход от имени другого пользователя для демонстрации членства в ролях (пример 6.36).

**Пример 6.36.** Добавление метода Logout() в файле Accountcontroller.cs

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Users.Models;
using Microsoft.AspNetCore.Identity;

namespace Users.Controllers
{
  [Authorize]
  public class AccountController : Controller
  {
    private UserManager<AppUser> userManager;
    private SignInManager<AppUser> signInManager;

    // для краткости другие методы действий не показаны

    [Authorize]
    public async Task<IActionResult> Logout()
```

```
    {
      await signInManager.SignOutAsync();
      return RedirectToAction("Index", "Home");
    }
  }
}
```

Обновим контроллер Home, добавив новый метод действия и передав представлению информацию об аутентифицированном пользователе (пример 6.37).

**Пример 6.37.** Добавление метода действия и информации об учетной записи в файле HomeController.cs

```
using System.Collections.Generic;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Authorization;
namespace Users.Controllers
{
  public class HomeController : Controller
  {
    [Authorize]
    public IActionResult Index() => View(GetData(nameof(Index)));
    [Authorize(Roles = "Users")]
    public IActionResult OtherAction() => View("Index",
      GetData(nameof(OtherAction)));
    private Dictionary<string, object> GetData(string
actionName) =>
      new Dictionary<string, object>
      {
        ["Action"] = actionName,
        ["User"] = HttpContext.User.Identity.Name,
        ["Authenticated"] = HttpContext.User.Identity.IsAu-
thenticated,
        ["Auth Type"] = HttpContext.User.Identity.Authenti-
cationType,
        ["In Users Role"] = HttpContext.User.IsInRole("Users")
      };
  }
}
```

Атрибут Authorize для метода действия Index() не изменялся, но в случае его применения к методу OtherAction() было установлено свойство Roles с целью указания на то, что доступ к этому методу должен быть разрешен только членам роли Users. Кроме того, определен метод Get Data(), который добавляет базо-

вые сведения об удостоверении пользователя с использованием свойств, доступных через объект HttpContext.

Атрибут Authorize можно также применять для авторизации доступа на основе списка индивидуальных пользовательских имен. Это привлекательная возможность в небольших проектах, но она требует изменения кода в контроллерах всякий раз, когда изменяется набор авторизуемых пользователей, и обычно означает необходимость в повторном проходе через цикл тестирования и развертывания. Использование для авторизации ролей изолирует приложение от изменений в отдельных пользовательских учетных записях и позволяет управлять доступом к приложению посредством информации о членстве в ролях, хранящейся в системе ASP.NET Core Identity.

Последнее изменение касается файла Index.cshtml из папки Views/Home, который применяется обоими действиями в контроллере Home, оно связано с добавлением ссылки, нацеленной на метод Logout() контроллера Account (пример 6.38).

**Пример 6.38.** Добавление ссылки на метод Logout() в файле Index.cshtml из папки Views/Home

```
@model Dictionary<string, object>
<div class="bg-primary panel-body"><h4>User Details</h4></div>
<table class="table ttable-condensed table-bordered">
  @foreach (var kvp in Model)
  {
    <tr><th>@kvp.Key</th><td>@kvp.Value</td></tr>
  }
</table>
@if (User?.Identity?.IsAuthenticated ?? false)
{
  <a asp-controller="Account" asp-action="Logout"
    class="btn btn-danger">Logout</a>
}
```

Чтобы протестировать аутентификацию, запустим приложение и перейдем на URL вида /Home/Index. Браузер будет перенаправлен так, что можно ввести пользовательские учетные данные. Не имеет значения, данные какого пользователя из табл. 6.2 будут выбраны, поскольку атрибут Authorize, примененный к действию Index, разрешает доступ любому аутентифицированному пользователю.

Однако если запросить URL вида /Home/OtherAction, выбор пользователя будет иметь значение, так как членами роли Users, требующейся для доступа к методу OtherAction(), являются только пользователи Alice и Joe. В случае входа от имени пользователя Bob браузер будет перенаправлен на URL вида /Account/AccessDenied, который используется, когда пользователь не имеет возможности получить доступ к методу действия. Для обработки этой ситуации в контроллер Account добавлен метод AccessDenied(), так что теперь имеется действие, обрабатывающее запрос (пример 6.39).

**Пример 6.39.** Добавление метода действия в файле AccountController.cs

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Users.Models;
using Microsoft.AspNetCore.Identity;
namespace Users.Controllers
{
  [Authorize]
  public class AccountController : Controller
  {
    private UserManager<AppUser> userManager;
    private SignInManager<AppUser> signInManager;
    public AccountController(UserManager<AppUser> userMgr,
      SignInManager<AppUser> signinMgr)
    {
      userManager = userMgr;
      signInManager = signinMgr;
    }
    // для краткости другие методы действий не показаны
    [AllowAnonymous]
    public IActionResult AccessDenied()
    {
      return View();
    }
  }
}
```

Чтобы снабдить действие AccessDenied представлением для отображения, создадим в папке Views/Account файл по имени AccessDenied.cshtml с содержимым из примера 6.40.

**Пример 6.40.** Содержимое файла AccessDenied.cshtml из папки Views/Account

```
<div class = "bg-danger panel-body"xh4>Access Denied</h4x/div>
<a asp-action="Index" asp-controller="Home" class="btn
  btn-primary">OK</a>
```

Запустим приложение, запросим URL вида /Account/Login и войдем от имени bob@example.com. Когда процесс аутентификации завершится, браузер будет перенаправлен на URL вида /Home/Index, который отобразит детали учетной записи, как показано слева на рис. 6.18, проясняя, что пользователь Bob не является членом роли Users. Затем запросим URL вида /Home/OtherAction, который нацелен на действие, защищенное с помощью доступа на основе ролей. Пользователь Bob не имеет требуемого членства в роли, поэтому браузер будет перенаправлен на URL вида /Account/AccessDenied, как демонстрируется справа на рис. 6.18.

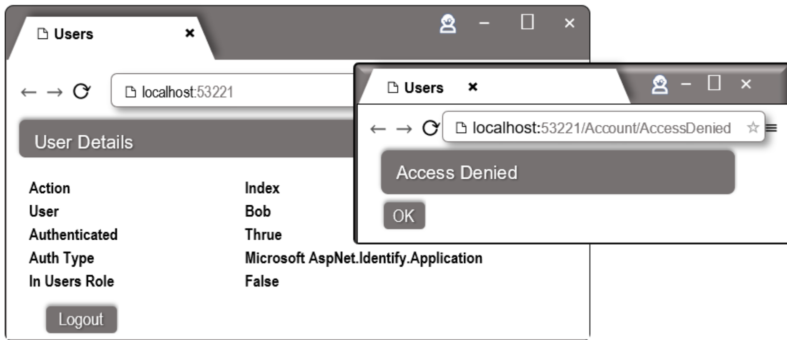


**Рис. 6.18.** Применение авторизации на основе ролей

## Помещение в базу данных начальных данных

В рассматриваемом проекте осталась одна проблема — доступ к контроллерам Admin и RoleAdmin не ограничен. Дело в том, что для ограничения доступа необходимо создать пользователей и роли, но контроллеры Admin и RoleAdmin являются инструментами управления пользователями. Если защитить их с помощью атрибута Authorize, не будет никаких учетных данных,

которые предоставят к ним доступ, особенно при первом развертывании приложения.

Проблема решается помещением в базу данных начальных данных, когда приложение запускается. В примере 6.41 приведено содержимое файла appsettings.json с добавленными конфигурационными данными, указывающими детали для учетной записи, которая будет создана.

**Пример 6.41.** Добавление конфигурационных данных в файле appsettings.json

```
{
  "Data": {
    "AdminUser": {
      "Name": "Admin",
      "Email": "admin@example.com",
      "Password": "secret",
      "Role": "Admins"
    },
    "SportStoreIdentity": {
      "ConnectionString": "Server=(localdb)\\MSSQLLocalDB;
Database=IdentityUsers; Trusted_Connection=True; MultipleAc-
tiveResultSets=true"
    }
  }
}
```

В категории Data:AdminUser предоставлены четыре значения, требующиеся для создания учетной записи и ее назначения роли, которая обеспечит возможность использования административных инструментов.

Добавьте в класс AppIdentityDbContext статический метод, как показано в примере 6.42. Код для создания стандартной учетной записи вовсе не обязан находиться в этом классе, но это место выглядит вполне естественным.

**Пример 6.42.** Добавление метода в файле AppIdentity-DbContext.cs

```
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using System;
using System.Threading.Tasks;
```

```
namespace Users.Models
{
  public class AppIdentityDbContext : IdentityDbContext<AppUser>
  {
    public AppIdentityDbContext(DbContextOptions<AppIdenti-
tyDbContext> options)
      : base(options) { }

    public static async Task CreateAdminAccount(IServicePro-
vider serviceProvider,
      IConfiguration configuration)
    {
      UserManager<AppUser> userManager =
        serviceProvider.GetRequiredService<UserManager<Ap-
pUser>>();
      RoleManager<IdentityRole> roleManager =
        serviceProvider.GetRequiredService<RoleManager<Iden-
tityRole>>();
      string username = configuration["Data:AdminUser:Name"];
      string email = configuration["Data:AdminUser:Email"];
      string password = configuration["Data:AdminUser:Pass-
word"];
      string role = configuration["Data:AdminUser:Role"];
      if (await userManager.FindByNameAsync(username) == null)
      {
        if (await roleManager.FindByNameAsync(role) == null)
        {
          await roleManager.CreateAsync(new Identity-
Role(role));
        }
        AppUser user = new AppUser
        {
          UserName = username,
          Email = email
        };
        IdentityResult result = await userManager
          .CreateAsync(user, password);
        if (result.Succeeded)
        {
          await userManager.AddToRoleAsync(user, role);
        }
      }
    }
  }
}
```

Метод CreateAdminAccount() принимает объект реализации IServiceProvider, который применяется для получения объектов UserManager и RoleManager, а также объект реализации Icon-

figuration, используемый для извлечения данных из файла appsetting.json. Код в методе CreateAdminAccount() проверяет, существует ли пользователь. Если нет, создает его и назначает указанной роли, которая также при необходимости создается. В примере 6.43 в класс Startup добавлен оператор, вызывающий метод CreateAdminAccount() после того, как остальная часть приложения настроена и сконфигурирована.

**Пример 6.43.** Вызов метода базы данных в файле Startup.cs

```
public void Configure(IApplicationBuilder app, IHostingEnvi-
ronment env)
{
  app.UseStatusCodePages();
  app.UseDeveloperExceptionPage();
  app.UseStaticFiles();
  app.UseAuthentication();
  app.UseMvcWithDefaultRoute();
  AppIdentityDbContext.CreateAdminAccount(app.Application-
Services,Configuration).Wait();
}
```

Имея надежную стандартную учетную запись в базе данных Identity, можно применить атрибут Authorize для защиты контроллеров Admin и RoleAdmin. В примере 6.44 приведены изменения, внесенные в контроллер Admin.

**Пример 6.44.** Ограничение доступа в файле AdminController.cs

```
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using Users.Models;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Authorization;

namespace Users.Controllers
{
  [Authorize(Roles = "Admins")]
  public class AdminController : Controller
  {
    // ... для краткости операторы не показаны ...
  }
}
```

В примере 6.45 показано соответствующее изменение, внесенное в контроллер RoleAdmin.

**Пример 6.45.** Ограничение доступа в файле RoleAdminController.cs

```
using System.ComponentModel.DataAnnotations;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using Users.Models;
using System.Collections.Generic;
using Microsoft.AspNetCore.Authorization;
namespace Users.Controllers
{
  [Authorize(Roles = "Admins")]
  public class RoleAdminController : Controller
  {
    // ... для краткости операторы не показаны ...
  }
}
```

Запустим приложение и запросим URL вида /Admin или /RoleAdmin. Если вы уже вошли от имени какого-то другого пользователя, необходимо выйти. В противном случае будет предложено предоставить учетные данные — ввести admin@example.com и пароль secret и получить доступ к административным функциям.

## Выводы по главе 6

Рассмотренные коды программ доступны в реальном проекте на сайте профессора В. П. Часовских (http://vikchas.ru).

Понятия «телекоммуникационные системы» и «информационной безопасности» тесно связаны. Телекоммуникационные системы представляют собой технические средства, предназначенные для передачи больших объемов информации через различные каналы линий связи. В зависимости от целей использования при оценке информации учитываются аспекты информационной безопасности. Информационная безопасность телекоммуникационных систем с десятками миллионов пользователей предполагает защиту локальных информационных систем и их баз данных эффективными системами авторизации и аутентификации пользователей.

В разделе рассмотрена одна из самых эффективных систем авторизации и аутентификации ASP.NET Identity, встроенная в ASP.NET.

Процесс настройки системы Identity затрагивает почти каждую часть приложения, требуя новые классы модели, изменений конфигурации, а также контроллеров и действий для поддержки операций аутентификации и авторизации.

Показано, что пакеты, требующиеся для Identity, должны добавляться отдельно, чтобы подчеркнуть разницу между пакетами, необходимыми для общей разработки приложений MVC, и пакетами, предназначенными для аутентификации и авторизации.

Рассмотрены вопросы добавления пакета Identity в приложение (информационную систему) и создание класса контекста базы данных. Определено конфигурирование настройки строки подключения к базе данных.

Продемонстрировано на примерах конфигурирование служб и промежуточного программного обеспечения Identity. Сложную авторизацию можно выполнять с помощью проверок, основанных на политиках и ресурсах.

Рассмотрены на примерах оценка пользовательских учетных записей, создание пользователей, проверка паролей, аутентификация пользователей. Продемонстрирована подготовка к реализации аутентификации, использование ролей для авторизации. Предложена технология по дополнительной защите веб-приложений, работа с заявками и политиками защиты данных.

## Вопросы для самоконтроля

1. Как авторизация влияет на информационную безопасность телекоммуникационных систем?
2. Как аутентификация влияет на информационную безопасность телекоммуникационных систем?
3. Каковы сравнительные характеристики платформы ASP.NET Identity?
4. Как создать и управлять записями пользователей в ASP.NET Identity?
5. Как организована технология хранения записей пользователей в базе данных?

6. Разрешается ли пользователю самому менять пароль и технологию изменений в базе данных ASP.NET Identity?
7. Можно ли создавать логин и пароль пользователя только администратором?
8. Каковы роли пользователей и их влияние на информационную безопасность информационной системы?
9. Какой фреймворк необходим для ASP.NET Identity? Перечислите основные характеристики.

# Заключение

Рассмотрена актуальная проблема настоящего времени — вопросы защиты информации, накапливаемой, хранимой и обрабатываемой в ЭВМ и построенных на их основе информационных системах. В последние годы появились информационные системы с технологиями искусственного интеллекта.

Было определено, что под защитой информации понимается создание в ЭВМ и информационных системах среды с технологиями искусственного интеллекта организованной совокупности средств, методов и мероприятий, предназначенных для предупреждения искажения, уничтожения или несанкционированного использования защищаемой информации.

Рассмотрены основные факторы, способствующие повышению уровня уязвимости телекоммуникационных систем и информационных приложений:

– резкое увеличение объемов информации накапливаемых, хранимых и обрабатываемых с помощью ЭВМ и других средств автоматизации, появление технологии Big Data, блокчейн, технологии искусственного интеллекта;

– сосредоточение в единых базах данных информации различного назначения и принадлежности;

– появление новых типов баз данных Big Data, баз данных в одно ранговых компьютерных сетях с технологией блокчейн, нейронных сетей;

– резкое расширение круга пользователей, имеющих непосредственный доступ к ресурсам вычислительной системы и находящимся в ней базам данных;

– усложнение режимов функционирования технических средств вычислительных систем: широкое внедрение многопрограммного режима, а также режимов разделения времени и реального времени, работа в сети Интернет и облачные технологии;

– кроссплатформенная обработка обмена информацией, в том числе и на больших расстояниях.

Определены необходимые специальные средства, методы и мероприятия, предназначенные для перекрытия потенциала повышения уровня уязвимости телекоммуникационных систем и информационных приложений и предупреждения этим несанкционированного использования информации.

Отдельно выделена практика применения криптокодирования. Наиболее ярким примером является технология блокчейн. Особую группу (как и в случае с аппаратными средствами) составляют программы шифрования информации.

Криптографическое закрытие (шифрование) информации заключается в таком преобразовании защищаемой информации, при котором по внешнему виду нельзя определить содержание закрытых данных.

Криптографической защите все специалисты уделяют особое внимание, считая ее наиболее надежной, а для информации, передаваемой по линиям связи большой протяженности, — единственным средством защиты информации от хищений.

Показано, что основой сенсорных технологий четвертой промышленной революции «Индустрия 4.0» (Industry 4.0) и Интернета вещей являются микроконтроллеры и их программное обеспечение. Именно они в рамках подключенных «умных» фабрик и «умных» домов предлагают огромный потенциал для роста и инноваций в этом бурно развивающемся бизнесе, но в то же время они делают системы уязвимыми для внешних атак.

Некоторые семейства микроконтроллеров уже включают множество функций безопасности. Дело в том, что они являются основными компонентами в среде управления в подключенных системах.

Определено, что информационная безопасность телекоммуникационных систем с десятками миллионов пользователей предполагает защиту локальных информационных систем и их баз данных эффективными системами авторизации и аутентификации.

Рассмотрена одна из самых эффективных систем авторизации и аутентификации ASP.NET Identity, встроенная в ASP.NET. Эта система позволяет пользователям создавать учетные записи,

аутентифицироваться, управлять учетными записями или использовать для входа на сайт учетные записи внешних провайдеров, таких как Facebook[1], Google, Microsoft, Twitter и др.

Также приведены конкретные программы и определены технологии их использования для создания эффективных средств авторизации и аутентификации с целью снижения уязвимости телекоммуникационных систем и информационных приложений.

[1] Facebook принадлежит компании Meta Platforms, Inc., которая признана экстремистской на территории Российской Федерации.

# Библиографический список

## Нормативные и правовые источники

*Об информации,* информационных технологиях и о защите информации: Федер. закон от 27 июля 2006 г. № 149-ФЗ.

*Об утверждении* Положения о государственной системе защиты информации в Российской Федерации от иностранных технических разведок и от утечки по техническим каналам: Постановление Правительства Российской Федерации от 15 сентября 1993 г. № 912-51.

*ГОСТ* 28147-89. Системы обработки информации. Защита криптографическая. Алгоритм криптографического преобразования.

*ГОСТ* Р ИСО/МЭК 9126-93. Информационная технлогия, Оценка программной продукции. Характеристики качества и руководства по их применению.

*ГОСТ* Р ИСО 7498-2-99. Информационная технология. Взаимосвязь открытых систем. Базовая эталонная модель. Часть 2. Архитектура защиты информации.

*ГОСТ* ISO/IEC TS 19249-2021. Межгосударственный стандарт Информационные технологии. Методы и средства обеспечения безопасности. Каталог принципов построения архитектуры и проектирования безопасных продуктов, систем и приложений Information technology. Security techniques. Catalogue of architectural and design principles for secure products, systems and applications.

# Рекомендуемая литература

*Баранова Е. К.* Основы информатики и защиты информации: учеб. пособие. — М.: РИОР: ИНФРА-М, 2013. — 183 с.

*Бирюков А. А.* Информационная безопасность: защита и нападение. — М.: ДМК Пресс, 2017. — 434 с.

*Глинская Е. В., Чичварин Н. В.* Информационная безопасность конструкций ЭВМ и систем: учеб. пособие. — М.: ИНФРА-М, 2019. — 118 с.

*Диогенес Ю., Озкайя Э.* Кибербезопасность: стратегии атак и обороны / пер. с англ. Д. А. Беликова. — М.: ДМК Пресс, 2020. — 326 с.

*Крамаров С. О., Тищенко Е. Н.* Криптографическая защита информации: учеб. пособие. — М.: РИОР, 2019. — 324 с.

*Маркс Р., Дембски У., Эверт У.* Введение в эволюционную информатику / пер. с англ. В. С. Яценкова. — М.: ДМК Пресс, 2020. — 276 с.

*Масалков А. С.* Особенности киберпреступлений в России: инструменты нападения и защита информации. — М.: ДМК Пресс, 2018. — 233 с.

*Омассон Ж.-Ф.* О криптографии всерьез / пер. с англ. А. А. Слинкина. — М.: ДМК Пресс, 2021. — 328 с.

*Силва В.* Разработка с использованием квантовых компьютеров. — СПб.: Питер, 2020. — 352 с.

*Хоффман Л. Дж.* Современные методы защиты информации. — М.: Советское радио, 1980. — 264 с.

*Шнайер Б.* Прикладная криптография: протоколы, алгоритмы и исходный код на С. — 2-е юбил. изд.; пер. с англ. — СПб.: Альфа-книга, 2017. — 1040 с.

*Freeman A.* Pro ASP.NET Core 6: Develop Cloud-Ready Web Applications Using MVC, Blazor, and Razor Pages. — 9th ed. — New York: Apress, 2022. — 1286 p.

# Информация об авторах

***Часовских Виктор Петрович*** — профессор кафедры шахматного искусства и компьютерной математики УрГЭУ, доктор технических наук, профессор

e-mail: u2007u@ya.ru

***Лабунец Валерий Григорьевич*** — профессор кафедры шахматного искусства и компьютерной математики УрГЭУ, доктор технических наук, профессор

e-mail: vlabunets05@yahoo.com

***Стариков Евгений Николаевич*** — заведующий кафедрой шахматного искусства и компьютерной математики УрГЭУ, кандидат экономических наук, доцент

e-mail: starikov_en@usue.ru

***Акчурина Галия Абдулазисовна*** — старший преподаватель кафедры шахматного искусства и компьютерной математики УрГЭУ

e-mail: mstrk@yandex.ru

***Кох Елена Викторовна*** — доцент кафедры шахматного искусства и компьютерной математики УрГЭУ, кандидат сельскохозяйственных наук, доцент

e-mail: elenakox@mail.ru

# Оглавление

**Введение ... 3**

**Глава 1. Основные определения и современные принципы проектирования систем, обеспечивающих безопасность ... 5**

1.1. Технологические аспекты обеспечения информационной безопасности ... 5

1.2. Организационные аспекты и принципы обеспечения информационной безопасности ... 9

*Выводы по главе 1 ... 15*

*Вопросы для самоконтроля ... 15*

**Глава 2. Полномочия и подлинность пользователей ... 16**

2.1. Идентификация и установление подлинности пользователя ... 16

2.2. Установка профилей полномочий пользователя ... 26

*Выводы по главе 2 ... 29*

*Вопросы для самоконтроля ... 30*

**Глава 3. Кибербезопасность средствами микроконтроллеров ... 32**

3.1. Микроконтроллеры как технология защиты от киберугроз ... 33

3.2. Функции защиты от киберугроз на аппаратной основе ... 35

3.3. Дополнительные средства защиты от кибератак ... 37

*Выводы по главе 3 ... 39*

*Вопросы для самоконтроля ... 39*

**Глава 4. Криптографические методы защиты информации ... 40**

4.1. Элементы криптосистемы ... 40

4.2. Криптоаналитические атаки ... 45

4.3. Безопасность алгоритмов ........................................... 47
*Выводы по главе 4*........................................................... *55*
*Вопросы для самоконтроля* ............................................... *56*

**Глава 5. Криптографическая безопасность ......................... 57**
5.1. Информационная и вычислительная безопасность ....... 57
5.2. Количественное измерение безопасности ...................... 60
5.3. Достижение безопасности ............................................. 65
5.4. Пути повышения криптобезопасности .......................... 68
*Выводы по главе 5*........................................................... *73*
*Вопросы для самоконтроля* ............................................... *74*

**Глава 6. Авторизация и аутентификация .............................. 75**
6.1. Создание веб-приложения с проверкой подлинности ................................................................ 76
6.2. Применение ASP.NET Core Identity............................ 116
*Выводы по главе 6*......................................................... *133*
*Вопросы для самоконтроля* ............................................. *134*

**Заключение ............................................................................ 136**

**Библиографический список ................................................. 139**

*Учебное издание*

**Часовских** Виктор Петрович,
**Акчурина** Галия Абдулазисовна,
**Лабунец** Валерий Григорьевич
и др.

# Информационная безопасность телекоммуникационных систем

Учебное пособие

Редактор и корректор *П. А. Давыдова*
Компьютерная верстка *И. В. Засухиной*

Поз. 24. Подписано в печать 11.09.2023.
Формат 60 × 84 1/16. Бумага офсетная. Печать плоская.
Уч.-изд. л. 6,3. Усл. печ. л. 8,4. Печ. л. 9,0. Заказ 505. Тираж 24 экз.
Издательство Уральского государственного экономического университета
620144, г. Екатеринбург, ул. 8 Марта / Народной Воли, 62/45

Отпечатано с готового оригинал-макета в подразделении оперативной полиграфии
Уральского государственного экономического университета

УРАЛЬСКИЙ ГОСУДАРСТВЕННЫЙ
ЭКОНОМИЧЕСКИЙ УНИВЕРСИТЕТ