Алексей Хлебников

OpenSSL 3: ключ к тайнам криптографии

ДМК Пресс
издательство

Современный интернет кишит червями, троянцами, незаконными посредниками и другими угрозами. Защита от них в наше время приобретает особую значимость.

OpenSSL — один из самых широко используемых и важных проектов с открытым исходным кодом в интернете — предназначен для обеспечения сетевой безопасности. Из этой книги вы узнаете о самых важных возможностях OpenSSL и о его широком потенциале. Приводятся пошаговые объяснения основ криптографии и сетевой безопасности, а также практические примеры, иллюстрирующие эти идеи. Прочитав книгу, вы сможете реализовать криптографическую защиту и TLS в своих приложениях и сетевой инфраструктуре.

Среди рассматриваемых тем:

- симметричное шифрование и способы его использования;
- хеши сообщений, MAC и HMAC;
- асимметричное шифрование и цифровые подписи;
- применение сертификатов X.509;
- продвинутое и специальное использование протокола TLS;
- эксплуатация миниатюрного удостоверяющего центра в организации.

Издание будет полезно разработчикам ПО, системным администраторам, инженерам по сетевой безопасности и специалистам по DevOps.

Алексей Хлебников – специалист в IT-сфере с более чем 20-летним стажем. В числе прочего работал над знаменитым интернет-браузером Opera, отвечая за модуль, связанный с безопасностью. В настоящее время проживает в Осло (Норвегия), работает старшим консультантом в bespoke AS.

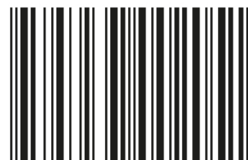
Packt>

Интернет-магазин:
www.dmkpress.com

Оптовая продажа:
КТК «Галактика»
books@alians-kniga.ru

www.дмк.рф

ISBN 978-5-93700-205-1



Алексей Хлебников

OpenSSL 3.
Ключ к тайнам криптографии

Alexei Khlebnikov

Demystifying Cryptography with OpenSSL 3.0

Discover the best techniques to enhance your network security with OpenSSL 3.0

BIRMINGHAM—MUMBAI

Алексей Хлебников

OpenSSL 3. Ключ к тайнам криптографии

Лучшие способы повысить безопасность сети с применением OpenSSL 3

Москва, 2023

УДК 003.26
ББК 32.373
Х55

Хлебников А.

Х55 OpenSSL 3. Ключ к тайнам криптографии / пер. с англ. А. А. Слинкина. – М.: ДМК Пресс, 2023. – 300 с.: ил.

ISBN 978-5-93700-205-1

Книга рассказывает о возможностях OpenSSL — одного из самых широко используемых и важных проектов с открытым исходным кодом в интернете, предназначенного для обеспечения сетевой безопасности. Вы изучите основы криптографии на конкретных примерах, узнаете о симметричном и асимметричном шифровании, об открытых и закрытых ключах и цифровых подписях, о специальном использовании протокола TLS и обо многом другом. Прочитав книгу, вы сможете реализовать криптографическую защиту и TLS в своих приложениях и сетевой инфраструктуре.

Издание адресовано разработчикам ПО, системным администраторам, инженерам по сетевой безопасности и специалистам по DevOps.

УДК 003.26
ББК 32.373

First published in the English language under the title ‘Demystifying Cryptography with OpenSSL 3.0’ – (9781800560345).

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

ISBN 978-1-80056-034-5 (англ.)
ISBN 978-5-93700-205-1 (рус.)

© 2022 Packt Publishing
© Перевод, оформление, издание, ДМК Пресс, 2023

Моей любимой матушке Татьяне Хлебниковой,
благодаря стараниям, заботе, любви и поддержке которой
я стал тем, кем я есть, и смог продолжить
свое развитие самостоятельно.

– Алексей Хлебников

Содержание

От издательства 13

Вступительное слово 14

Об авторе 15

О рецензенте 16

Предисловие 17

Часть I. Введение 21

Глава 1. OpenSSL и другие библиотеки SSL/TLS 22

Что такое OpenSSL? 22
История OpenSSL 24
Что нового в OpenSSL 3.0? 24
Сравнение OpenSSL с GnuTLS 25
Сравнение OpenSSL с NSS 26
Сравнение OpenSSL с Botan 27
Сравнение OpenSSL с облегченными библиотеками TLS 27
Сравнение OpenSSL с LibreSSL 29
Сравнение OpenSSL с BoringSSL 30
Резюме 31

Часть II. Симметричная криптография 32

Глава 2. Симметричное шифрование и расшифрование 33

Технические требования 33
Что такое симметричное шифрование 34
Обзор симметричных шифров, поддерживаемых OpenSSL 35

Сравнение блочных и потоковых шифров 35
Стойкость симметричного шифра 37
Сколько битов стойкости достаточно? 38
Обзор шифра AES 38
Обзор шифров DES и 3DES 39
Обзор шифра RC4 40
Обзор шифра ChaCha20 40
Обзор других симметричных шифров, поддерживаемых OpenSSL 41
Национальные шифры 41
Семейство шифров RC 43
Другие шифры 43
Режимы работы блочных шифров 44
Обзор режима простой замены 44
Обзор режима простой замены с зацеплением 45
Обзор режима CTR 47
Обзор режима GCM 48
Обзор режима AES-GCM-SIV 51
Другие режимы работы блочных шифров 51
Выбор режима работы блочного шифра 52
Дополнение для блочных шифров 52
Как сгенерировать ключ симметричного шифрования 53
Скачивание и установка OpenSSL 54
Шифрование и расшифрование с применением AES в командной строке 56
Инициализация и очистка библиотеки OpenSSL 60
Компиляция и компоновка с OpenSSL 61
Шифрование с применением AES из программы 63
Реализация программы шифрования 64
Выполнение программы encrypt 67
Расшифрование с применением AES из программы 68
Реализация программы decrypt 68
Выполнение программы decrypt 70
Резюме 71

Глава 3. Хеш-значения сообщений 72

Технические требования 72
Что такое хеш-значение сообщения и криптографическая функция хеширования? 73
Зачем нужны хеши сообщений? 74
Проверка целостности данных 74
Хеш-значения как основа HMAC 74
Цифровые подписи 75
Сетевые протоколы 75
Проверка пароля 75
Идентификатор содержимого 75
Блокчейн и криптовалюты 76

Доказательство выполнения работы ... 76
Оценка стойкости криптографических функций хеширования ... 77
Обзор криптографических функций хеширования, поддерживаемых OpenSSL ... 78
Семейство функций хеширования SHA-2 ... 78
Семейство криптографических функций SHA-3 ... 79
Функции хеширования SHA-1 и SHA-0 ... 81
Семейство функций хеширования MD ... 82
Семейство функций хеширования BLAKE2 ... 83
Менее популярные функции хеширования, поддерживаемые OpenSSL ... 84
Национальные криптографические функции хеширования ... 84
Другие криптографические хеш-функции ... 84
Какую криптографическую функцию хеширования выбрать? ... 85
Вычисление хеш-значения в командной строке ... 85
Вычисление хеш-значения из программы ... 86
Реализация программы digest ... 87
Выполнение программы digest ... 88
Резюме ... 88

Глава 4. MAC и HMAC ... 89

Технические требования ... 89
Что такое имитовставка? ... 90
Стойкость функции вычисления MAC ... 90
HMAC – MAC на основе функции хеширования ... 91
MAC, шифрование и принцип криптографической обреченности ... 93
Вычисление HMAC в командной строке ... 95
Вычисление HMAC из программы ... 96
Реализация программы hmac ... 97
Выполнение программы hmac ... 98
Резюме ... 98

Глава 5. Формирование ключа шифрования из пароля ... 100

Технические требования ... 100
В чем разница между паролем и ключом шифрования? ... 101
Что такое функция формирования ключа? ... 101
Обзор функций формирования ключа, поддерживаемых OpenSSL ... 104
Формирование ключа из пароля в командной строке ... 105
Формирование ключа из пароля из программы ... 105
Реализация программы kdf ... 106
Выполнение программы kdf ... 107
Резюме ... 108

Часть III. Асимметричная криптография и сертификаты 109

Глава 6. Асимметричное шифрование и расшифрование 110

Технические требования 110
Что такое асимметричное шифрование 111
Что такое атака с человеком посередине 111
Личная встреча 112
Проверка цифрового отпечатка ключа по телефону 112
Разделение ключа 113
Подписание ключа доверенной третьей стороной 113
Какие виды асимметричного шифрования доступны в OpenSSL 113
Что такое сеансовый ключ 114
Криптостойкость RSA 115
Генерирование пары ключей RSA 118
Шифрование и расшифрование с помощью RSA в командной строке 120
Шифрование с помощью RSA из программы 122
Реализация программы rsa-encrypt 123
Выполнение программы rsa-encrypt 125
Что такое очередь ошибок в OpenSSL 126
Расшифрование с помощью RSA из программы 129
Реализация программы rsa-decrypt 130
Выполнение программы rsa-decrypt 131
Резюме 131

Глава 7. Цифровые подписи и их проверка 132

Технические требования 133
Что такое цифровая подпись 133
Различие между цифровыми подписями и имитовставками 134
Обзор алгоритмов цифровых подписей, поддерживаемых OpenSSL 135
Алгоритм RSA 135
Алгоритм DSA 135
Алгоритм ECDSA 136
Алгоритм EdDSA 137
Алгоритм SM2 138
Какой алгоритм цифровой подписи выбрать? 139
Генерирование пары ключей эллиптического шифрования 139
Подписание и проверка подписи в командной строке 141
Подписание из программы 142
Реализация программы ec-sign 143
Выполнение программы ec-sign 145
Проверка подписи из программы 145
Реализация программы ec-verify 146
Выполнение программы ec-verify 147
Резюме 148

Глава 8. Сертификаты X.509 и инфраструктура открытых ключей 149

Технические требования 150
Что такое сертификат X.509? 150
Цепочки сертификатов 152
Как выпускаются сертификаты X.509 156
Что такое расширения X509v3? 159
Инфраструктура открытых ключей X.509 159
Генерирование самоподписанного сертификата 160
Генерирование несамоподписанного сертификата 162
Проверка сертификата в командной строке 165
Проверка сертификата из программы 166
Реализация программы x509-verify 166
Выполнение программы x509-verify 168
Резюме 169

Часть IV. TLS-подключения и безопасная связь 170

Глава 9. Установление TLS-подключений и передача по ним данных 171

Технические требования 172
Что такое протокол TLS 172
TLS-квитирование 174
Что происходит после TLS-квитирования? 176
История протокола TLS 177
Установление клиентского TLS-подключения в командной строке 180
Подготовка сертификатов для подключения к TLS-серверу 181
Прием запроса на подключение к TLS-серверу в командной строке 183
Базовые объекты ввода-вывода в OpenSSL 183
Установление клиентского TLS-подключения из программы 186
Реализация программы tls-client 187
Выполнение программы tls-client 190
Прием запроса на подключение к TLS-серверу из программы 191
Реализация программы tls-server 192
Выполнение программы tls-server 197
Резюме 199

Глава 10. Использование сертификатов X.509 в TLS 200

Технические требования 200
Специальная проверка сертификата другой стороны в программах на C 201
Регистрация функции обратного вызова для проверки 203
Реализация функции обратного вызова для проверки 204

Выполнение программы 207
Использование списков отозванных сертификатов в программах на C 208
Регистрация функции обратного вызова для поиска в CRL 210
Реализация функции обратного вызова для поиска в CRL 211
Реализация функции скачивания CRL с точки распространения 213
Реализация функции скачивания CRL с URL-адреса типа HTTP 214
Выполнение программы 216
Протокол онлайновой проверки состояния сертификата 217
Что такое протокол онлайновой проверки состояния сертификата 217
Использование OCSP в командной строке 218
Использование OCSP в программах на C 221
Регистрация функции обратного вызова для OCSP 223
Реализация функции обратного вызова для OCSP 223
Выполнение программы 227
Использование клиентских сертификатов в TLS 228
Генерирование клиентских сертификатов TLS 228
Упаковка клиентских сертификатов в контейнер PKCS #12 229
Программный запрос и проверка клиентского сертификата TLS на стороне сервера 230
Проверка клиентского сертификата TLS 231
Реализация функции построения ответа 232
Выполнение программы 233
Программное установление TLS-подключения с клиентским сертификатом 235
Изменение кода, унаследованного от программы tls-client 236
Загрузка сертификата TLS-клиента 237
Выполнение программы 240
Резюме 240

Глава 11. Специализированные применения TLS 242

Технические требования 242
Что такое закрепление сертификатов в TLS 243
Использование закрепления сертификатов 245
Изменение функции run_tls_client() 246
Реализация функции cert_verify_callback() 248
Выполнение программы tls-cert-pinning 249
Что такое блокирующие и неблокирующие сокеты 251
Использование неблокирующих сокетов в TLS 252
Изменение функции run_tls_client() 252
Выполнение программы tls-client-non-blocking 257
Что такое TLS на нестандартных сокетах 258
Использование TLS на нестандартных сокетах 259
Реализация функции service_bios() 260
Переработанная функция run_tls_client() 262
Выполнение программы tls-client-memory-bio 267
Резюме 269

Часть V. Мини-УЦ ... 270

Глава 12. Эксплуатация мини-УЦ ... 271

Технические требования ... 271
Подкоманда openssl ca ... 271
Генерирование сертификата корневого УЦ ... 273
Генерирование сертификата промежуточного УЦ ... 278
Генерирование сертификата для веб-сервера ... 282
Генерирование сертификата веб-клиента и почтового клиента ... 283
Отзыв сертификатов и генерирование CRL ... 285
Информирование о состоянии отзыва сертификатов по протоколу OCSP ... 288
Резюме ... 291

Предметный указатель ... 292

От издательства

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв на нашем сайте www.dmkpress.com, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com; при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг, мы будем очень благодарны, если вы сообщите о ней главному редактору по адресу dmkpress@gmail.com. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательство «ДМК Пресс» очень серьезно относится к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу электронной почты dmkpress@gmail.com.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

Вступительное слово

Больше тридцати лет занимаясь кодированием, я много раз встречал на своем пути ребят, которые могли бы написать книгу, а то и десять книг о распространенных, но от того не менее сложных технологиях. Но редко доводилось сталкиваться с людьми, которые умеют исследовать тему так глубоко, как Алексей Хлебников.

Я имею удовольствие работать и дружить с ним на протяжении вот уже нескольких лет и с нетерпением жду возможности с головой погрузиться в эту книгу. Вне всяких сомнений, эта тема заслуживает самого пристального внимания.

В этой книге мы совершим путешествие по основам OpenSSL, криптографии вообще, криптографическим режимам, «прелестям» сертификатов и установлению TLS-подключений. И с мельчайшими деталями, если я правильно понимаю, что делает Алексей, а я уверен, что понимаю правильно.

Это важная книга, в ней подробно рассматриваются технологии, которые мы принимаем как само собой разумеющиеся и к которым прибегаем постоянно, чтобы обезопасить свое присутствие в сети.

В книге достаточно примеров и пошаговых объяснений основополагающих концепций, чтобы вы могли продвигаться без затруднений. Прочитав ее до конца, вы сможете использовать наиболее популярные средства OpenSSL в своих программах как для настольных компьютеров, так и для веба.

Книга, безусловно, представляет интерес для практиков, но будет небезынтересна также руководителям и всем прочим, кто считает безопасность важным вопросом, но мало знает о ней. Не волнуйтесь: чтобы прочитать и понять ее, глубокие знания математики не понадобятся.

Как технарь с длинным послужным списком, я считаю, что узнавать новое или глубже погружаться в темы, которые знаешь поверхностно, – дело полезное и благодарное, поддерживающее нас в форме!

Твой выход, Алексей!

– Ярле Адольфсен
Серийный предприниматель, технический директор bspoke,
бывший технический директор Link Mobility
и пионер компьютерной графики в конце 1980-х – начале 1990-х

Об авторе

Алексей Хлебников более 20 лет профессионально занимается ИТ, где ему довелось попробовать себя в разных ролях: разработчик ПО, системный администратор, инженер DevOps, технический руководитель, архитектор и руководитель проекта. За эти годы Алексей поработал с разными технологиями: безопасность, искусственный интеллект, веб-разработка, встраиваемые приложения, мобильные приложения и робототехника. В частности, Алексей работал в компании Opera Software над знаменитым браузером Opera. Алексей всегда интересовался безопасностью. Он был одним из тех, кто сопровождал модули браузера Opera, относящиеся к безопасности, отвечал за криптографию, SSL/TLS и интеграцию с OpenSSL. Он также был членом архитектурной группы по безопасности, отвечавшей за безопасность Opera. Ныне Алексей живет в Осло, Норвегия, и работает старшим консультантом в компании bspoke AS. Он также возглавляет архитектурную группу у своего нынешнего работодателя.

Прежде всего я хочу поблагодарить свою любимую жену Ларису и нашего сына Дмитрия за постоянную любовь и поддержку, в том числе во время написания этой книги, когда я уделял им меньше внимания. Я также благодарен всем редакторам, менеджерам и другим сотрудникам издательства Packt Publishing, работавшим над этой книгой, и ее техническому рецензенту Крису. Их помощь и ценные советы помогли мне улучшить книгу во благо читателей

О рецензенте

Кшиштоф Квятковский – инженер-криптограф, в фокусе интересов которого находятся исследования по криптографии и их реализация. Он имеет ученую степень магистра математики по вычислительным методам. На протяжении своей 15-летней карьеры Крис работал над различными вопросами криптографии, связи и безопасности программного обеспечения – от встраиваемых до больших распределенных систем. В настоящее время его интересует реализация современных криптографических схем, стойких относительно квантовых технологий, а также помощь организациям, желающим перейти на них.

Я хочу поблагодарить свою чудесную любящую семью, которая понимает мою загруженность и всегда занимает мою сторону

Предисловие

В наши дни безопасность и использование сетей – неотъемлемая особенность программного обеспечения. Современный интернет кишит червями, троянцами, незаконными посредниками и другими угрозами. Поэтому обеспечение безопасности важнее, чем когда-либо в прошлом.

OpenSSL – один из самых широко используемых и важных проектов с открытым исходным кодом в интернете – предназначен именно для этой цели. Если вы разработчик ПО, системный администратор, инженер по сетевой безопасности или специалист по DevOps, то, скорее всего, сталкивались с этим комплектом инструментов в прошлом. Но как использовать его с наибольшей пользой? Из этой книги вы узнаете о самых важных возможностях OpenSSL и получите представление обо всем его потенциале.

В книге приводятся пошаговые объяснения основ криптографии и сетевой безопасности, а также практические примеры, иллюстрирующие эти идеи. Мы начнем с простого – как выполнить симметричное шифрование и вычислить хеш-значение (digest) сообщения. Затем пойдем дальше и поговорим о MAC и HMAC, открытых и закрытых ключах и цифровых подписях. По ходу дела вы узнаете о сертификатах стандарта X.509, инфраструктуре открытых ключей и TLS-подключениях.

Прочитав книгу, вы сможете использовать наиболее популярные средства OpenSSL, что позволит реализовать криптографическую защиту и TLS в своих приложениях и сетевой инфраструктуре.

Целевая аудитория

Эта книга ориентирована на разработчиков ПО, системных администраторов, специалистов по DevOps, инженеров по сетевой безопасности и аналитиков, которые хотят обезопасить свои приложения и инфраструктуру. Разработчики узнают, как использовать библиотеку OpenSSL, чтобы включить в свои программы криптографические средства и TLS. Специалисты по DevOps и системные администраторы научатся работать с криптографическими ключами и сертификатами из командной строки и узнают, как организовать миниатюрный удостоверяющий центр для своей организации. Предполагается знакомство с основами безопасности и сетевых технологий.

Структура книги

В главе 1 «OpenSSL и другие библиотеки SSL/TLS» в общих чертах описывается, что такое OpenSSL и его сильные стороны, рассматривается история

OpenSSL и новые возможности, появившиеся в OpenSSL 3.0. Мы также сравним OpenSSL с другими библиотеками SSL/TLS.

В главе 2 «Симметричное шифрование и расшифрование» рассматриваются важные понятия симметричного шифрования: шифры, режимы шифрования и дополнение. Мы поговорим о современных шифрах, режимах шифрования и типах дополнения и дадим совет, какую технологию использовать в конкретной ситуации. Применение этих технологий иллюстрируется с помощью команд и кода на C.

Глава 3 «Хеш-значения сообщений» посвящена хеш-значениям сообщений, или, как их еще называют, криптографическим хешам: зачем они нужны и как используются. Мы дадим обзор современных криптографических функций хеширования, которые вычисляют хеш-значения сообщений, и порекомендуем, какие функции в каких ситуациях использовать. Вычисление хеш-значений иллюстрируется с помощью команд и кода на C.

В главе 4 «MAC и HMAC» объясняется, зачем нужен код аутентичности сообщений (имитовставка, *англ.* Message Authentication Codes – MAC) и где он используется. Затем обсуждается популярный тип MAC – имитовставка на основе односторонних хеш-функций (*англ.* hash-based MAC – HMAC). Мы также узнаем о том, как сочетать HMAC с шифрованием, и о принципе криптографической обреченности. Вычисление HMAC иллюстрируется примером кода.

В главе 5 «Формирование ключа шифрования из пароля» показано, почему сам пароль нельзя использовать для шифрования, а необходимо вывести из него ключ. Мы дадим обзор современных функций формирования ключа и порекомендуем, в каких ситуациях какую использовать. Формирование ключа шифрования иллюстрируется с помощью команд и кода на C.

В главе 6 «Асимметричное шифрование и расшифрование» речь пойдет о том, зачем нужно асимметричное шифрование, как оно работает и какую роль в шифровании и расшифровании играют закрытые и открытые ключи. Шифрование и расшифрование с применением RSA иллюстрируются с помощью команд и кода на C.

В главе 7 «Цифровые подписи и их проверка» мы расскажем, для чего нужны цифровые подписи и как они используются. Мы дадим обзор современных алгоритмов цифровой подписи, в частности RSA, ECDSA и EdDSA, и порекомендуем, какую схему в какой ситуации использовать. Формирование и проверка цифровой подписи иллюстрируются с помощью команд и кода на C.

В главе 9 «Сертификаты X.509 и инфраструктура открытых ключей» подробно описывается, что такое сертификаты X.509, зачем они нужны и где используются. Мы объясним, как одни сертификаты применяются для подписания других, как формируются цепочки сертификатов, а также что такое **инфраструктура открытых ключей** (Public Key Infrastructure – PKI) и как проверка сертификатов используется для проверки идентификаторов, например идентификаторов сайтов. Применение этих методов иллюстрируется с помощью команд и кода на C.

В главе 9 «Установление TLS-подключений и передача по ним данных» подробно рассматривается, что такое протокол TLS, зачем он нужен и по-

чему так широко используется. Мы также узнаем, в чем разница между SSL и TLS. Затем научимся устанавливать и разрывать TLS-подключение, а также передавать и принимать по нему данные. Работа с TLS иллюстрируются с помощью команд и кода на C.

Глава 10 «Использование сертификатов X.509 в TLS» рассматривается, как работать с сертификатами X.509 в TLS и почему сертификаты так важны для TLS. Мы также узнаем, как проверить удаленный сертификат. Затем сделаем еще один шаг и научимся проверять действительность сертификата с помощью CRL и OCSP. Наконец, мы покажем, как использовать клиентский сертификат. Работа с сертификатами иллюстрируется с помощью команд и кода на C.

В главе 11 «Специализированные применения TLS» мы познакомимся с такими вопросами, как закрепление сертификатов, использование неблокирующего режима сетевого взаимодействия и установление TLS-подключений по нестандартным сокетам с помощью базовых объектов ввода-вывода OpenSSL (Basic Input-Output Object – BIO). Применение описанных методов иллюстрируется с помощью кода на C.

В главе 12 «Эксплуатация мини-УЦ» приводятся инструкции, как обустроить собственный миниатюрный удостоверяющий центр (УЦ) для управления сертификатами и построения инфраструктуры открытых ключей в организации. Эксплуатация мини-УЦ иллюстрируется с помощью команд и кода на C.

Как извлечь максимум пользы из этой книги

Вам понадобится установить OpenSSL на свой компьютер, чтобы выполнять примеры команд и кода на C. Если вы этого еще не сделали, то в главе 2 найдете подробные инструкции. Для сборки примеров потребуются совместимый со стандартом C11 компилятор C и компоновщик. Эти программы следует установить в соответствии с инструкциями в документации. Все примеры были протестированы на Kubuntu Linux 22.04 с использованием компилятора GNU C, компоновщика GNU (LD) и программы сборки GNU Make из вышеупомянутого дистрибутива Linux. Другие инструменты разработки, например LLVM Clang или Microsoft Visual C++, также должны быть совместимы с приведенными примерами кода.

ПО, рассматриваемое в этой книге	Системные требования
OpenSSL 3.0	• Linux, FreeBSD, macOS, Windows или любая другая ОС, поддерживаемая OpenSSL • Компилятор C, например GNU C Compiler • Компоновщик, например GNU Linker (LD) • Среда разработки на C/C++ или редактор кода (по собственному усмотрению) • Средство сборки, например GNU Make (факультативно)

Если вы пользуетесь цифровой версией этой книги, рекомендуем набирать код самостоятельно или брать его из репозитория книги на

GitHub (ссылка приведена в следующем разделе). Это позволит избежать потенциальных ошибок, связанных с копированием и вставкой кода.

Хотя объяснения средств OpenSSL и примеры кода иногда весьма детальны, эта книга задумана как руководство, а не замена документации OpenSSL. Если вас заинтересует какая-то функциональность OpenSSL, не рассмотренная в книге, обращайтесь к документации или к исходному коду OpenSSL либо просто экспериментируйте в своем коде!

Графические выделения

В этой книге применяется ряд соглашений о наборе текста.

`CodeInText`: код в тексте, имена таблиц базы данных, папок и файлов, расширения имен файлов, пути к файлам, данные, вводимые пользователем, и адреса в Твиттере. Например: «Открытые ключи пользователя SSH хранятся на сервере в файле `authorized_keys`».

Блок кода выглядит так:

```
if (pinned_server_cert)
    X509_free(pinned_server_cert);
if (pinned_server_cert_file)
    fclose(pinned_server_cert_file);
```

Ввод и вывод команд печатаются следующим образом:

```
$ ./tls-server 4433 server_keypair.pem server_cert.pem
*** Listening on port 4433
```

Полужирный: новые термины и важные слова, а также части пользовательского интерфейса. Так выделяются команды меню и текст в диалоговых окнах, например: «Уменьшается потребность в обслуживании, потому что не нужно создавать **запрос на подписание сертификата (CSR)** и обращаться в УЦ. Можно даже использовать самоподписанный сертификат».

Часть I

ВВЕДЕНИЕ

Любое путешествие начинается первым шагом. Наш первый шаг – понять, что такое OpenSSL, познакомиться с историей и узнать о том, что нового появилось в версии OpenSSL 3. Мы также дадим обзор других библиотек SSL/TLS, конкурирующих с OpenSSL.

Эта часть состоит всего из одной главы:

- главы 1 «OpenSSL и другие библиотеки SSL/TLS».

Глава 1

OpenSSL и другие библиотеки SSL/TLS

В настоящее время разработчикам, которые хотят включить в свои программы криптографические средства или SSL/TLS, доступно несколько библиотек. В этой главе мы сравним их и расскажем о сильных сторонах OpenSSL. Будут также рассмотрены история OpenSSL и новые возможности, появившиеся в версии OpenSSL 3.0.

В этой главе разбираются следующие темы:

- что такое OpenSSL;
- история OpenSSL;
- что нового в OpenSSL 3.0;
- сравнение OpenSSL с GnuTLS;
- сравнение OpenSSL с NSS;
- сравнение OpenSSL с Botan;
- сравнение OpenSSL с облегченными библиотеками TLS;
- сравнение OpenSSL с LibreSSL;
- сравнение OpenSSL с BoringSSL.

Что такое OpenSSL?

OpenSSL – комплект программ с открытым исходным кодом, включающий библиотеку **криптографических** средств и **SSL/TLS**, а также командные утилиты, которые пользуются этой библиотекой и предоставляют полезную функциональность в форме командной строки, например генерирование ключей шифрования и **сертификатов X.509**. Основная часть OpenSSL – именно библиотека, поэтому комплект ориентирован прежде всего на разработчиков ПО. Однако системные администраторы и инженеры DevOps также найдут командные утилиты OpenSSL весьма полезными.

Аббревиатура **SSL** означает **Secure Sockets Layer** (уровень безопасных сокетов). Это протокол, призванный обеспечить безопасное взаимодействие по небезопасным компьютерным сетям. Под небезопасной компьютерной сетью понимается сеть, в которой передаваемые данные можно прочитать или даже изменить, воспользовавшись вредоносным промежуточным сетевым узлом. Примером такой небезопасной сети является интернет. Безопасным называется взаимодействие, при котором передаваемые данные нельзя ни прочитать, ни изменить. SSL обеспечивает безопасность, применяя средства **симметричной** и **асимметричной** криптографии. Протокол SSL был изобретен в 1995 году корпорацией *Netscape Communications*, а в 2015-м его сменил протокол TLS. Аббревиатура **TLS** означает **Transport Layer Security** (безопасность транспортного уровня).

Комплект инструментов OpenSSL был создан еще до того, как протокол SSL был объявлен нерекомендуемым, поэтому в названии упоминается «SSL», а не «TLS».

Исторически OpenSSL распространялся на условиях *лицензии типа BSD*, но начиная с версии 3.0 распространяется на условиях *лицензии Apache 2.0*, которая также принадлежит семейству BSD. Эта лицензия разрешает использовать OpenSSL в приложениях как с открытым, так и с закрытым исходным кодом.

OpenSSL поддерживает великое множество **криптографических алгоритмов**, т. е. алгоритмов **симметричного** и **асимметричного шифрования**, **цифровой подписи, вычисления хеш-значений сообщений** и **обмена ключами**. OpenSSL поддерживает **сертификаты X.509**, протоколы **SSL**, **TLS** и **DTLS**, а также другие, менее распространенные криптографические технологии.

OpenSSL существует уже довольно давно и за годы разработки стал поддерживать много операционных систем. Первоначально OpenSSL разрабатывался для Unix-подобных ОС и до сих поддерживает различные варианты Unix, включая *GNU/Linux*, *BSD*, а также старые и новые коммерческие версии Unix, например *IBM AIX* и *macOS*. OpenSSL также поддерживает другие популярные операционные системы, в частности *Microsoft Windows*, мобильные ОС типа *Android* и *iOS* и даже такие старые и экзотические ОС, как MS-DOS и VMS.

За долгие годы разработки в библиотеку были включены многочисленные оптимизации, например *ассемблерные вставки* для большинства популярных процессорных архитектур, в т. ч. **x86**, **x86_64** и **ARM**. В настоящее время OpenSSL является одной из самых быстрых библиотек криптографии и TLS.

В силу своей универсальности, поддержки большого числа алгоритмов и операционных систем, а также быстродействия OpenSSL стала промышленным стандартом де-факто. OpenSSL так популярна, что другие библиотеки TLS включают так называемые **уровни совместимости с** OpenSSL, чтобы с ними можно было работать, применяя **интерфейсы прикладного программирования** (**API**) OpenSSL.

OpenSSL – очень популярная библиотека, но как она пришла к столь единодушной поддержке? Разберемся – и для этого поговорим об истории OpenSSL.

История OpenSSL

Библиотека OpenSSL основана на **библиотеке SSLeay**, написанной Эриком Эндрю Янгом. Суффикс *eay* в названии SSLeay как раз и означает *Eric Andrew Young*. Разработка SSLeay началась в 1995 году как реализация с открытым исходным кодом библиотеки SSL. В то время библиотека *NSS* еще не была доступна. Впоследствии к команде разработчиков присоединился Тим Хадсон. Но в 1998 году Эрика и Тима наняла на работу компания *RSA Corporation*, и у них не стало хватать времени на дальнейшую разработку SSLeay.

В 1998 году от SSLeay ответвился проект OpenSSL, т. е. OpenSSL стала преемником SSLeay. Членами-основателями стали Марк Кокс, Ральф Энгельшалл, Стивен Хенсон, Бен Лаури и Пол Саттон. Первая версия OpenSSL, с номером 0.9.1, была выпущена 23 декабря 1998 года, всего через неделю после перехода Эрика и Тима в RSA Corporation, когда работа над SSLeay фактически прекратилась.

За прошедшие годы многие люди и компании дописывали код и выполняли другие работы, связанные с OpenSSL. Список компаний, внесших свой вклад, впечатляет: Oracle, Siemens, Akamai, Red Hat, IBM, VMware, Intel и Arm, – и это далеко не все.

В настоящее время разработку OpenSSL координирует управляющий комитет OpenSSL, насчитывающий семь членов. Команда разработчиков ядра, имеющая право фиксации изменений, состоит примерно из 20 человек. И только двое посвящают все свое время работе над OpenSSL. Остальные занимаются этим либо в свободное время, либо по поручению компаний-работодателей.

История OpenSSL интересна, но какое будущее ей уготовано? Рассмотрим последние изменения, внесенные в комплект.

Что нового в OpenSSL 3.0?

Одним из главных изменений в **OpenSSL 3.0** стал порядок лицензирования. Лицензия программного проекта редко изменяется за время его существования. До выхода версии 3.0 проект OpenSSL распространялся на условиях лицензии *в духе BSD*, а начиная с версии 3.0 – на условиях лицензии *Apache License 2.0*.

В версии OpenSSL 3.0 произошли значительные изменения во внутренней архитектуре библиотеки. Архитектурные изменения не закончились и будут продолжены в OpenSSL 4.0. Введена концепция **поставщиков реализации операций OpenSSL**. **Поставщиком** называется часть кода, предоставляющая реализацию криптографических алгоритмов. Существующий криптографический код в OpenSSL доступен в основном через поставщиков *Default* и *Legacy*. **Движки** по-прежнему поддерживаются, но предпочтение отдается поставщикам. В OpenSSL 4.0 поддержка API движков может быть прекращена. Поддерживаются также сторонние поставщики, что позволяет

независимым разработчикам включать свои криптографические алгоритмы в OpenSSL.

Еще одна интересная возможность OpenSSL 3.0 – **ядерный TLS** (Kernel TLS – **KTLS**). Приложение, пользующееся KTLS, может создать специальный **TLS-сокет**, похожий на TCP-сокет. Затем OpenSSL выполняет процедуру **TLS-квитирования** и передает согласованный ключ шифрования и другие данные **ядру операционной системы** в виде **параметров TLS-сокета**. После этого передачей данных по протоколу TLS занимается код KTLS. Такой перенос нагрузки TLS на ядро может ускорить передачу данных в высоконагруженных системах, где производительность очень важна, особенно если ядро может воспользоваться аппаратными средствами ускорения шифра **Advanced Encryption Standard** (**AES**) и, следовательно, разгрузить главный процессор. Конечно, поддержка KTLS необходима как в самой библиотеке TLS, так и в ядре операционной системы. На момент написания этой книги KTLS поддерживали только Linux и FreeBSD.

Из других существенных изменений в OpenSSL 3.0 отметим следующие:

- поддержка **протокола управления сертификатами**;
- простейший HTTP/HTTPS-клиент;
- **режим счетчика с аутентификацией Галуа с синтетическим вектором инициализации** (**AES-GCM-SIV**) для шифра AES;
- новые алгоритмы вычисления **имитовставки** (**MAC**), в частности **GMAC** и **KMAC**;
- новые алгоритмы **функций формирования ключа** (**KDF**), в частности **SSKDF** и **SSHKDF**;
- новые высокоуровневые API, например **EVP_MAC**, **EVP_KDF** и **EVP_RAND**;
- низкоуровневые API объявлены нерекомендуемыми, предпочтение отдается новым высокоуровневым API;
- почищен код;
- изменен механизм обработки ошибок;
- старые небезопасные алгоритмы недоступны, если задан уровень безопасности по умолчанию;
- из командной программы `openssl` исключен интерактивный режим.

OpenSSL – зрелый пакет программ, т. е. самые важные средства в нем уже реализованы. Поэтому последние изменения не обещают пользователям много новой функциональности. В последней версии основное внимание уделено архитектурным улучшениям библиотеки.

Хотя OpenSSL – самая популярная библиотека криптографии и TLS, она не единственная. В следующих разделах мы сравним OpenSSL с конкурентами.

Сравнение OpenSSL с GnuTLS

GnuTLS – свободная библиотека TLS, созданная для нужд **проекта GNU**. Когда создавалась GnuTLS, большинство приложений в проекте GNU распространялись на условиях *лицензии GPL 2.0*, несовместимой со старой лицен-

зией OpenSSL. Авторы ПО, лицензированного на условиях GPL 2.0, должны были оговаривать исключения из лицензии, если хотели компоновать свой продукт с OpenSSL. GnuTLS изначально лицензировалась на условиях *LGPL 2.0*, поэтому не требовала таких исключений.

В настоящее время GnuTLS лицензируется на условиях *LGPL 2.1*. Эта лицензия разрешает использовать библиотеку в **свободном программном обеспечении с открытым исходным кодом** (free and open source software – **FOSS**). Также разрешено использовать библиотеку в проектах с закрытым кодом, но на определенных условиях, в частности только с **динамической компоновкой**.

GnuTLS не включает криптографических средств, арифметики больших чисел и некоторой другой функциональности, имеющейся в OpenSSL. Вместо этого GnuTLS *использует другие библиотеки*, входящие в **экосистему GNU**, которые предоставляют необходимую функциональность: **Nettle** для криптографии, **GMP** для арифметики больших чисел, **Libtasn1** для **ASN.1** (**Abstract Syntax Notation One** – абстрактная синтаксическая нотация версии 1) и т. д.

У GnuTLS есть интересная особенность – она поддерживает не только сертификаты X.509, но и сертификаты OpenPGP. В отличие от сертификата X.509, который подписывается выпускающей стороной в момент выпуска, сертификат OpenPGP поддерживает так называемую *сеть доверия*, может иметь несколько подписей, которые могут добавляться даже после выпуска сертификата. К сожалению, сертификаты OpenPGP не стали распространенными для **TLS-подключений**.

Если не считать поддержки сертификатов OpenPGP, GnuTLS и ее криптографическая библиотека, Nettle, поддерживают меньше криптографических алгоритмов, чем OpenSSL, а с точки зрения производительности работают чуть медленнее. Впрочем, все популярные алгоритмы GnuTLS и Nettle поддерживают.

Что выбрать, OpenSSL или GnuTLS? Я рекомендую остановиться на GnuTLS, если вы разрабатываете программу, распространяемую на условиях лицензии GPL; в противном случае выбирайте OpenSSL.

Еще одним конкурентом OpenSSL является библиотека NSS, которую мы обсудим в следующем разделе.

Сравнение OpenSSL с NSS

Network Security Services (**NSS**) – исторически первая библиотека SSL/TLS. Начиналась она как код обеспечения безопасности, встроенный в браузер *Netscape Navigator 1.0*, который был выпущен в 1994 году компанией Netscape Communications Corporation. Netscape также изобрела протокол SSL, предшественник TLS. Netscape Navigator сменил Netscape Communicator, за которым последовал браузер Mozilla и, наконец, Firefox. Вскоре после выпуска Netscape Communicator 4.0 в 1997 году код безопасности Netscape был выпущен в виде отдельной библиотеки, получившей название **Hard Core Library** (**HCL**). Впоследствии HCL была переименована в NSS.

Библиотека NSS распространялась на условиях лицензии *Mozilla Public License 2.0*, позволявшей разработчикам приложений использовать ее в программах с открытым и закрытым исходным кодом.

Известные приложения, в которых применяется библиотека NSS, – это приложения компании Mozilla Foundation, в частности Firefox и Thunderbird, некоторые приложения Oracle и Red Hat, а также некоторые приложения, входящие в открытые офисные пакеты LibreOffice и Evolution. Несмотря на то что все это – солидные и уважаемые компании и приложения, а сама библиотека NSS имеет долгую историю, она не снискала такой же популярности, как OpenSSL. Не знаю, почему. Некоторые говорят, что OpenSSL проще пользоваться, а документация у нее лучше.

Что выбрать, NSS или OpenSSL? Я рекомендую OpenSSL из-за лучшей документации и большего сообщества разработчиков и пользователей.

Как для NSS, так и для GnuTLS основной API написан на **C**. С библиотекой **Botan**, которую мы рассмотрим в следующем разделе, дело обстоит иначе.

Сравнение OpenSSL с Botan

Большинство библиотек TLS написаны на **C**, и основной API также рассчитан на C. Библиотека же Botan написана на **C++11** и в версии 3.0 переработана под стандарт **C++17**. Основной API Botan рассчитан на **C++**, но имеются также **привязки** к **C** и **Python**. Сторонние проекты предлагают еще привязки к Ruby, Rust и Haskell. Имеются экспериментальные привязки API к Java и Ocaml.

Стоит отметить, что библиотека Botan хорошо документирована. Botan распространяется на условиях **лицензии BSD с двумя оговорками**, которая разрешает использовать библиотеку в приложениях с открытым и закрытым исходным кодом.

Я рекомендую Botan разработчикам, которые хотят использовать C++ API и готовы смириться с менее популярной библиотекой, имеющей более узкое сообщество разработчиков. Если вам нужен C API, более производительная библиотека и широкое сообщество разработчиков, лучше выбрать OpenSSL.

Библиотека Botan потребляет больше памяти, чем OpenSSL, в отличие от облегченных библиотек TLS, которые мы обсудим далее.

Сравнение OpenSSL с облегченными библиотеками TLS

Есть несколько **облегченных библиотек TLS**. Они ориентированы на рынки встраиваемых устройств, в частности для **интернета вещей** и других небольших устройств, которыми люди пользуются, не считая их компьютерами. Сюда входят платежные терминалы в магазинчиках у дома, умные

часы, умные лампочки и разнообразные промышленные датчики. Обычно такие устройства оснащены *слабыми процессорами*, *небольшой оперативной памятью* и *небольшой внешней памятью*.

Своей облегченностью такие библиотеки TLS чаще всего обязаны *модульности*, т. е. возможности компилировать только необходимые модули, поддерживать меньшее число криптографических алгоритмов и протоколов и раскрывать менее развитый API; то есть они предоставляют разработчикам приложений меньше возможностей и настроек.

Из облегченных библиотек TLS лучше других известны **wolfSSL** (раньше yaSSL, или Yet Another SSL), **Mbed TLS** (раньше PolarSSL) и **MatrixSSL**.

Самой развитой из трех этих библиотек, похоже, является wolfSSL. Она поддерживает много криптографических алгоритмов, в т. ч. такие новые, как **ChaCha20** и **Poly1305**.

В последние версии wolfSSL включены *ассемблерные вставки*, поэтому она работает очень быстро. Согласно *тестам производительности* на сайте wolfSSL, производительность чистого шифрования часто не уступает, а в некоторых случаях значительно превосходит производительность OpenSSL – иногда аж на 50 %. Из другой имеющейся в интернете информации следует, что старые версии wolfSSL заметно медленнее OpenSSL. Если вас интересует производительность, то я рекомендую прогнать тесты производительности на целевом оборудовании, взяв последние версии библиотек. Объясняется это тем, что библиотеки постоянно развиваются и последние версии могут содержать больше оптимизаций.

Mbed TLS и MatrixSSL поддерживают заметно меньше криптографических алгоритмов.

WolfSSL и MatrixSSL имеют две лицензии: *GPL 2.0* и коммерческую. Лицензия GPL 2.0 позволяет использовать библиотеку только в GPL-совместимых FOSS-приложениях. Коммерческая лицензия позволяет применять библиотеку в приложениях *с закрытым исходным кодом*.

Mbed TLS распространяется на условиях лицензии *Apache License 2.0*, позволяющей использовать библиотеку в приложениях как с открытым, так с закрытым исходным кодом.

Хотя авторы облегченных библиотек TLS заявляют, что их творения значительно, примерно в 20 раз, меньше OpenSSL, в конфигурациях по умолчанию они вовсе не так малы. Ниже приведены размеры библиотек, откомпилированных на машине с архитектурой x86_64 и ОС Ubuntu 22.04:

- wolfSSL (версия 5.2.0, libwolfssl.so): 1768 КиБ;
- Mbed TLS (версия 2.28.0, libmbedtls.so + libmbedcrypto.so): 664 КиБ;
- MatrixSSL (версия 4.5.1, libssl_s.a + libcrypt_s.a + libcore_s.a): 1772 КиБ;
- OpenSSL (версия 3.0.2, libssl.so + libcrypto.so): 5000 КиБ.

Чтобы уменьшить размер, пользователь должен сам откомпилировать библиотеку и отключить все ненужные ему модули.

OpenSSL также имеет модульную структуру и позволяет исключать ненужные модули при компиляции. Однако это труднее, чем в случае облегченных библиотек. Какие-то модули OpenSSL трудно исключить, потому что от них зависят другие, хотя и не должны бы. А некоторые модули OpenSSL попросту

очень велики, особенно модуль `x509`, содержащий код для работы с сертификатами X.509. Таким образом, уменьшить размер откомпилированных файлов OpenSSL можно, но не так сильно, как в случае облегченных библиотек.

Используйте облегченную библиотеку TLS, если OpenSSL не помещается в память целевого устройства. В противном случае пользуйтесь полномасштабной библиотекой TLS, например OpenSSL.

Последние две библиотеки, которые мы рассмотрим, ответвились от самой OpenSSL. Разберемся, почему первозданная OpenSSL показалась кому-то неподходящей.

Сравнение OpenSSL с LibreSSL

LibreSSL – ответвление от OpenSSL, созданное в 2014 году в рамках проекта OpenBSD Project как реакция на печально знаменитую уязвимость Heartbleed, обнаруженную в OpenSSL. При разработке LibreSSL ставилась цель повысить *безопасность* и *удобство сопровождения* библиотеки ценой удаления старых, непопулярных и больше не считающихся стойкими криптографических алгоритмов и других средств.

OpenBSD – операционная система на базе Unix, одна из семейства систем BSD, ориентированная на безопасность. Создатели OpenBSD разрабатывают не только ядро операционной системы и утилиты. Еще один известный их проект – OpenSSH. Другие проекты, запущенные как приложения для OpenBSD, – OpenNTPD, OpenSMTPD и т. д. Сейчас к ним добавилась и библиотека LibreSSL.

После разветвления разработчики LibreSSL исключили из OpenSSL много оригинального кода, который сочли устаревшим или небезопасным. Они заявляли, что уже в первую неделю сократили код OpenSSL примерно наполовину. Они также добавили несколько новых криптографических алгоритмов, в частности Advanced Encryption Standard в режиме счетчика с аутентификацией Галуа (AES-GCM) и ChaCha-Poly1305. Кроме добавления и удаления, существующий код был частично переработан и укреплен.

Несмотря на всю нацеленность LibreSSL на повышение безопасности, в ней самой обнаружены некоторые уязвимости, не затронувшие OpenSSL, например CVE-2017-8301.

За последние несколько лет команда OpenSSL также проделала работу над повышением безопасности и пригодности к сопровождению, а еще удалила или объявила нерекомендуемой часть кода – хотя и не так много, как в LibreSSL. Однако много кода и новых возможностей было и добавлено.

У OpenSSL гораздо больше ресурсов для развития, чем у LibreSSL, а это означает, что темп продвижения выше. Например, с конца 2018 по начало 2021 года в LibreSSL было включено примерно *1500 исправлений и дополнений от 36 разработчиков*. За то же время в OpenSSL было включено более *5000 исправлений и дополнений от 276 разработчиков*. Помимо разработчиков ядра, OpenSSL получает код от больших компаний, таких как Oracle, Red Hat, IBM и др.

Хотя основная задача LibreSSL – получить библиотеку TLS для проекта OpenBSD, она также ставит целью поддерживать другие платформы и сохранять совместимость на уровне API с OpenSSL. На момент написания книги API совместим с OpenSSL 1.0.1, но включает не все новейшие API, появившиеся начиная с версии OpenSSL 1.0.2.

Сразу после ответвления от OpenSSL библиотека LibreSSL стала библиотекой TLS по умолчанию в OpenBSD, но на большинстве других платформ готовность принять ее невелика. В большинстве дистрибутивов Linux попрежнему используется OpenSSL, некоторые все же решили попробовать LibreSSL в качестве общесистемной библиотеки, но впоследствии отказались от этого. Большинство поставщиков коммерческих приложений, которые традиционно использовали OpenSSL, решили держаться этого курса и дальше.

В конкуренции с LibreSSL выигрывает OpenSSL. Я рекомендую выбирать OpenSSL, если только вы не пишете программы специально для сообщества OpenBSD, а в таком случае следует предпочесть LibreSSL.

В следующем разделе мы рассмотрим еще одно ответвление от OpenSSL, созданное могущественной корпорацией Google.

Сравнение OpenSSL с BoringSSL

BoringSSL – еще одно разветвление OpenSSL, о котором было объявлено с 2014 года. BoringSSL была создана для нужд корпорации Google. В течение многих лет Google сопровождала собственные дополнения для OpenSSL, используемые в различных продуктах Google, в частности *Chrome*, *Android* и серверной инфраструктуре. Наконец, они решили *разветвить* OpenSSL и поддерживать свою ветвь в качестве отдельной библиотеки.

Как и в случае LibreSSL, Google удалила из BoringSSL значительную часть оригинального кода OpenSSL, отвечавшего за поддержку старых и непопулярных алгоритмов и средств. Google также добавила функциональность, отсутствовавшую в OpenSSL. Например, **CRYPTO_BUFFER** позволяет избавиться от дубликатов сертификатов X.509 в памяти, сократив тем самым ее потребление. Кроме того, она дает возможность исключить входящий в OpenSSL код X.509 и ASN.1 из приложения, если OpenSSL компонуется с приложением статически. А надо сказать, что код X.509 составляет заметную часть OpenSSL.

В отличие от LibreSSL, BoringSSL не ставит себе целью добиться **совместимости на уровне API** с OpenSSL или даже с предыдущими версиями BoringSSL. Google хочет оставить за собой свободу менять API библиотеки по собственному усмотрению. Это разумно, потому что Google контролирует как саму BoringSSL, так и основные программные проекты, в которых она используется, поэтому нетрудно синхронизировать изменения API в BoringSSL и этих проектах. Если не требовать стабильности API, то можно освободить ресурсы, которые иначе пришлось бы зарезервировать для сопровождения старых API.

Но это также означает, что если кто-то за пределами Google захочет использовать BoringSSL, то должен быть готов к *несовместимым изменениям* в API библиотеки, причем в самый неподходящий момент. Это очень неудобно для разработчиков, использующих библиотеку. Понимая это, Google заявляет, что хотя BoringSSL – проект с открытым исходным кодом, он не предназначен для общего пользования.

На мой взгляд, код BoringSSL открыт в основном для сторонних соразработчиков таких проектов Google, как Chrome и Android.

Я не рекомендую использовать BoringSSL в своих приложениях по причине **нестабильности API**. Кроме того, OpenSSL может похвастаться более широкой функциональностью, лучшей документацией и значительно более крупным сообществом.

На этом мы заканчиваем обзор некоторых конкурентов OpenSSL. Теперь вы понимаете основные различия между популярными библиотеками TLS и знаете, какую библиотеку в какой ситуации лучше использовать.

Резюме

В этой главе мы узнали, что такое библиотека OpenSSL и зачем ее использовать. Мы бегло рассмотрели историю OpenSSL и заглянули в ее будущее. Мы поговорили о других библиотеках TLS, конкурирующих с OpenSSL, и описали их сильные и слабые стороны.

Теперь вы должны лучше понимать, какую библиотеку TLS использовать в конкретной ситуации. Если ситуация стандартная, то я рекомендую выбирать OpenSSL, потому что многие другие поступили так же, а это значит, что OpenSSL – самая популярная библиотека и промышленный стандарт де-факто.

Зная, почему следует выбирать OpenSSL, пора узнать, как ей пользоваться. В следующих нескольких главах, носящих практический характер, мы научимся использовать мощные средства OpenSSL – от симметричного шифрования до создания TLS-подключений с сертификатами X.509.

Часть II

СИММЕТРИЧНАЯ КРИПТОГРАФИЯ

В этой части мы узнаем о симметричной криптографии и сопутствующих технологиях, таких как криптографически стойкие хеш-значения сообщений, коды аутентичности сообщений (имитовставки) и генерирование симметричных криптографических ключей на основе паролей. Здесь приводится обзор симметричных шифров, таких как AES и ChaCha, а также алгоритмов вычисления хеш-значений сообщений (например, SHA-256), поддерживаемых OpenSSL. Использование всех технологий иллюстрируется примерами команд и кода на C. Мы научимся инициализировать и очищать OpenSSL и напишем первые программы, в которых OpenSSL будет использоваться.

Глава 2

Симметричное шифрование и расшифрование

В этой главе мы узнаем о важных понятиях симметричного шифрования – шифр, режим шифрования и дополнение. Мы дадим обзор современных шифров, режимов шифрования и типов дополнения, а также порекомендуем, какую технологию в какой ситуации использовать. Применение технологий иллюстрируется примерами кода. Это первая глава, содержащая примеры кода, поэтому мы заодно научимся инициализировать старые версии библиотеки OpenSSL, на случай если вам когда-нибудь придется компоновать код с одной из таких версий.

В этой главе рассматриваются следующие темы:

- что такое симметричное шифрование;
- обзор симметричных шифров, поддерживаемых OpenSSL;
- режимы работы блочных шифров;
- дополнение в блочных шифрах;
- как генерируются ключи симметричного шифрования;
- скачивание и установка OpenSSL;
- применение AES для шифрования и расшифрования в командной строке;
- инициализация и очистка библиотеки OpenSSL;
- компиляция и компоновка с OpenSSL;
- программное шифрование шифром AES;
- программное расшифрование AES.

Технические требования

Эта глава содержит команды, выполняемые из командной строки, а также исходный код на C, который можно собрать и выполнить. Для выполнения

команд понадобятся утилита `openssl` и динамические библиотеки OpenSSL. Для сборки кода на C понадобятся динамические или статические библиотеки OpenSSL, заголовочные файлы библиотек, компилятор C и компоновщик. Если компоненты OpenSSL еще не установлены, то из этой главы вы узнаете, как их установить.

В этой главе мы напишем несколько примеров программ, чтобы на практике закрепить то, чему научились. Полный исходный код программ можно найти на GitHub по адресу https://github.com/PacktPublishing/Demystifying-Cryptography-with-OpenSSL-3/tree/main/Chapter02.

Что такое симметричное шифрование

Симметричное шифрование применяется для шифрования данных **ключом шифрования**. Отметим, что ключ шифрования – не то же, что пароль, но его можно сформировать на основе пароля. Как это делается, объяснено в главе 5.

Симметричное шифрование называется симметричным, потому что для шифрования и расшифрования используется один и тот же ключ. Имеется также **асимметричное шифрование**, при котором ключи шифрования и расшифрования (открытый и закрытый) различны. Асимметричное шифрование рассматривается в главе 6.

Чтобы что-то симметрично зашифровать, нужно использовать **алгоритм симметричного шифрования**. Такие алгоритмы называются также **шифрами**. Отметим, что слово *шифр* трактуется широко и может относиться к любому алгоритму шифрования или деятельности, как-то связанной с шифрованием, например: алгоритмы симметричного и асимметричного шифрования, вычисления хеш-значений сообщений и даже алгоритмы, применяемые на этапе установления TLS-подключения, скажем `ECDHE-ECDSA-AES256-GCM-SHA384`. В этой книге слово *шифр* будет обозначать разные вещи в зависимости от контекста. Прямо сейчас контекстом является симметричное шифрование, поэтому под *шифром* понимается алгоритм симметричного шифрования.

Полезные данные, подвергающиеся шифрованию, называются **открытым текстом**, даже если это не текст. Изображения, музыка, видео, архивы, базы данных – в общем, все, что вы хотите зашифровать, – это открытый текст с точки зрения шифрования.

Результат шифрования, т. е. зашифрованные данные, называется шифртекстом. Почти никогда шифртекст не напоминает текст, но тем не менее применяется именно такая терминология.

Таким образом, можно сказать, что шифрование преобразует открытый текст в шифртекст, а расшифрование – шифртекст обратно в открытый текст.

Чтобы зашифровать открытый текст, нужно выполнить следующие действия:

1. Выбрать алгоритм шифрования (шифр).
2. Сгенерировать ключ шифрования.

3. Сгенерировать **вектор инициализации** (**IV**), или **синхропосылку**.
4. Выбрать **режим работы шифра**.
5. Выбрать **тип дополнения**.
6. Зашифровать открытый текст, применив выбранный шифр с выбранным ключом и типом дополнения в выбранном режиме работы.

В зависимости от шифра и его режима некоторые шаги могут быть опущены. Например, для потоковых шифров и некоторых режимов работы блочных шифров дополнение не нужно.

У человека, плохо знакомого с криптографией, может возникнуть вопрос: «Зачем так много параметров? Когда я шифровал архивы, мой архиватор просил задать только пароль и не спрашивал ни о шифре, ни о ключе, ни о IV, ни о режиме, ни о типе дополнения!» Да просто архиватор и любая другая простая и удобная программа решают все эти вопросы за кулисами, принимая большинство решений за пользователя. И ключ шифрования генерируется по заданному пользователем паролю.

Мы же все-таки поговорим немного о базовых понятиях шифрования, чтобы принимать решения осознанно и понимать, как все параметры работают совместно. Начнем с изучения алгоритмов шифрования (или шифров).

Обзор симметричных шифров, поддерживаемых OpenSSL

В этом разделе мы дадим обзор симметричных алгоритмов шифрования, поддерживаемых OpenSSL, но сначала нужно ввести некоторые понятия, которые помогут разобраться в различиях между шифрами, их свойствами, преимуществами и недостатками. Симметричные шифры делятся на две категории: блочные и потоковые.

Сравнение блочных и потоковых шифров

Блочные шифры работают с блоками данных. Например, в популярном шифре **Advanced Encryption Standard** (**AES**) **размер блока** составляет 128 бит, т. е. данные шифруются блоками по 128 бит. Если объем данных больше размера блока, то данные разбиваются на блоки нужного размера. Если длина **открытого текста** не кратна размеру блока, то последний блок обычно дополняется до размера блока с учетом выбранного **типа дополнения**. Таким образом, в большинстве **режимов работы блочного шифра** длина **шифртекста** кратна размеру блока.

Что происходит во время шифрования данных блочным шифром? На вход алгоритма шифрования подается блок открытого текста (например, 16 байт), ключ шифрования и, возможно, какие-то данные, зависящие от режима шифрования (например, предыдущий блок шифртекста или IV). Отдельные

байты открытого текста помещаются в список или матрицу. Затем над этими байтами, байтами ключа шифрования и дополнительными, зависящими от режима байтами данных выполняются различные логические и арифметические операции: **поразрядный циклический сдвиг**, **исключающее или** (**XOR**), сложение или вычитание либо обмен полных или частичных байтов в списке либо матрице. Результатом этих операций является блок шифртекста того же размера, что входной блок открытого текста. При расшифровании входом являются блок шифртекста, ключ шифрования и данные, зависящие от режима работы. Операции, проделанные на этапе шифрования, выполняются в обратном порядке, и в результате блок шифртекста преобразуется обратно в блок открытого текста.

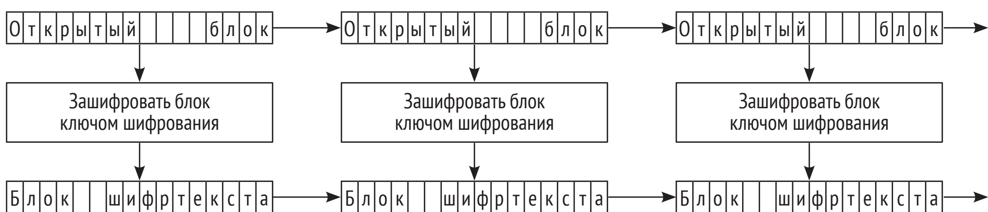


Рис. 2.1 ❖ Как работают блочные шифры

Потоковые шифры оперируют не блоками, а отдельными байтами или даже битами данных. Это значит, что потоковые шифры не нуждаются в дополнении. У них также нет режима работы, хотя некоторые исследователи пытаются ввести режим и для потоковых шифров. Благодаря таким характеристикам потоковые шифры проще использовать, особенно для поступающих потоком данных, их проще реализовать, и работают они быстрее блочных. Однако криптографы полагают, что в общем случае стойкость существующих потоковых шифров ниже, чем блочных, да и ослабить ее легко, допустив ошибку при реализации.

А как работают потоковые шифры? В процессе шифрования и расшифрования потоковый шифр генерирует так называемый **поток псевдослучайных цифр шифра**, или **гамму**, взяв за основу ключ шифрования и **IV** или **одноразовое число** (nonce, т. е. number used once). Шифр принимает байты ключа шифрования и IV и выполняет над ними логические и математические операции с целью породить бесконечный поток кажущихся случайными байтов. Это и есть гамма. Можно сказать, что при порождении гаммы потоковый шифр работает как **генератор псевдослучайных чисел** (**ГПСЧ**), который **инициализируется** ключом шифрования и IV. На этапе шифрования поток шифртекста порождается путем объединения гаммы с потоком открытого текста. Обычно к байтам гаммы и открытого текста просто применяется операция XOR. На этапе расшифрования порождается та же гамма, которая объединяется с потоком шифртекста. Если *поток шифртекста* был порожден применением XOR к *гамме* и *потоку открытого текста*, то для восстановления потока открытого текста нужно применить ту же самую операцию XOR, но на этот раз к *гамме* и *потоку шифртекста*.

Для порождения той же гаммы на этапе расшифрования следует использовать те же самые ключ шифрования и IV.

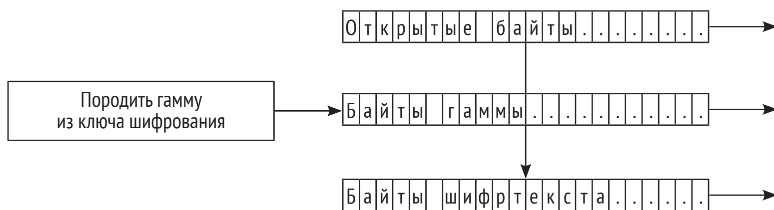


Рис. 2.2 ❖ Как работают потоковые шифры

Стойкость симметричного шифра

Криптостойкость, или просто стойкость **шифра**, криптографы обычно измеряют в битах. Например, если противнику для вскрытия шифра нужно выполнить 2^{256} вычислительных операций, то стойкость шифра равна 256 бит.

Максимально возможная стойкость шифра равна длине используемого ключа шифрования. Например, если используется 256-битовый ключ, то стойкость шифра не может превышать 256 бит. Это довольно просто доказать. Представим, что у противника имеется какой-то шифртекст, но нет ни ключа, ни соответствующего открытого текста, однако он может отличить правильный открытый текст от случайных данных. Цель противника – восстановить ключ шифрования или открытый текст. Если **длина ключа** равна 256 бит, то всего существует 2^{256} возможных ключей. Противник может произвести **полный перебор**, или **исчерпывающий поиск**, т. е. попробовать для расшифрования все возможные ключи. Тогда ему придется выполнить не более 2^{256} попыток, т. е. 2^{256} вычислительных операций.

Здесь вычислительная операция необязательно должна быть простейшей, скажем машинной командой; это всего лишь полная попытка дешифрировать небольшой объем шифртекста, например один блок. Важно, что количество вычислительных операций зависит от длины ключа. Расшифрование одного блока шифртекста на современных процессорах производится быстро и дешево, но на то, чтобы выполнить 2^{256} таких операций, уйдут миллиарды лет. Стойкость шифрования основана на том, что для расшифрования шифртекста или восстановления ключа противник должен будет выполнить практически неосуществимое число операций.

Важно понимать, что вычислительная сложность экспоненциально возрастает с увеличением длины ключа, т. е. вскрытие шифра с 256-битовым ключом займет не в 2, а в 2^{128} раз больше времени, чем вскрытие шифра с 128-битовым ключом. Чем длиннее ключ, тем больше времени занимают также обычные операции, но в случае симметричного шифрования это не важно. Поэтому имеет смысл выбирать более длинные ключи.

Цель большинства шифров – обеспечить такую стойкость, какую позволяет ключ. На практике криптоаналитикам часто удается найти более хитро-

умные способы атаковать шифр, чем полный перебор. Поэтому стойкость шифра может оказаться ниже, например не 256, а 224 бит. Шифр считается стойким, если после применения самой хитроумной из известных атак число все равно остается достаточным.

Сколько битов стойкости достаточно?

По состоянию на 2021 год в профессиональной среде сложилось общее мнение, подтвержденное и **Национальным институтом стандартов и технологий** (**NIST**), о том, что:

- 112 бит должно быть достаточно до 2030 года;
- 128 бит должно быть достаточно до следующего революционного прорыва в технологии или в математике.

Одного такого технологического прорыва ожидают в ближайшие годы. Это **квантовые вычисления**. Квантовые компьютеры уже существуют, но они очень дорогие и недостаточно мощные для вскрытия шифров. Ожидается, что квантовые вычисления способны *вдвое уменьшить* число битов стойкости для **симметричных криптографических алгоритмов**. Это значит, что шифры, стойкость которых раньше составляла 256 бит, будет иметь только 128-битовую стойкость, а стойкость шифров со 128 битами сократится до 64.

Как уже было сказано, криптоаналитики постоянно ищут более хитроумные способы вскрыть криптографические алгоритмы. Поэтому эффективная стойкость любого шифра обычно несколько меньше длины ключа.

Популярные симметричные шифры поддерживают длину ключа до 256 бит.

С учетом всего сказанного я рекомендую всюду, где возможно, использовать 256-битовые ключи шифрования и стойкий симметричный шифр. Такой стойкости должно быть достаточно на обозримое будущее, даже с учетом исследований в области криптоанализа и прогресса в области квантовых вычислений.

Обзор шифра AES

AES – самый популярный из современных симметричных шифров. Если вы не уверены, какой симметричный шифр использовать, смело выбирайте этот. Он быстрый и стойкий. AES – блочный шифр с размером блока 128 бит. Существуют три варианта AES с длинами ключей 128, 192 и 256 бит. Они называются соответственно AES-128, AES-192 и AES-256.

Было проведено много исследований стойкости AES и предпринято немало попыток найти способ вскрытия, более эффективный, чем полный перебор. AES устоял против всех таких попыток. Лучшим из известных атак удалось уменьшить стойкость AES всего на 2 бита: до 126, 190 и 254 бит.

Стоит отметить, что в современных процессорах x86 и x86_64 имеются команды аппаратного ускорения AES, в совокупности называемые **Advanced Encryption Standard New Instructions** (**AES-NI** – расширение системы ко-

манд AES), которые делают и так уже быстрый шифр AES еще быстрее. Некоторые процессоры ARMv8 поддерживают **криптографическое расширение**, также ускоряющее работу AES.

В AES применяется алгоритм шифрования **Rijndael**, который до стандартизации поддерживал больше размеров блоков и длин ключа. Поэтому можно сказать, что AES является подмножеством Rijndael. Многие употребляют термины *AES* и *Rijndael* как синонимы. Алгоритм Rijndael разработан двумя бельгийскими криптографами, *Винсентом Рэйманом* (Vincent Rijmen) и *Йоанном Дайменом* (Joan Daemen). Название *Rijndael* образовано из частей их имен, *Rij* и *Dae*.

Почему в названии AES есть слово *Standard* (стандартный)? Потому что AES был стандартизирован в 2001 году институтом NIST в качестве стандартного алгоритма шифрования для секретной коммерческой и правительственной информации (в США). Отметим, что AES используется не только для шифрования правительственной информации в США, но является также самым популярным общедоступным симметричным шифром в мире.

Обзор шифров DES и 3DES

Предыдущим стандартом шифрования был **Data Encryption Standard** (**DES**), предшественник AES. Это блочный шифр с размером блока 64 бит и длиной ключа 56 бит. И размер блока, и длина ключа слишком малы и не обеспечивают стойкости шифра. Криптоаналитики еще и снизили стойкость DES до 39–41 бит. На практике ключ DES можно восстановить на современном персональном компьютере потребительского класса за несколько дней или даже часов.

3DES, или **Triple DES**, – шифр, основанный на обычном DES. По сути дела, 3DES сводится к троекратному применению DES с двумя или тремя разными ключами. Считается, что стойкость 3DES составляет 112 бит для варианта с тремя ключами и всего 80 бит для варианта с двумя ключами.

Не рекомендуется использовать шифры DES и 3DES в новых приложениях. Однако DES находился в употреблении много лет, и в некоторых организациях до сих пор могут храниться данные, зашифрованные DES. В каких-то унаследованных системах сетевая безопасность, возможно, реализована с применением 3DES, но не AES, поэтому 3DES может быть необходим для обеспечения интероперабельности. Если старые нестойкие шифры нужны для поддержки унаследованных систем, то настоятельно рекомендуется ограничить их использование внутренними сетями и как можно скорее перейти на более современные шифры.

3DES в три раза медленнее обычного DES. Шифр DES и сам довольно медленный по сравнению с AES и другими конкурентами. На современных рабочих станциях медленность DES или 3DES не играет существенной роли, но если сервер должен обслуживать много клиентов, применяя медленный шифр, то такое торможение может стать заметным. Это еще одна причина перейти на более быстрый и стойкий шифр.

Обзор шифра RC4

RC4 – старый потоковый шифр, допускающий использование ключей длиной от 40 до 2048 бит. В прошлом он был очень популярен благодаря своей скорости и простоте реализации, но с годами в алгоритме обнаружилось много дефектов, и теперь он считается нестойким. Например, одна из атак снижает стойкость RC4 с 128-битовыми ключами всего до 26 бит.

Интересно, что на протяжении части 2013 года RC4 был основным шифром SSL/TLS, потому что все блочные шифры, применявшиеся в SSL 3.0 и TLS 1.0, были скомпрометированы знаменитой атакой **Browser Exploit Against SSL/TLS** (**BEAST**), которая, однако, не распространялась на потоковый RC4.

RC4 страдает той же проблемой, что DES. Этот шифр долгое время был популярен, и в некоторых организациях могли сохраниться зашифрованные им данные. Возможно также, что еще эксплуатируются унаследованные системы, не поддерживающие современных алгоритмов шифрования. Таким организациям можно порекомендовать те же меры, что для DES: переходить на современные системы и шифры, а в переходный период ограничить использование данных или сетевых потоков, защищенных шифром RC4, только внутренними сетями.

На момент написания книги популярные современные браузеры все еще поддерживают 3DES, чтобы иметь возможность работать со старыми серверами, но ни DES, ни RC4 больше не поддерживаются.

Обзор шифра ChaCha20

ChaCha20 – современный потоковый шифр, стойкий и быстрый. ChaCha20 можно применять с 128- или 256-битовым ключом. В основном варианте шифра используется 256-битовый ключ, и OpenSSL поддерживает только этот вариант. На момент написания книги неизвестны атаки, способные понизить стойкость полного варианта шифра ChaCha20 с 20 раундами, следовательно, его стойкость совпадает с длиной ключа – 256 бит.

Шифр ChaCha20 разработал *Даниэль Дж. Бернштейн* (Daniel J. Bernstein – *DJB*), знаменитый криптограф и криптоаналитик. DJB известен также другими проектами, в т. ч. **qmail**, **djbdns**, **Poly1305** и **Curve25519**. ChaCha20 – вариант шифра **Salsa20** того же автора, оптимизированный с целью повышения производительности. ChaCha20 стал популярен, когда компания Google начала использовать его в своем браузере Chrome наряду с алгоритмом вычисления **имитовставки** (**MAC**) Poly1305. Затем поддержка ChaCha20 и Poly1305 была добавлена в OpenSSH. Став популярным потоковым шифром, ChaCha20 сменил в этом качестве RC4.

Как уже отмечалось, ChaCha20 – быстрый шифр. Его программная реализация быстрее, чем AES. Аппаратно ускоренная реализация AES на x86 или x86_64 и ARMv8 работает быстрее ChaCha20, но такое ускорение доступно не на всех процессорах. Например, процессоры ARM младших моделей, применяемые в мобильных и других устройствах, не имеют такой оптимизации

и не обладают такой вычислительной мощностью, как x86_64 в настольных компьютерах. Поэтому ChaCha20 может оказаться предпочтительнее AES на устройствах низшей ценовой категории, особенно для шифрования сетевых потоков данных.

Недостатком ChaCha20, хотя и не очень существенным, является то, что в нем используется **счетчик блоков** длиной всего 32 бит. Из-за этого шифр может считаться стойким только при работе с непрерывными открытыми текстами, не превышающими 256 ГиБ. Этого должно быть достаточно для сетевых потоков данных, но может не хватить для шифрования больших данных. Обходной путь – разбивать очень большие открытые тексты на более короткие порции, длиной не более 256 ГиБ, и перед шифрованием каждой порции сбрасывать **одноразовое число** и счетчик блоков.

Обзор других симметричных шифров, поддерживаемых OpenSSL

OpenSSL поддерживает и другие, менее популярные шифры.

Некоторые из них, например **Blowfish**, **CAST5**, **ГОСТ89**, **IDEA**, **RC2** и **RC5**, – блочные шифры с размером блока 64 бит или переменным размером блока, который в OpenSSL устанавливается равным 64 бит. Эти шифры не рекомендуется использовать для очень длинных шифртекстов (длиннее 32 ГиБ) из-за **коллизионных атак** и **парадокса дней рождения**. Чтобы решить проблему, нужно разбивать длинные открытые тексты на порции, меньшие 32 ГиБ, и для каждой порции использовать разные ключи шифрования и IV. Алгоритмы с 64-битовыми блоками довольно старые. Многие из них разрабатывались еще в 1990-е, а некоторые даже в 1970-е годы.

Другие алгоритмы, например **ARIA**, **Camellia**, **SEED** и **SM4**, – блочные шифры со 128-битовыми блоками. При таком размере блока можно шифровать очень длинные открытые тексты (до 256 экзабайт), не опасаясь коллизионных атак.

Некоторые из этих менее популярных алгоритмов хороши, другие имеют технические недостатки по сравнению с AES. Многие старые алгоритмы значительно, до 10 раз, медленнее AES, но большинство из тех, что поддерживаются OpenSSL, по-прежнему стойкие. Мы не станем обсуждать все эти шифры подробно, но описанные ниже достойны упоминания.

Менее популярные шифры можно разбить на несколько категорий.

Национальные шифры

Национальные шифры разрабатываются и стандартизируются в конкретной стране. Обычно криптосообщество этой страны испытывает гордость за свой национальный шифр. Зачастую такие шифры хороши, но не очень популярны за пределами страны происхождения.

Шифр CAST5, известный также под названием **CAST-128**, разработан канадскими криптографами *Карлайлом Адамсом* и *Стаффордом Таваресом*

в 1996 году. Название *CAST* образовано инициалами авторов шифра, но также напоминает о случайности, что имеет смысл, поскольку хороший шифр должен порождать шифртекст, не отличимый от случайных данных. В шифре CAST5 используются 128-битовые ключи. Неизвестно ни одной успешной атаки на полный CAST5 с 16 раундами. Поэтому стойкость совпадает с длиной ключа и равна 128 бит. Интересно, что CAST5 долгое время был симметричным шифром по умолчанию в программе **GNU Privacy Guard** (**GnuPG**), пока его не заменили на AES. В течение некоторого времени он также был шифром по умолчанию в оригинальной программе **Pretty Good Privacy** (**PGP**). CAST5 – один из стандартных шифров, используемых канадским правительством. От вас могут потребовать его использования, если ваша программа должна взаимодействовать с компьютерными системами государственных организаций Канады.

ГОСТ89[1], или **Магма**, – старый государственный стандартный шифр, применявшийся в СССР, а затем в России. Он был разработан КГБ в 1970-х годах и впервые опубликован в 1989 году. На смену ГОСТ89 пришел шифр **ГОСТ2015**[2], или **Кузнечик**, но ГОСТ89 при этом не был объявлен устаревшим. Аббревиатура **ГОСТ** означает **ГОсударственный СТандарт**. В ГОСТ89 применяются 256-битовые ключи. Шифр ГОСТ89 был подвергнут скрупулезному криптоанализу, и лучшая известная теоретическая атака снижает его стойкость до 178 бит ценой потребления гигантских, недоступных на практике объемов оперативной и внешней памяти. Некоторые криптоаналитики указывают, что для ГОСТ89 имеются небезопасные **S-блоки**, но в OpenSSL включен только ГОСТ89 с безопасными блоками. Поскольку ГОСТ89 – один из государственных стандартов в Российской Федерации, от вас могут потребовать его использования в программах, взаимодействующих с компьютерными системами российских государственных учреждений.

SEED и ARIA – южнокорейские шифры. Оба применяются в Южной Корее, но больше почти нигде. SEED был разработан в 1998-м, а ARIA – в 2003 году. Оба являются блочными шифрами с размером блока 128 бит. ARIA основан на AES и поддерживает те же длины ключей – 128, 192 и 256 бит. В SEED используются 128-битовые ключи. На момент написания книги неизвестно ни о каких практически осуществимых атаках на SEED и ARIA – поэтому считается, что их стойкость совпадает с длиной ключа. ARIA оптимизирован для аппаратной реализации, в т. ч. в оборудовании низшей ценовой категории. Его программная реализация работает от полутора до двух раз медленнее AES. SEED и ARIA – национальные стандарты Южной Кореи. От вас могут потребовать их использования в программах, взаимодействующих с компьютерными системами южнокорейских государственных организаций.

Camellia – 128-битовый блочный шифр, разработанный в 2000 году в Японии компаниями NTT и Mitsubishi Electric. Назван в честь цветка *Camellia japonica* (камелия японская). Шифр Camellia по структуре аналогичен AES, его стойкость и производительность сравнимы с AES. Расширение системы команд AES-NI для процессоров x86_64 можно использовать для аппаратного

[1] Правильное обозначение ГОСТ 28147–89. – *Прим. перев.*
[2] Правильное обозначение ГОСТ Р 34.13–2015. – *Прим. перев.*

ускорения Camellia. Многие результаты криптоанализа для AES применимы и к Camellia. Как и AES, Camellia поддерживает ключи длиной 128, 192 и 256 бит. Шифр Camellia запатентован, но доступен по бесплатной лицензии. Camellia – один из стандартизованных в Японии шифров. От вас могут потребовать его использования в программах, взаимодействующих с компьютерными системами японских государственных организаций.

SM4 – еще один шифр, пришедший с Востока, на этот раз из Китая. Год его разработки не раскрывается, поскольку в течение некоторого времени шифр был секретным, гриф секретности был снят в 2006 году. В шифре используются 128-битовые блоки и 128-битовые ключи. Как и для Camellia, возможно аппаратное ускорение SM4 с помощью расширения системы команд AES-NI для процессоров x86_64. Было предпринято несколько криптоаналитических попыток вскрытия, но до сих пор неизвестно ни одной успешной атаки на полный шифр SM4 с 32 раундами, т. е. его стойкость остается на уровне 128 бит. SM4 – китайский национальный стандарт и используется в беспроводных **локальных сетях** (**LAN**) в Китае. От вас могут потребовать его использования в программах, взаимодействующих с компьютерными системами китайских государственных организаций.

Семейство шифров RC

Семейство RC – набор шифров, разработанных знаменитым криптографом *Роном Ривестом*. RC означает «Ron's Code» (код Рона) или Rivest Cipher (шифр Ривеста). RC2, предшественник RC4, – старый шифр, разработанный в 1987 году. RC2 уязвим для атак, связанных с ключом. Этот шифр не привлекал пристального внимания независимых криптоаналитиков, но криптографическое сообщество скептически относится к его стойкости. Еще один недостаток RC2 состоит в том, что, будучи шифром с переменной длиной ключа, он поддерживает небезопасные короткие ключи. RC5 был разработан в 1994 году и замечателен простотой реализации. Это шифр с переменной длиной ключа (от 0 до 2040 бит) и переменным размером блока (32, 64 или 128 бит). OpenSSL поддерживает для RC5 только 64-битовые блоки и ключи длиной до 2040 бит. По умолчанию в OpenSSL принимается длина ключа 128 бит, именно она рекомендована автором шифра. До сих пор неизвестны практически осуществимые атаки на RC5, но, как и RC2, RC5 поддерживает небезопасные короткие ключи, что является недостатком.

Другие шифры

Шифр **International Data Encryption Algorithm** (**IDEA**) был разработан в 1991 году как предполагаемая замена DES. Он был шифром по умолчанию в PGP версии 2, т. е. как раз в то время, когда PGP получила широкое распространение. При создании IDEA был защищен патентами, но срок действия последнего из них истек в 2012 году. Из-за патентной защиты невозможно было использовать IDEA в ядре GnuPG. По той же причине автор PGP решил сделать CAST5 шифром по умолчанию в PGP версии 3. В шифре используются 128-битовые ключи. Лучшая из известных атак снижает его стойкость до

126 бит. Шифр Blowfish был разработан, пожалуй, самым знаменитым в мире криптографом *Брюсом Шнайером* как свободный симметричный шифр – и это в 1993 году, когда большая часть других шифров была защищена патентами. Blowfish – шифр с переменной длиной ключа, поддерживающий ключи длиной от 32 до 448 бит. До сих пор неизвестно ни одной практической атаки против полного Blowfish с 16 раундами. Но, будучи шифром с переменной длиной ключа, Blowfish поддерживает небезопасные короткие ключи. Поэтому заранее нельзя сказать, будут ли стойкими данные, зашифрованные Blowfish. Программа расшифрования должна проверить длину ключа, принять решение об уровне стойкости зашифрованных данных и проинформировать пользователя, является ли процесс расшифрования интерактивным, но не все программы так делают. Это вторая слабость Blowfish, помимо малого размера блока (64 бит). Автор Blowfish разработал его преемника, шифр Twofish, в котором используется блок размера 128 бит и длины ключей 128, 192 или 256 бит. Он рекомендует пользователям Blowfish переходить на Twofish.

Режимы работы блочных шифров

У блочных шифров могут быть разные **режимы шифрования**, или **режимы работы**. Как мы уже знаем, блочный шифр зашифровывает открытые данные поблочно. Режим работы определяет, как блоки шифртекста зацепляются друг с другом. Мы дадим обзор самых популярных режимов работы.

Обзор режима простой замены

Простейшим режимом работы является **режим простой замены** (Electronic Code Book – **ECB**). В этом режиме каждый блок открытого текста преобразуется в блок шифртекста с применением только ключа шифрования, без использования **IV** или предыдущих блоков открытого или шифртекста. Таким образом, порожденные блоки шифртекста конкатенируются.

Режим ECB показан на следующем рисунке.

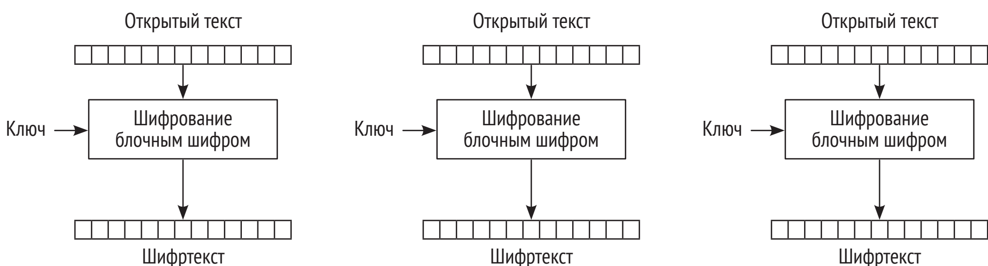


Рис. 2.3 ❖ Как работает режим ECB

В режиме ECB один и тот же открытый текст всегда порождает один и тот же шифртекст. С точки зрения безопасности, это проблема, потому что любые закономерности в открытом тексте сохраняются в шифртексте и видны наблюдателю.

Рассмотрим следующее изображение и его зашифрованное представление.

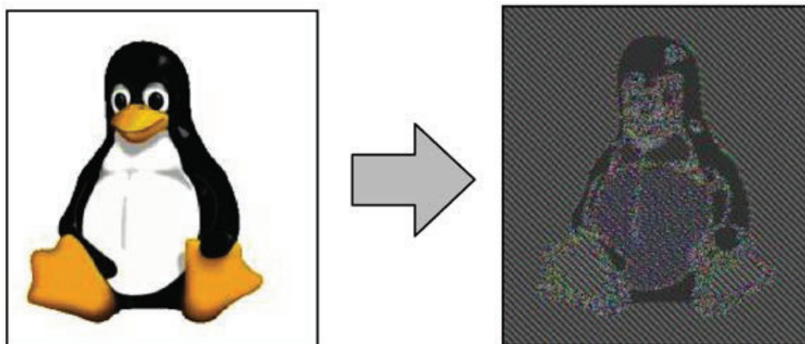


Рис. 2.4 ❖ Пример изображения и его зашифрованного представления

Источник обоих изображений пингвинов: https://isc.tamu.edu/~lewing/linux/, *автор: Ларри Эрвинг, лицензия: «Использовать и (или) модифицировать это изображение разрешается при условии ссылки на меня по адресу lewing@isc.tamu.edu и на GIMP, если кто-то спросит»*

Хотя второе изображение зашифровано, видно, что было изображено, поэтому режим ECB не дает достаточной защиты при шифровании. Следовательно, необходимо использовать лучше защищенный режим работы, например режим **простой замены с зацеплением** (Cipher Block Chaining – **CBC**).

Обзор режима простой замены с зацеплением

В режиме CBC шифртекст текущего блока зависит от шифртекста предыдущего блока. В отличие от режима ECB, где каждый блок открытого текста шифруется сам по себе, в режиме CBC к текущему блоку открытого текста и к предыдущему блоку шифртекста применяется операция XOR, а ее результат зашифровывается, порождая текущий блок шифртекста.

Что делать с самым первым блоком открытого текста, для которого нет предыдущего шифртекста? Он объединяется операцией XOR с вектором инициализации IV, т. е. с блоком данных, который используется для инициализации шифра. Во многих режимах, включая и CBC, IV обычно генерируется случайным образом и имеет такой же размер, как блок шифра. *Как правило*, IV не является секретом и хранится вместе с шифртекстом, потому что необходим для расшифрования. Но *иногда*, если это возможно и имеет смысл, например в протоколе TLS версии 1.1 и выше, IV передается в зашифрованном виде.

Режим работы CBC показан на следующем рисунке.

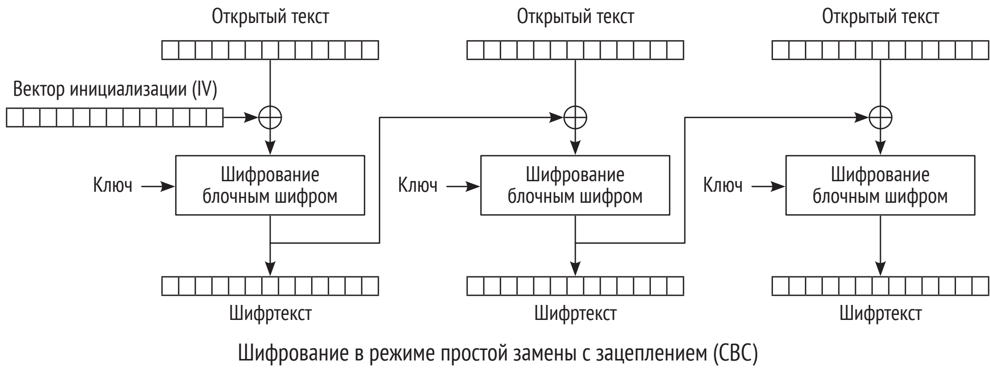


Рис. 2.5 ❖ Как работает режим CBC

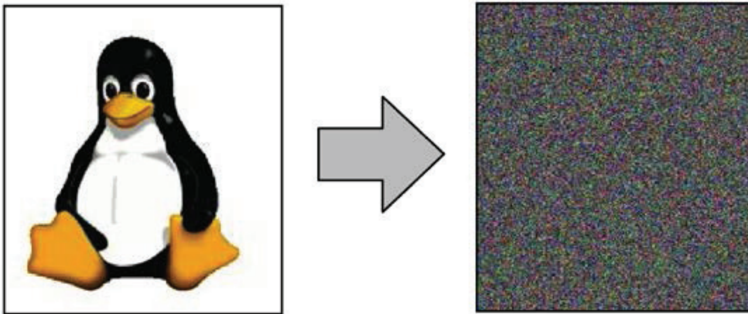


Рис. 2.6 ❖ Пример изображения и его представления, зашифрованного в режиме CBC

Видно, что характерные части исходного изображения не сохраняются при шифровании. Глядя на зашифрованное изображение, невозможно сказать, что было изображено первоначально. Таким образом, режим CBC обеспечивает хорошую защиту шифрования. Он скрывает от наблюдателя как сами исходные данные, так и связи между их частями. Другие режимы работы, отличные от ECB, также дают похожие случайные изображения.

Какой IV использовался в режиме CBC для данного сообщения, должно быть непредсказуемо, чтобы не пасть жертвой **атаки на основе подобранного открытого текста**. Проще всего удовлетворить этому требованию, генерируя IV случайным образом.

Почти во всех режимах работы, включая CBC, IV должен использоваться лишь для шифрования одного сообщения заданным ключом, повторное использование запрещено. Поэтому можно сказать, что IV обычно является **одноразовым значением**. Для некоторых режимов работы говорят, что IV формируется из одноразового значения.

Режим CBC был предложен в 1976 году и на протяжении многих лет был самым популярным режимом работы для шифрования файлов и сетевого трафика. Сегодня набирает популярность режим GCM (описан ниже), особенно в сетевых приложениях. Но CBC по-прежнему хорош для шифрования

файлов. GCM основан на режиме **гаммирования** (Counter – **CTR**), который рассматривается ниже.

Обзор режима CTR

В режиме CTR сам открытый текст не обрабатывается алгоритмом шифрования, как в случае AES. Вместо этого последовательность блоков счетчика шифруется блочным шифром. Например, каждый блок счетчика может состоять из результата конкатенации 64-битового случайного одноразового значения и фактического 64-битового целочисленного счетчика, который инкрементируется для каждого блока. Важно, что первый блок счетчика не используется повторно для шифрования другого открытого текста тем же ключом шифрования и что все блоки счетчиков различны.

Самый простой и популярный способ инкрементирования счетчика – прибавить к нему 1, но это необязательно. Вполне возможны и другие операции, например прибавление или вычитание числа, большего 1, или еще сложнее, например XOR и поразрядный сдвиг. Важно лишь, чтобы последовательность значений счетчика не начала повторяться.

Ну хорошо, в режиме CTR мы зашифровываем какую-то последовательность счетчиков, но чем это нам помогает? Как защитить открытый текст? Мы должны взять зашифрованные блоки *счетчиков* и применить к ним и к блокам *открытого текста* операцию XOR. Результатом и будет шифртекст.

Режим работы CTR изображен на следующем рисунке.

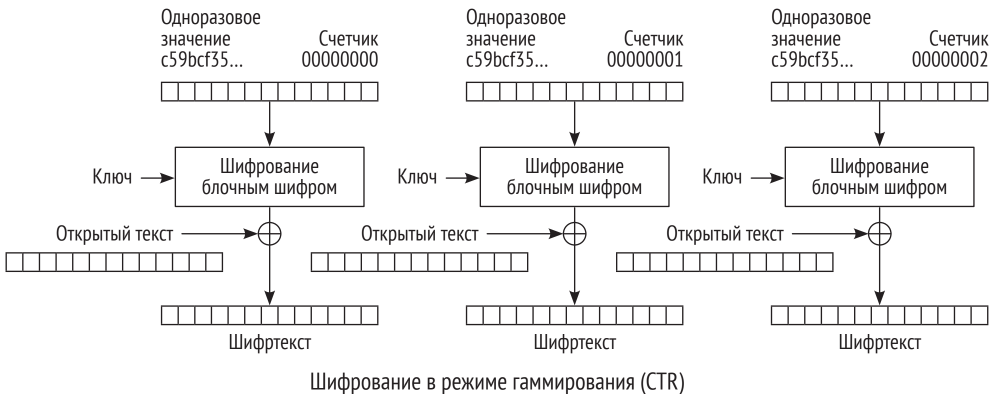


Рис. 2.7 ❖ Как работает режим CTR

Можно сказать, что режим CTR преобразует блочный шифр в потоковый. Действительно, последовательность *зашифрованных* блоков счетчика – это и есть гамма! Мы объединяем гамму с открытым текстом с помощью XOR – точно так же, как при использовании потокового шифра!

Первый блок счетчика можно интерпретировать по аналогии с IV. Как и IV в режиме CBC, первый блок счетчика (или, по крайней мере, его случайная часть) обычно сохраняется вместе с шифртекстом, потому что он нужен для

расшифрования. Метод инкрементирования блока счетчика обычно тоже не является секретом. В случае OpenSSL его даже можно найти в коде OpenSSL. Поэтому все блоки счетчика известны наблюдателю или противнику. Но *зашифрованные* блоки счетчика наблюдателю неизвестны, если только он не знает ключа шифрования. Если наблюдатель не может получить зашифрованные блоки счетчика (гамму), то не сможет и восстановить открытый текст, применив XOR к гамме и шифртексту.

IV, используемый в режиме CTR, никогда не следует применять повторно для шифрования другого сообщения тем же ключом. Если это сделать, то одна и та же гамма будет сгенерирована для шифрования нескольких сообщений. К двум шифртекстам, зашифрованным одной гаммой, можно применить XOR, и результат будет таким же, как если бы мы применили XOR к соответствующим открытым текстам. Для открытых текстов обычно характерна гораздо меньшая случайность, чем для гаммы, поэтому атака с целью восстановления открытого текста оказывается гораздо проще атаки на шифртекст. Кроме того, если возможна атака на основе подобранного открытого текста, то гамму можно восстановить, просто применив XOR к открытому и шифртексту, а затем использовать восстановленную гамму для расшифрования любого сообщения, зашифрованного той же гаммой или порожденного тем же ключом и IV. В то время как уникальность IV – очень важное требование в режиме CTR, его непредсказуемость в этом режиме не требуется, в отличие от режима CBC. Объясняется это тем, что устойчивость CTR к атаке на основе подобранного открытого текста базируется на непредсказуемости следующего блока гаммы, а не на непредсказуемости IV. Некоторые криптографы критикуют режим CTR на том основании, что на вход блочного шифра подаются несекретные данные для порождения гаммы. Тем не менее в настоящее время режим CTR широко распространен и признается такими криптографами, как *Нильс Фергюсон* и *Брюс Шнайер*, которые верят в его стойкость. Отметим также, что режим CTR был придуман еще двумя авторитетными криптографами, *Уитфилдом Диффи* и *Мартином Хеллманом*, прославившимися изобретением метода **обмена ключами Диффи–Хеллмана**, который используется в TLS.

Поскольку режим CTR преобразует блочный шифр в потоковый, он обладает всеми преимуществами потоковых шифров. В режиме CTR не требует дополнять открытый текст, а значит, невозможны атаки с оракулом дополнения. Гамму можно вычислять заранее, а не в момент шифрования. Наконец, операции шифрования и расшифрования сводятся к применению XOR к гамме и открытому тексту, а значит, разные блоки открытого текста можно зашифровывать и расшифровывать параллельно, пользуясь заранее вычисленной гаммой.

Режим CTR служит основой более развитого режима гаммирования с аутентификацией Галуа (Galois/Counter mode – **GCM**).

Обзор режима GCM

В режиме GCM шифртекст порождается так же, как в режиме CTR. Блоки счетчика шифруются блочным шифром, в результате чего порождается гамма.

Затем к гамме и открытому тексту применяется операция XOR, и получается шифртекст.

Отличие от режима CTR заключается в том, что GCM не только *шифрует открытый текст*, но также *аутентифицирует шифртекст*. Это означает, что знание ключа шифрования позволяет проверить **целостность** шифртекста, иными словами, обнаружить несанкционированные изменения шифртекста, если проверка не проходит. Такое обнаружение полезно, потому что в основе многих атак лежит изменение шифртекста: измененный шифртекст подается **оракулу**, и анализируется его ответ.

В криптографии **оракулом** называется программа, устройство или сетевой узел, который располагает ключом шифрования и может зашифровывать и расшифровывать данные. Оракул выдает ответ, иногда в форме (правильно или неправильно) зашифрованных или расшифрованных данных, иногда в форме сообщения об ошибке, а иногда какой-то другой информации, например времени, затраченного на расшифрование. Это время может различаться для успешного и безуспешного расшифрования. Ответ оракула позволяет противнику или криптоаналитику получить данные, которые могут помочь во вскрытии шифра.

Режимы шифрования с аутентификацией (authenticated encryption – **AE**), в частности GCM, дают возможность обнаружить несанкционированные изменения зашифрованных данных и отказаться от использования поврежденных данных.

Некоторые режимы шифрования с аутентификацией позволяют комбинировать зашифрованные и незашифрованные данные для аутентификации. Это необходимо для хранения или передачи данных, которые сами по себе не являются конфиденциальными, но связаны с зашифрованными конфиденциальными данными и нуждаются в защите от несанкционированного изменения. Такие режимы работы называются **шифрованием с аутентификацией и присоединенными данными** (authenticated encryption with associated data – **AEAD**). GCM является AEAD-режимом – дополнительные данные будут только аутентифицироваться, без шифрования.

Как GCM аутентифицирует шифртекст? Каждый блок шифртекста преобразуется в число, и к нему, и к начальному блоку, который зависит от ключа шифрования, применяются арифметические операции над конечным полем *Галуа*. Именно поэтому в названии режима фигурирует имя Галуа. Операция аутентификации порождает аутентификационный жетон, который расшифровывающая сторона может использовать для проверки целостности шифртекста.

Таким образом, выход шифрования в режиме GCM содержит IV, шифртекст и аутентификационный жетон (рис. 2.8).

Режим GCM не вполне обычный. Он работает только для 128-битовых блоков и требует 96-битового IV. Поэтому его невозможно использовать со старыми 64-битовыми блочными шифрами. В OpenSSL режим GCM можно использовать только с шифром AES.

GCM и другие режимы с аутентификацией аутентифицируют данные. Если расшифровывающая сторона хочет верифицировать целостность шифртекста, то она должна обработать весь шифртекст, чтобы проверить аутенти-

фикационный жетон. Обычно расшифрование и проверка производятся одновременно. Поэтому если проверка заканчивается неудачно, то расшифрованные данные следует отбросить.

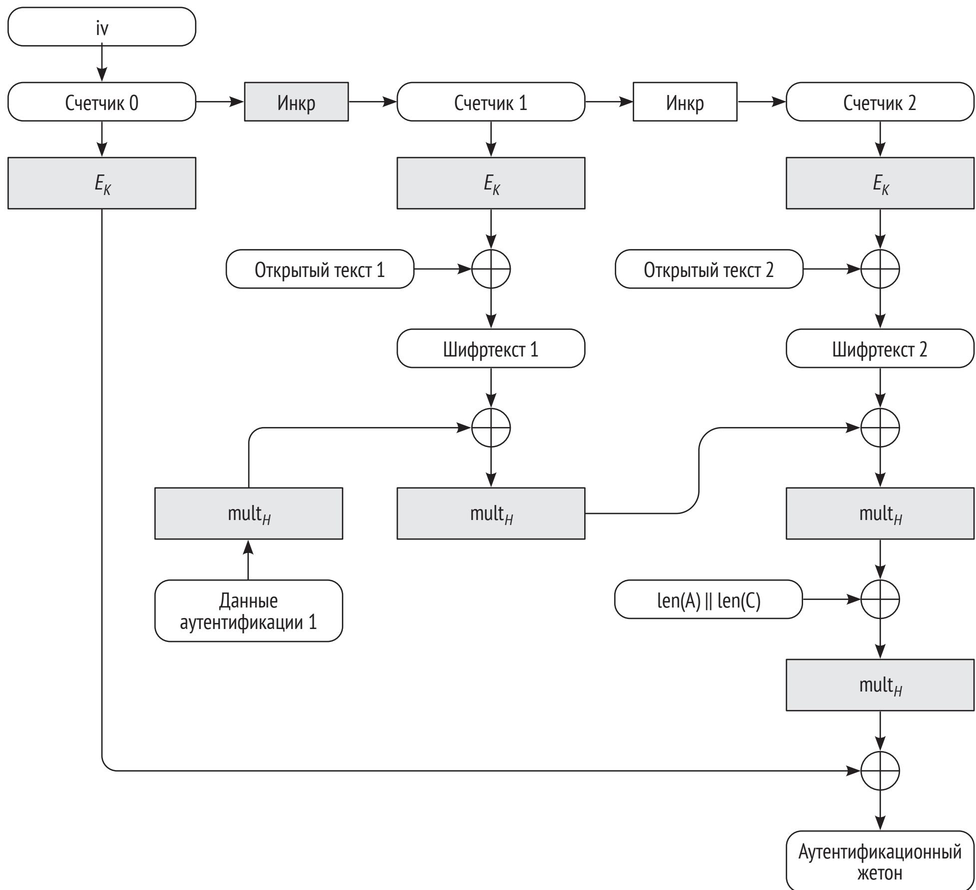


Рис. 2.8 ❖ Как работает GCM

Требование к IV в AES-GCM такое же, как в режиме CTR: IV должен быть уникален, т. е. один и тот же IV нельзя использовать повторно для шифрования другого сообщения тем же ключом. Если IV используется повторно, то против AES-GCM можно применить те же атаки, что против CTR. Кроме того, противник может восстановить ключ, использованный для аутентификации сообщений (не ключ шифрования). Зная ключ аутентификации, противник может сгенерировать поддельные аутентификационные жетоны и внедрить в поток шифртекста посторонние данные, кажущиеся подлинными.

В настоящее время GCM является самым популярным режимом блочного шифрования из используемых в протоколе TLS. GCM поддерживается начиная с версии TLS 1.2. В более старых версиях протокола TLS использовался режим CBC.

Протокол TLS всегда проверяет целостность зашифрованных данных, даже если используется режим шифрования без аутентификации. Для верификации применяются имитовставки. В интернете с трафиком может случиться много плохого. Кто-то может попытаться организовать атаку на данные **с человеком посередине** (Man in the Middle – **MITM**). Еще кто-то может попробовать использовать ваш сервер в качестве оракула. Поэтому в сети аутентификация данных важнее, чем в локальной файловой системе.

Мы уже говорили, что режим GCM уязвим для различных атак, если пара ключ–IV повторно используется для нескольких сообщений. Чтобы купировать эту слабость, был предложен **AES в режиме гаммирования с аутентификацией Галуа с синтетическим вектором инициализации** (Advanced Encryption Standard in Galois/Counter Mode with a Synthetic Initialization Vector – **AES-GCM-SIV**).

Обзор режима AES-GCM-SIV

AES-GCM-SIV – вариант шифра AES в режиме GCM. Его отличительной особенностью и преимуществом является то, что алгоритм не допускает неправильного использования IV или одноразового значения, т. е. гарантирует, что один и тот же IV никогда не будет использован повторно для разных сообщений.

AES-GCM-SIV не принимает IV в составе входных данных, зато принимает второй секретный ключ для аутентификации. Алгоритм генерирует аутентификационный жетон на основе открытого текста, дополнительных аутентифицированных данных и ключа аутентификации. Сгенерированный жетон используется также в роли IV для шифрования открытого текста. Этот сгенерированный IV называется **синтетическим вектором инициализации** (Synthetic Initialization Vector – **SIV**). Поскольку IV зависит от открытого текста, он будет разным для различных открытых текстов.

Недостаток AES-GCM-SIV – необходимость двух проходов по открытому тексту: первый – для генерирования аутентификационного жетона и IV, второй – для шифрования.

Другие режимы работы блочных шифров

Существует много других режимов работы блочных шифров, например **CBC с распространением** (propagating CBC – PCBC), **режим гаммирования с обратной связью по шифртексту** (Cipher Feedback – **CFB**), **режим гаммирования с обратной связью по выходу** (output feedback – OFB), **режим гаммирования с имитовставкой** (Counter with CBC-MAC – **CCM**), **режим Картера–Вегмана + CTR** (Carter-Wegman + CTR – CWC), **режим гаммирования с аутентификацией Софи Жермен** (Sophie Germain Counter Mode – SGCM) и т. д. Но они не очень популярны, поэтому здесь не рассматриваются.

Выбор режима работы блочного шифра

Считается, что лучшим режимом работы из поддерживаемых OpenSSL является GCM. GCM поддерживает не только шифрование, но и аутентификацию, не требует дополнения, а следовательно, неуязвим для атак с оракулом. GCM также позволяет предварительно вычислить гамму и распараллелить шифрование и расшифрование. При использовании GCM обязательно нужно задавать разные IV для всех открытых текстов, шифруемых одним и тем же ключом. Это очень важно – в противном случае ни о какой стойкости шифрования говорить не приходится.

Прежним стандартом де-факто для режима работы блочных шифров был CBC. В существующем ПО можно найти множество примеров применения CBC, а также немало данных, зашифрованных в этом режиме. В режиме CBC к открытому тексту необходимо применять дополнение. Если вам доведется разрабатывать или сопровождать программы, в которых используется режим CBC, то вы должны понимать, что такое дополнение.

Дополнение для блочных шифров

В режиме CBC открытый текст шифруется поблочно, но что делать с последним блоком, который обычно короче всех остальных? Его нельзя зашифровать просто так, потому что блочный шифр требует полного блока данных на входе. Поэтому последний блок дополняется до требуемого размера блока.

OpenSSL умеет производить дополнение автоматически на последнем шаге шифрования и удалять добавленные байты на последнем шаге расшифрования. Эту возможность можно запретить, и в таком случае разработчик должен добавлять и удалять дополняющие открытый текст байты самостоятельно.

Криптографы придумали различные типы дополнения. Для симметричного шифрования OpenSSL поддерживает только **дополнение по стандарту криптографии с открытым ключом номер 7** (Public Key Cryptography Standard number 7 – **PKCS #7**), которое также называют **дополнением PKCS7**, или просто **дополнением PKCS**, или **стандартным дополнением блока**. Дополнение PKCS #7 состоит из N байт, значение каждого из которых равно N. Например, если размер блока шифра равен 16 байт (128 бит), а последний блок открытого текста содержит только 10 байт данных, то длина дополнения будет равна 6 байт и каждый байт будет равен `0x06`. Все дополнение выглядит так: `0x06 0x06 0x06 0x06 0x06 0x06`.

Как видим, достоинство дополнения PKCS #7 в том, что кодируется длина дополнения, т. е. неявно кодируется длина полезных данных в последнем блоке открытого текста. Поэтому не нужно передавать вместе с данными длину всего открытого текста. Ну и заодно наблюдатель не видит длину последнего блока, потому что дополняющие байты шифруются вместе с последними полезными байтами данных.

Что будет, если длина открытого текста кратна размеру блока, так что последний блок целиком заполнен данными? В таком случае PKCS #7 все равно требует дополнять данные. Блок, содержащий только дополняющие байты, дописывается в конец открытого текста и подлежит шифрованию. В случае современных шифров со 128-битовыми блоками дописывается 16 дополняющих байт, каждый из которых имеет значение `0x10`. Если бы не это кажущееся лишним дополнение, то после расшифрования последнего блока невозможно было бы сказать, является ли последний байт дополнением или байтом данных.

Что означает *поддержка* дополнения PKCS #7 в OpenSSL? То, что OpenSSL автоматически добавляет дополняющие байты при шифровании, удаляет при расшифровании и проверяет, корректно ли применено дополнение. Дополнение некорректно, если расшифрованный открытый текст не заканчивается *N* байтами со значением *N*. В таком случае OpenSSL вернет ошибку на последнем шаге расшифрования.

Заметим, что дополнение PKCS #7 иногда называют дополнением PKCS #5. В стандарте дополнения PKCS #5 используется тот же принцип (*N* байт со значением *N*), но он определен только для блоков длиной не более 64 бит.

Недостаток дополнения PKCS #7 в том, что получающийся шифртекст уязвим для атак на дополнение с оракулом. Однако такие атаки возможны, только если противник способен получить много образцов шифрования подобранных им открытых текстов одним и тем же ключом, например отправляя с помощью JavaScript-программы данные по TLS-подключениям. Если мы контролируем весь зашифрованный открытый текст (например, только шифруем один файл), то невозможно или, по крайней мере, очень трудно провести такую атаку с оракулом.

Если вы контролируете метод шифрования и хотите избежать атак на дополнение с оракулом, то можете добавлять и удалять случайные дополняющие байты. В этом случае придется сохранять или отправлять вместе с шифртекстом длину открытого текста или хотя бы длину его последнего блока.

Как сгенерировать ключ симметричного шифрования

Сгенерировать ключ для симметричного шифрования на удивление просто. Нужно лишь запросить нужное число случайных байтов у **криптографически стойкого генератора случайных чисел**!

Какой генератор случайных чисел считается *криптографически стойким*? Тот, что генерирует байты, которые чрезвычайно трудно предсказать. Непредсказуемость достигается за счет использования энтропии, содержащейся в непредсказуемых событиях внешнего мира. Часто криптографически стойкие генераторы случайных чисел используют энтропию, содержащуюся в непредсказуемых входных данных от мыши или клавиатуры. Если клавиатура и мышь недоступны, например когда приложение выполняется в контейнере, то можно использовать какой-нибудь другой источник энтропии,

например микрофлуктуации скорости процессора. Еще один распространенный источник энтропии – кольцевой генератор, который может быть включен в процессор или еще какую-то микросхему.

Сколько должно быть случайных байтов? Их количество равно длине ключа шифрования. Например, если выбран шифр AES-256, то необходимо 256 случайных бит, или 32 случайных байта.

У некоторых шифров, например DES и IDEA, есть классы слабых ключей, при выборе которых шифр становится нестойким, но для всех популярных шифров вероятность случайно сгенерировать слабый ключ очень мала.

Где взять *криптографически стойкий* генератор случайных чисел? Зависит от **операционной системы** (**ОС**). Например, в Linux получить случайные байты можно от устройства `/dev/random`, выполнив следующую команду:

```
$ xxd -plain -len 32 -cols 32 /dev/random
```

Здесь и далее в примерах командных строк соблюдаются следующие соглашения:

- строки, начинающиеся знаком `$`, представляют команды, вводимые после приглашения;
- сам префикс `$` – приглашение в командной строке;
- все следующее за префиксом `$` – введенная команда;
- строки, не начинающиеся знаком `$`, представляют вывод команд.

Для генерирования криптографически стойких байтов в коде на C OpenSSL предлагает функцию `RAND_bytes()`, в которой используется зависящий от ОС **криптографически стойкий генератор псевдослучайных чисел** (**КСГПСЧ**) и которая возвращает затребованное количество стойких случайных байтов. Из командной строки к этой функции можно обратиться, воспользовавшись командой `openssl rand`:

```
$ openssl rand -hex 32
```

Полученный ключ можно использовать для симметричного шифрования. Его следует сохранить в безопасном месте или передать другой стороне по безопасным каналам, чтобы она могла расшифровать шифртекст.

Часто случайный ключ шифрования не нужен, достаточно ключа, сгенерированного детерминированно, например из пароля или парольной фразы. В таких случаях необходима **функция формирования ключа на основе пароля** (Password-Based Key Derivation Function – **PBKDF**). Дополнительную информацию о PBKDF см. в главе 5.

Скачивание и установка OpenSSL

Мы достаточно узнали о строительных блоках симметричного шифрования. Пришло время воспользоваться полученными знаниями и зашифровать какие-нибудь данные с помощью OpenSSL, но сначала нужно получить необходимые компоненты OpenSSL.

Комплект инструментов OpenSSL выложен на официальном сайте (https://www.openssl.org/) в виде исходного кода. Если вы разработчик ПО, то можете откомпилировать OpenSSL, следуя прилагаемой к исходному коду документации. Компиляция OpenSSL из исходного кода – предпочтительный способ, если необходима конкретная версия OpenSSL или нужно откомпилировать библиотеку с конкретными параметрами.

Можно также получить OpenSSL в откомпилированном виде. Почти все дистрибутивы Linux содержат OpenSSL в форме одного или нескольких установочных пакетов. Например, чтобы установить OpenSSL на Debian или Ubuntu Linux, достаточно выполнить всего одну команду:

```
$ sudo apt install openssl libssl3 libssl-dev libssl-doc
```

В другом дистрибутиве Linux команда может быть другой, но сама процедура установки во всех дистрибутивах одинакова. Пользователи FreeBSD, NetBSD и OpenBSD могут установить OpenSSL из перенесенных на BSD дистрибутивов. Пользователям macOS доступна также установка OpenSSL с помощью менеджера пакетов, например Homebrew или MacPorts.

В Windows для установки OpenSSL можно использовать один из менеджеров пакетов, например Scoop, Chocolatey или winget. Но в отличие от других операционных систем, пользователи Windows редко обращаются к менеджерам пакетов, а предпочитают автономные установочные пакеты. Такие пакеты для OpenSSL тоже изготавливаются сторонними разработчиками. Неполный их перечень можно найти по адресу https://wiki.openssl.org/index.php/Binaries.

Что окажется в системе после установки OpenSSL? Комплект OpenSSL состоит из следующих компонентов:

- динамические библиотеки;
- статические библиотеки;
- заголовочные файлы языка C;
- командная программа `openssl`;
- документация.

Динамические библиотеки (файлы с расширением `.so` или `.dll`) необходимы для динамической компоновки приложений с библиотекой OpenSSL и последующего выполнения этих приложений. В состав комплекта OpenSSL входят две библиотеки, `libcrypto` и `libssl`. `libcrypto` содержит реализации криптографических алгоритмов, а `libssl` – реализации протоколов SSL и TLS. `libssl` зависит от `libcrypto` и не может использоваться без нее. С другой стороны, `libcrypto` можно использовать без `libssl`, если требуется только криптография, а SSL/TLS не нужен. В системах с менеджерами пакетов, например Linux и варианты BSD, пакет, содержащий динамические библиотеки OpenSSL, обычно называется `libssl`, иногда с суффиксом версии, например `libssl3`. Пакет `libssl` необходимо установить, если какой-нибудь другой из установленных пакетов нуждается в библиотеке OpenSSL. Библиотека OpenSSL используется во многих приложениях, поэтому пакет `libssl` установлен по умолчанию в большинстве систем.

Статические библиотеки (файлы с расширением `.a` или `.lib`) необходимы для статической компоновки OpenSSL с приложениями. В этом случае ста-

тическая библиотека включается в исполняемый файл приложения и может отсутствовать на этапе выполнения, в отличие от динамических библиотек. Таким образом, статические библиотеки нужны только на этапе разработки. В системах с менеджерами пакетов пакет, содержащий статические библиотеки OpenSSL, обычно называется `libssl-dev`, `libssl-devel`, `openssl-devel` или еще как-то в этом роде. Обратите внимание на суффикс `dev` или `devel` – он напоминает, что пакет необходим для разработки.

Заголовочные файлы C необходимы для компиляции приложений, написанных на C/C++, которым требуется библиотека OpenSSL. Они нужны только на этапе разработки, поэтому в системах с менеджерами пакетов включаются в те же пакеты, что статические библиотеки – `libssl-dev`, `libssl-devel` или `openssl-devel`.

Команда `openssl` позволяет использовать функциональность OpenSSL из командной строки и не писать собственные приложения. Этот инструмент очень полезен для системных администраторов и инженеров DevOps. В системах с менеджерами пакетов команда `openssl` обычно включается в одноименный пакет `openssl`. Пакет `openssl` может содержать страницы **руководства** по команде `openssl`, хотя иногда они находятся в отдельном пакете с документацией.

Как и любая библиотека, OpenSSL нуждается в документации. Поскольку изначально эта библиотека писалась для Unix, документация по **интерфейсу прикладного программирования** (**API**) представлена в виде страниц руководства. В системах с пакетными менеджерами пакет, содержащий документацию, может называться `libssl-doc`, `openssl-man` или еще как-то в этом роде. Пакет может также включать примеры кода. Отметим, что на официальном сайте OpenSSL имеются и страницы руководства в формате HTML: https://www.openssl.org/docs/manpages.html.

Установив компоненты OpenSSL, мы можем приступить к шифрованию.

Шифрование и расшифрование с применением AES в командной строке

Мы собираемся зашифровать файл командой `openssl`.

Сначала сгенерируем файл:

```
$ seq 1000 >somefile.txt
```

Воспользовавшись полученными знаниями, выберем следующие параметры шифрования:

- шифр: AES-256;
- режим работы: CBC (надо бы выбрать GCM, но этот режим командной программой не поддерживается);
- тип дополнения: стандартное дополнение блока.

Как узнать из документации о том, какую команду вызывать для шифрования? Можно начать со страницы руководства по openssl:

```
$ man openssl
```

На этой странице описаны различные *подкоманды*, поддерживаемые программой openssl. Из страницы руководства мы узнаем, что интересующая нас подкоманда называется enc. Затем обратимся к странице руководства openssl-enc:

```
$ man openssl-enc
```

На странице openssl-enc можно узнать о параметрах этой подкоманды. Мы видим, что среди прочего ей нужен ключ шифрования и IV. Их можно сгенерировать случайным образом с помощью подкоманды rand. Документацию по ней можно прочитать обычным образом:

```
$ man openssl-rand
```

Итак, документацию мы прочитали – и можем сгенерировать 256-битовый ключ шифрования, пользуясь приобретенными знаниями:

```
$ openssl rand -hex 32
74c8a19fee9e0710683afe526462ce8a960ca1356e113bf3a08736a68a48eca0
```

Нам также понадобится 128-битовый IV той же длины, что размер блока:

```
$ openssl rand -hex 16
4007fbd36f08cb04869683b1f8a15c99
```

Наконец, зашифруем файл:

```
$ openssl enc \
  -aes-256-cbc \
  -K 74c8a19fee9e0710683afe526462ce8a960ca1356e113bf3a08736a68a48eca0 \
  -iv 4007fbd36f08cb04869683b1f8a15c99 \
  -e \
  -in somefile.txt \
  -out somefile.txt.encrypted
```

openssl ничего не выводит на терминал. По старой традиции Unix, это означает, что выполнение прошло успешно. В случае ошибки было бы напечатано сообщение.

Проверим, что зашифрованный файл создан:

```
$ ls -o somefile.txt*
-rw-rw-r-- 1 alexei 3893 Jul 27 20:17 somefile.txt
-rw-rw-r-- 1 alexei 3904 Jul 27 20:26 somefile.txt.encrypted
```

Обратите внимание, что зашифрованный файл на 11 байт длиннее (3904 вместо 3893 байт) и что число 3904 кратно 16. Вы, наверное, догадались, что лишние 11 байт – результат дополнения до 16-байтового блока.

Заглянув внутрь зашифрованного файла, вы увидите кажущиеся случайными двоичные данные, как и должно быть.

Попробуем расшифровать файл:

```
$ openssl enc \
  -aes-256-cbc \
  -K 74c8a19fee9e0710683afe526462ce8a960ca1356e113bf3a08736a68a48eca0 \
  -iv 4007fbd36f08cb04869683b1f8a15c99 \
  -d \
  -in somefile.txt.encrypted \
  -out somefile.txt.decrypted
```

Снова ничего не напечатано, а значит, все хорошо.

Проверим, что длина расшифрованного файла совпадает с длиной исходного. Это можно сделать, взглянув на содержимое файла в редакторе, но надежнее сравнить контрольные суммы:

```
$ cksum somefile.txt*
1830648734 3893 somefile.txt
1830648734 3893 somefile.txt.decrypted
4000774547 3904 somefile.txt.encrypted
```

Как видим, размеры (`3893`) и контрольные суммы (`1830648734`) файлов `somefile.txt` и `somefile.txt.decrypted` совпадают, и, значит, файлы идентичны. Размер и контрольная сумма файла `somefile.txt.encrypted` другие, как и следовало ожидать.

Если вы работаете в Windows, где нет команды `cksum`, то можете воспользоваться встроенной в OpenSSL возможностью вычислить криптографически стойкое **хеш-значение сообщения**:

```
$ openssl dgst somefile.txt*

SHA256(somefile.txt)= 67d4ff71d43921d5739f387da09746f405e425b-07d727e4c69d029461d1f051f

SHA256(somefile.txt.decrypted)= 67d4ff71d43921d5739f-387da09746f405e425b07d727e4c69d029461
d1f051f

SHA256(somefile.txt.encrypted)= 5a6a88e37131407c9dc9b8601b-3195c8a33c75a20421a1733233a963b
a42aef3
```

Хеш-значения сообщений гораздо надежнее контрольных сумм, вычисляемых командой `cksum`, но их вычисление занимает существенно больше времени. Мы не ожидаем, что файл `somefile.txt.decrypted` был специально обработан для получения конкретной контрольной суммы, поэтому и достоинства криптографически стойких контрольных сумм нам не нужны, так что применения `cksum` вполне достаточно. Кроме того, сравнивать напечатанные `openssl dgst` хеш-значения труднее, потому что они не выровнены. Хеш-значения сообщения подробно рассматриваются в главе 3.

Что будет, если попробовать расшифровать файл, задав неверный ключ шифрования или IV? Посмотрим. Обратите внимание, что первый байт ключа изменен с `0x74` на `0x00`:

```
$ openssl enc \
  -aes-256-cbc \
  -K 00c8a19fee9e0710683afe526462ce8a960ca1356e113bf3a08736a68a48eca0 \
  -iv 4007fbd36f08cb04869683b1f8a15c99 \
  -d \
  -in somefile.txt.encrypted \
  -out somefile.txt.decrypted
bad decrypt
140283139302720:error:06065064:digital envelope routines:EVP_
DecryptFinal_ex:bad decrypt:../crypto/evp/evp_enc.c:610:
```

Расшифрование завершилось ошибкой, но как openssl это обнаружила? Расшифрование – это просто последовательность арифметических и логических операций над 16-байтовым блоком. Мы не использовали шифрование с аутентификацией для проверки целостности данных. Мы не сообщили openssl никакую контрольную сумму для проверки результата расшифрования. Помогло стандартное дополнение! При расшифровании последнего блока получились мусорные данные, не содержащие в конце правильного дополнения, и openssl это заметила.

Посмотрим на контрольные суммы:

```
$ cksum somefile.txt*
1830648734 3893 somefile.txt
1498277506 3888 somefile.txt.decrypted
4000774547 3904 somefile.txt.encrypted
```

Обратите внимание, что расшифрованный файл на 16 байт короче зашифрованного (3888 и 3904 байт соответственно). Дело в том, что до последнего зашифрованного блока обнаружить ошибку было невозможно, и openssl записала в выходной файл все блоки, кроме последнего. Контрольные суммы файлов somefile.txt и somefile.txt.decrypted различны, как и должно быть. Содержимое somefile.txt.decrypted выглядит как набор случайных данных, поскольку мы пытались расшифровать файл неверным ключом.

Что, если бы мы применили режим работы, не требующий дополнения, например CTR? Попробуем расшифровать файл в режиме CTR:

```
$ openssl enc \
  -aes-256-ctr \
  -K 00c8a19fee9e0710683afe526462ce8a960ca1356e113bf3a08736a68a48eca0 \
  -iv 4007fbd36f08cb04869683b1f8a15c99 \
  -d \
  -in somefile.txt.encrypted \
  -out somefile.txt.decrypted
```

Ничего не печатается, т. е. openssl не обнаружила ошибок. И это притом, что мы задали неверный ключ и неверный режим! Результат расшифрования просто не может быть правильным. Сравним размеры и контрольные суммы файлов:

```
$ cksum somefile.txt*
1830648734 3893 somefile.txt
```

```
1916084428 3904 somefile.txt.decrypted
4000774547 3904 somefile.txt.encrypted
```

Контрольные суммы somefile.txt и somefile.txt.decrypted различны, как и должно быть. Содержимое somefile.txt.decrypted похоже на мусор – тоже правильно. Отметим, что размер файла somefile.txt.decrypted (3904) такой же, как размер somefile.txt.encrypted. Оно и понятно – ведь мы расшифровывали в режиме CTR, а значит, все расшифрованные данные были (неверно) интерпретированы как открытый текст без дополнения. Дополняющие байты не проверялись и не удалялись. Потому-то и размер расшифрованного файла совпадает с размером зашифрованного, и по той же причине openssl не смогла обнаружить ошибку при расшифровании.

Как видим, очень полезно иметь какую-то контрольную сумму или хеш-значение сообщения для проверки результата расшифрования.

Ну и достаточно экспериментов с командной строкой. Давайте теперь напишем свою первую программу, использующую OpenSSL как библиотеку.

Инициализация и очистка библиотеки OpenSSL

Начиная с версии 1.1.0 библиотеку OpenSSL не требуется явно инициализировать и очищать. Эти операции выполняются автоматически.

Однако до сих пор возможно инициализировать OpenSSL явно, если нужна какая-то нестандартная инициализация. Очистку тоже можно произвести явно, но это не рекомендуется, особенно в многопоточных программах или когда OpenSSL используется в одном процессе самой программой и еще какой-то библиотекой. Заметим, что после очистки заново инициализировать OpenSSL не получится.

Явную инициализацию выполняет функция OPENSSL_init_ssl(), а явную очистку – функция OPENSSL_cleanup().

В старых версиях OpenSSL, до 1.1.0, инициализация не производится автоматически. Если вам нужная старая версия, то делать это придется явно, вызывая ныне не рекомендуемые функции SSL_library_init() и EVP_cleanup().

Прочитать обо всех этих функциях можно на соответствующих страницах руководства:

```
$ man OPENSSL_init_ssl
$ man OPENSSL_cleanup
$ man SSL_library_init
$ man EVP_cleanup
```

Для большинства функций OpenSSL имеется страница руководства с таким же названием, как у самой функции. Это хороший источник информации. Как уже отмечалось, на сайте OpenSSL имеются также страницы руководства в формате HTML.

Напишем простую программу, которая только инициализирует и очищает OpenSSL, просто чтобы показать, как обращаться к библиотеке и как компилировать и компоновать программу, которая ее использует. Вот полный исходный код:

```
// Файл: init.c
#include <openssl/ssl.h>
#include <stdio.h>
int main() {
    printf("Инициализируется OpenSSL...\n");
    OPENSSL_init_ssl(0, NULL);
    printf("Очищается OpenSSL...\n");
    OPENSSL_cleanup();
    printf("Готово.\n");
    return 0;
}
```

Исходный код программы `init` доступен на сайте GitHub по адресу https://github.com/PacktPublishing/Demystifying-Cryptography-with-OpenSSL-3/blob/main/Chapter02/init.c.

Программа очень простая. Посмотрим, как ее откомпилировать и скомпоновать с OpenSSL.

Компиляция и компоновка с OpenSSL

Способ компиляции и компоновки программы с OpenSSL зависит от того, как пакет OpenSSL был установлен. В Unix вы могли установить заголовочные и библиотечные файлы OpenSSL в системные каталоги `include` и `lib`, например `/usr/include` и `/usr/lib/x86_64-linux-gnu`. В таком случае для компиляции и компоновки достаточно одной простой команды:

```
$ cc init.c -lssl -lcrypto
```

Заголовочные и библиотечные файлы ищутся в системных каталогах `include` и `lib`. Поэтому нет необходимости явно указывать, в каких каталогах производить поиск. Выходной исполняемый файл будет называться `a.out`.

Если вы ведете разработку в Unix, а показанная выше команда не работает, проверьте, установлен ли пакет OpenSSL для разработки, например `libssl-dev` или `openssl-devel`.

Если OpenSSL установлен не на системном уровне или вы не хотите использовать общесистемную установку, то нужно будет явно указать пути к заголовочным и библиотечным файлам OpenSSL:

```
cc \
  -I/path/to/openssl/c/headers \
  -L/path/to/openssl/library/files \
  -o init \
  init.c \
  -lssl -lcrypto
```

Обратите внимание на флаг `-o init`, который задает имя выходного исполняемого файла, поскольку `a.out` мало что говорит о назначении программы. Также отметим, что флаги `-lssl -lcrypto` находятся в командной строке после имени исходного файла. Это важно. Библиотеки должны указываться после файлов, в которых они используются. Если одна библиотека использует другую, то первая должна указываться раньше второй.

Путь может вести к динамическим или статическим библиотекам OpenSSL. Поддерживаются оба типа. Если имеются те и другие, то компоновщик выберет динамические.

Такой метод вызова компилятора с указанием немногих флагов годится только для небольших программ. Для более крупных лучше использовать какую-нибудь систему сборки, например GNU Make, BSD Make, CMake или интегрированную среду разработки (IDE), где можно задать необходимые пути и флаги и не повторять их в каждой командной строке.

При работе в Microsoft Windows вы, скорее всего, будете использовать IDE, например Microsoft Visual Studio. Понадобится задавать пути к заголовочным и библиотечным файлам OpenSSL C, потому что в Windows нет понятия системных путей к каталогам включаемых файлов и библиотек. Но не стоит недооценивать возможности командных инструментов сборки, потому что они весьма полезны для автоматизации сборки и **непрерывной интеграции**.

Ниже приведен пример файла `Makefile`, который можно использовать вместе с программой `make` для сборки нашей программы `init`:

```
# Файл: Makefile
CFLAGS += -I/usr/include
LDFLAGS += -L/usr/lib/x86_64-linux-gnu
LDLIBS += -lssl -lcrypto
init:
```

Здесь мы воспользовались неявными правилами компиляции, встроенными в `make`, которые определяют параметры компилятора в неявных переменных. Если заголовочные файлы и библиотеки находятся в каталогах, определяемых системными путями, то можно опустить пути в неявно подразумеваемых переменных `CFLAGS` и `LDFLAGS`. Программа `make` и компилятор не могут самостоятельно решить, какие библиотеки нужны для компоновки, поэтому опустить флаги `-lssl -lcrypto` в переменной `LDLIBS` нельзя.

Для сборки с применением `make` просто выполните команду

```
$ make
```

Это нужно сделать, находясь в том же каталоге, где хранятся файлы `Makefile` и `init.c`.

Убедимся, что откомпилированный исполняемый файл действительно скомпонован с библиотеками `libssl` и `libcrypto`:

```
$ ldd init
...
libssl.so.3 => /lib/x86_64-linux-gnu/libssl.so.3 (0x00007fd72096a000)
```

```
libcrypto.so.3 => /lib/x86_64-linux-gnu/libcrypto.so.3 (0x00007fd72068d000)
...
```

И наконец, выполним этот файл!

```
$ ./init
Инициализируется OpenSSL...
Очищается OpenSSL...
Готово.
```

Напечатано то, что мы и ожидали.

Итак, первый опыт оказался успешен – мы написали и собрали очень простое приложение, которое обращается к OpenSSL, но не делает ничего полезного. Однако мы узнали, как собирать программы, использующие OpenSSL. Теперь разовьем успех и напишем более полезную программу, которая будет шифровать файл, как это делает команда `openssl enc`.

Шифрование с применением AES из программы

Используя OpenSSL как библиотеку, мы не обязаны ограничиваться той функциональностью, которую предоставляет команда `openssl`. Конечно, `openssl` – отличный инструмент, но она не раскрывает всей функциональности OpenSSL. Например, `openssl enc` не поддерживает шифрование и расшифрование в режиме GCM, хотя OpenSSL как библиотека позволяет это сделать.

В этом разделе мы разработаем программу, которая умеет шифровать файл шифром AES-256 в режиме GCM. Назовем ее `encrypt`.

Чтобы не передавать слишком много параметров в командной строке, будем хранить IV и аутентификационный жетон в зашифрованном файле. В отличие от ключа шифрования, IV и жетон являются открытой информацией, которую необязательно держать в секрете. Зашифрованный файл будет иметь следующий формат:

Таблица 2.1. ***Формат зашифрованного файла***

Позиция, байты	Длина, байты	Содержимое
0	12	IV
12	?	Шифртекст
?	16	Аутентификационный жетон

Наша программа шифрования будет принимать три аргумента в командной строке.

1. Имя входного файла.
2. Имя выходного файла.
3. Ключ шифрования в шестнадцатеричном виде.

План реализации таков:

1. Сгенерировать случайный IV и записать его в выходной файл.
2. Инициализировать шифрование.
3. Шифровать порциями, читая порции открытого текста из входного файла и записывая результирующий шифртекст в выходной файл.
4. Завершить шифрование.
5. Получить аутентификационный жетон и записать его в выходной файл.

Посмотрим, как эти шаги выполняются.

Реализация программы шифрования

1. Сначала выделим память для входного и выходного буферов:

   ```
   const size_t BUF_SIZE = 64 * 1024;
   const size_t BLOCK_SIZE = 16;
   unsigned char* in_buf  = malloc(BUF_SIZE);
   unsigned char* out_buf = malloc(BUF_SIZE + BLOCK_SIZE);
   ```

   Эти буферы будут использоваться для чтения входных данных и хранения выходных перед записью. Отметим, что выходной буфер немного длиннее входного, потому что, согласно документации по OpenSSL, в некоторых случаях шифрование может порождать шифртекст, который длиннее открытого, но не более чем на `BLOCK_SIZE - 1` байт. В нашем случае такого быть не должно, но все равно послушаемся документации.
2. Сгенерировать IV просто – нужно лишь воспользоваться уже упоминавшейся выше функцией `RAND_bytes()`:

   ```
   RAND_bytes(iv, IV_LENGTH);
   ```

3. Далее нужно инициализировать шифрование. Для этого прежде всего создадим контекст шифра:

   ```
   EVP_CIPHER_CTX* ctx = EVP_CIPHER_CTX_new();
   ```

Скажем несколько слов о префиксе `EVP_` и суффиксе `_CTX`. При разработке программы с применением OpenSSL эти акронимы будут встречаться очень часто.

EVP означает **Envelope** и говорит об использовании **Envelope API**. Это высокоуровневый API, позволяющий выполнять высокоуровневые операции, например шифровать длинный открытый текст в требуемом режиме. Envelope API предоставляет разработчику набор структур данных и функций, в частности инициализацию шифрования, задание ключа и IV, добавление порций открытого текста, завершение шифрования и получение данных после шифрования, таких как аутентификационный жетон. Помимо Envelope API, OpenSSL включает также старый API шифрования и такие функции, как `AES_encrypt()`, `AES_cbc_encrypt()` и `AES_ofb128_encrypt()`. У старого API имеется ряд недостатков.

- Несогласованность имен. Например, в имени функции `AES_ofb128_encrypt()` указан размер блока (128), а в имени функции `AES_cbc_encrypt()` – нет.
- Высокоуровневые функции, например `AES_cbc_encrypt()`, и низкоуровневые функции, например `AES_encrypt()`, которая зашифровывает только один блок, находятся в одном и том же пространстве имен `AES_`. Это путает.
- Выбор шифра, например *AES*, *SEED* или *Camellia*, обозначается префиксом имени функции, т. е. `AES_`, `SEED_` или `Camellia_`. Режим работы шифра также сообщается в имени функции: такие подстроки, как `cbc` или `ofb128`. В результате образуется много похожих функций, по одной на каждую комбинацию шифра и режима работы. А надо бы передавать шифр и режим работы в виде параметров функции.
- Набор функций неполон. Например, функция `AES_ctr128_encrypt()` для шифрования в режиме CTR может присутствовать или не присутствовать в заголовочных файлах OpenSSL. Все зависит от версии OpenSSL. Функции для некоторых шифров, например ARIA и SM4, вообще отсутствуют в старом API.
- Авторы, вероятно, понимали, что множество похожих функций – это плохо, и не хотели заводить отдельную функцию расшифрования для каждой функции шифрования, поэтому решили передавать действие – шифрование или расшифрование – в параметре функции. В результате, чтобы выполнить расшифрование в режиме CBC, пользователь библиотеки должен вызвать функцию `AES_cbc_encrypt()`, но передать параметр `enc=0`, означающий расшифрование. Тогда как для расшифрования одного блока нужно вызывать функцию `AES_decrypt()`. Никакого единообразия!
- Невозможно задать дополнительные параметры шифрования или расшифрования при вызове высокоуровневых функций, например разрешить или запретить дополнение. В таком случае разработчик должен пользоваться функцией `AES_encrypt()` на уровне блока и программировать нужную ему операцию самостоятельно.
- Функции, поддерживающие режимы работы, могут принимать открытый или шифртекст только целиком. Невозможно подавать данные порциями, например если данные не помещаются в буфер или если выполняется потоковое шифрование. Это очень серьезный недостаток.

К счастью, все вышеупомянутые проблемы разрешены в EVP API. Старый API шифрования официально объявлен нерекомендуемым начиная с версии OpenSSL 3.0.

А теперь об аббревиатуре CTX. **CTX** означает **Context**. При работе с OpenSSL нам будет встречаться множество разных контекстов: для шифрования, для вычисления хеш-значений сообщений и имитовставок, для формирования ключей, для TLS-подключений, для TLS-сеансов и для настроек TLS. Что такое контекст? Это C-структура, в которой хранятся данные, относящиеся к текущей операции. Бывают высокоуровневые контексты, в которых хранятся

настройки, применимые к нескольким однородным операциям, например к нескольким TLS-подключениям.

В контексте EVP_CIPHER_CTX хранится тип шифра, ключ шифрования, IV, флаги и текущее внутреннее состояние операции шифрования. Есть возможность записать данные в контекст, например ключ и IV, и получить из него данные, например аутентификационный жетон. Объект EVP_CIPHER_CTX передается всем функциям, которые необходимо вызывать в процессе шифрования в выбранном режиме работы.

1. Итак, мы создали объект EVP_CIPHER_CTX. Следующая задача – инициализировать контекст и процесс шифрования, задав выбранные режим работы, ключ и IV:

   ```
   EVP_EncryptInit(ctx, EVP_aes_256_gcm(), key, iv);
   ```

2. С инициализацией покончено. Теперь можно шифровать порции, пользуясь функцией EVP_EncryptUpdate():

   ```
   while (!feof(in_file)) {
       size_t in_nbytes = fread(in_buf, 1, BUF_SIZE, in_file);
       int out_nbytes = 0;
       EVP_EncryptUpdate(
           ctx, out_buf, &out_nbytes, in_buf, in_nbytes);
       fwrite(out_buf, 1, out_nbytes, out_file);
   }
   ```

3. Шифрование почти закончено. Осталось только завершить его, вызвав функцию EVP_EncryptFinal():

   ```
   int out_nbytes = 0;
   EVP_EncryptFinal(ctx, out_buf, &out_nbytes);
   fwrite(out_buf, 1, out_nbytes, out_file);
   ```

   Завершение шифрования – операция, зависящая от используемого шифра и режима. Например, в режиме CBC она заключается в дополнении и шифровании последнего блока данных, в результате чего порождается последний блок шифртекста. Таким образом, операция завершения может порождать данные, поэтому принимает выходной буфер в качестве одного из параметров.
4. В GCM операция завершения вычисляет аутентификационный жетон, напоминая нам, что мы должны получить его из контекстного объекта и записать в выходной файл:

   ```
   EVP_CIPHER_CTX_ctrl(
       ctx, EVP_CTRL_GCM_GET_TAG, AUTH_TAG_LENGTH, auth_tag);
   fwrite(auth_tag, 1, AUTH_TAG_LENGTH, out_file);
   ```

5. И последняя операция – освобождение уже не нужного объекта EVP_CIPHER_CTX с целью избежать утечки памяти:

   ```
   EVP_CIPHER_CTX_free(ctx);
   ```

Это все операции OpenSSL, которые необходимы в этой программе!

Полный исходный код программы шифрования имеется на GitHub по адресу https://github.com/PacktPublishing/Demystifying-Cryptography-with-OpenSSL-3/blob/main/Chapter02/encrypt.c.

Там вы найдете полную реализацию, но почти без обработки ошибок, чтобы не загромождать код и сделать поток выполнения более наглядным.

А в файле https://github.com/PacktPublishing/Demystifying-Cryptography-with-OpenSSL-3/blob/main/Chapter02/encrypt-with-extended-error-checking.c приведена реализация с расширенной обработкой ошибок. Этот код не так понятен, как первая реализация, зато может быть полезен для отладки и исправления ошибок.

Небольшое замечание по поводу использования предложения `goto` в коде. Все мы знаем статью Дейкстры «О вреде оператора *goto*» и о том, что злоупотреблять им не стоит. Некоторые разработчики считают, что использовать `goto` не следует вообще никогда, но бывают случаи, когда `goto` полезен, например чтобы не дублировать код очистки, как в примерах выше. В C++ мы могли бы воспользоваться деструкторами для автоматической очистки, но в C такой возможности нет.

Выполнение программы encrypt

Попробуем выполнить нашу программу `encrypt`.

1. Сгенерируем тестовый файл для шифрования:

```
$ seq 20000 >somefile.txt
```

2. Зашифруем файл:

```
$ ./encrypt somefile.txt somefile.txt.encrypted \
74c8a19fee9e0710683afe526462ce8a960ca1356e113bf3a08736a68a48eca0
Encryption succeeded
```

3. Проверим размеры и контрольные суммы файлов:

```
$ cksum somefile.txt*
3231941463 108894 somefile.txt
3036900166 108922 somefile.txt.encrypted
```

Исходный и зашифрованный файлы различны, как и следовало ожидать. Зашифрованный файл на 28 байт длиннее, потому что содержит IV и аутентификационный жетон вдобавок к шифртексту.

Пока все хорошо, но как расшифровать то, что мы зашифровали?

Расшифрование с применением AES из программы

В этом разделе мы напишем программу `decrypt`, которая будет расшифровывать файл, зашифрованный программой `encrypt`.

Наша программа расшифрования будет похожа на программу шифрования и тоже будет принимать три аргумента в командной строке:

1) имя входного файла;
2) имя выходного файла;
3) ключ шифрования в шестнадцатеричном виде.

На этот раз на вход подается зашифрованный файл, созданный программой `encrypt`.

Составим общий план, как делали раньше.

1. Прочитать IV из входного файла.
2. Инициализировать расшифрование.
3. Расшифровывать порциями, читая порции шифртекста из входного файла и записывая результирующие порции открытого текста в выходной файл.
4. Прочитать аутентификационный жетон из входного файла и поместить его в контекст шифра.
5. Завершить расшифрование.

Как видим, план расшифрования очень похож на план шифрования: инициализировать, обработать и завершить. Посмотрим, как он воплощается в коде.

Реализация программы decrypt

1. Читаем IV:

   ```
   size_t in_nbytes = fread(iv, 1, IV_LENGTH, in_file);
   size_t current_pos = in_nbytes;
   ```

   Обратите внимание на переменную `current_pos`. Теперь мы не можем полагаться на чтение до конца файла и должны отслеживать текущую позицию в файле.
2. Инициализируем расшифрование. На сей раз используем для этой цели функцию `EVP_DecryptInit()`, а не `EVP_EncryptInit()`:

   ```
   ctx = EVP_CIPHER_CTX_new();
   EVP_DecryptInit(ctx, EVP_aes_256_gcm(), key, iv);
   ```

3. Основная часть программы – расшифрование порциями. Для этого используется функция `EVP_DecryptUpdate()` вместо `EVP_EncryptUpdate()`:

   ```
   while (current_pos < auth_tag_pos) {
       size_t in_nbytes_left = auth_tag_pos - current_pos;
   ```

```
        size_t in_nbytes_wanted =
            in_nbytes_left < BUF_SIZE ?
            in_nbytes_left : BUF_SIZE;
        in_nbytes = fread(in_buf, 1, in_nbytes_wanted, in_file);
        current_pos += in_nbytes;
        int out_nbytes = 0;
        EVP_DecryptUpdate(
            ctx, out_buf, &out_nbytes, in_buf, in_nbytes);
        fwrite(out_buf, 1, out_nbytes, out_file);
    }
```

Обратите внимание на следующую часть цикла расшифрования:

```
size_t in_nbytes_wanted =
    in_nbytes_left < BUF_SIZE ?
    in_nbytes_left : BUF_SIZE;
```

Смысл в том, чтобы не прочитать из входного файла слишком много, иначе аутентификационный жетон может попасть в буфер шифртекста.

4. После того как шифртекст прочитан и расшифрован, настало время прочитать аутентификационный жетон и поместить его в контекст шифра:

```
fread(auth_tag, 1, AUTH_TAG_LENGTH, in_file);
EVP_CIPHER_CTX_ctrl(
    ctx, EVP_CTRL_GCM_SET_TAG, AUTH_TAG_LENGTH, auth_tag);
```

Мы должны сделать это до завершения расшифрования, иначе операция завершения закончится неудачно.

5. Наконец, для завершения расшифрования следует использовать функцию `EVP_DecryptFinal()`:

```
int out_nbytes = 0;
EVP_DecryptFinal(ctx, out_buf, &out_nbytes);
fwrite(out_buf, 1, out_nbytes, out_file);
```

Аутентификационный жетон проверяется во время операции завершения. Таким образом, эта операция завершается неудачно в следующих случаях:

- аутентификационный жетон не был помещен в контекст шифра;
- шифртекст поврежден;
- при расшифровании был задан неверный ключ, IV или аутентификационный жетон.

Полный исходный код программы `decrypt` находится на GitHub по адресу https://github.com/PacktPublishing/Demystifying-Cryptography-with-OpenSSL-3/blob/main/Chapter02/decrypt.c.

Альтернативная реализация с расширенной обработкой, полезная для отладки, находится по адресу https://github.com/PacktPublishing/Demystifying-Cryptography-with-OpenSSL-3/blob/main/Chapter02/decrypt-with-extended-error-checking.c.

Итак, программа расшифрования написана. Давайте выполним ее.

Выполнение программы decrypt

Настало время выполнить нашу программу `decrypt` и проверить контрольные суммы файлов.

1. Расшифровать файл программой `decrypt`:

   ```
   $ ./decrypt somefile.txt.encrypted somefile.txt.decrypted \
   74c8a19fee9e0710683afe526462ce8a960ca1356e113bf3a08736a68a48eca0
   Decryption succeeded
   ```

2. Для начала неплохо! Проверим контрольные суммы файлов:

   ```
   $ cksum somefile.txt*
   3231941463 108894 somefile.txt
   3231941463 108894 somefile.txt.decrypted
   3036900166 108922 somefile.txt.encrypted
   ```

   У исходного и расшифрованного файлов одинаковые размер и контрольная сумма. Расшифрование и вправду сработало!
3. Что будет, если указать неверный ключ нашей программе расшифрования? Мы ожидаем получить сообщение об ошибке и мусор в неправильно расшифрованном файле. Воспользуемся версией программы с расширенной проверкой ошибкой:

   ```
   $ ./decrypt-with-extended-error-checking \ somefile.txt.encrypted
   somefile.txt.decrypted \
   74c8a19fee9e0710683afe526462ce8a960ca1356e113bf3a08736a68a48ec0a
   Could not finalize decryption
   Decryption failed
   ```

   Как и следовало ожидать, произошла ошибка на этапе завершения расшифрования. Проверка аутентификационного жетона оказалась неудачной из-за неверного ключа шифрования.
4. Взглянем на файлы:

   ```
   $ cksum somefile.txt*
   3231941463 108894 somefile.txt
   502982927 108894 somefile.txt.decrypted
   3036900166 108922 somefile.txt.encrypted
   ```

   У исходного и расшифрованного файлов размеры одинаковы, но контрольные суммы различны. Взглянув на содержимое расшифрованного файла, мы увидим мусор. Вот так влияет на расшифровку изменение всего лишь одного байта ключа.

Резюме

В этой главе мы узнали о симметричном шифровании – о том, что такое шифр вообще, что такое блочный и потоковый шифр, как измеряется стойкость шифра и какую стойкость можно считать достаточной. Затем мы узнали, какие шифры поддерживает OpenSSL, какой шифр в какой ситуации использовать, а каких шифров следует избегать. Мы также узнали, какие существуют режимы работы шифров, чем они отличаются и какие стоит использовать.

Мы обсудили дополнение – зачем оно нужно и в каких случаях обязательно. Мы поговорили об оракулах и о том, как скачать и установить OpenSSL. Мы рассмотрели, что включено в комплект инструментов OpenSSL, как инициализировать библиотеку OpenSSL и как компилировать и компоновать с ней свою программу. Наконец, узнали, как сгенерировать ключ шифрования и IV, как зашифровать и расшифровать файл с помощью командной утилиты OpenSSL и из программы.

Для проверки идентичности файлов мы пользовались контрольными суммами. Это очень удобно. Мы также упомянули хеш-значения сообщений, которые являются не чем иным, как криптографически стойкими контрольными суммами. В следующей главе продолжим разговор о хеш-значениях сообщений, а именно о тех алгоритмах их вычисления, которые поддерживаются OpenSSL.

Глава 3

Хеш-значения сообщений

В этой главе мы будем говорить о **хеш-значениях сообщений**, или **криптографических хешах**. У хешей сообщений, порождаемых криптографическими функциями хеширования, есть много применений, например проверка целостности данных, проверка паролей, цифровые подписи, безопасные сетевые протоколы типа TLS и даже блокчейны. Мы дадим обзор криптографических функций хеширования, поддерживаемых **OpenSSL**, и порекомендуем, какие функции стоит использовать, а от каких лучше держаться подальше. Затем в практической части главы мы научимся вычислять хеш сообщения с помощью командной программы и кода на C. Прочитав эту главу и проработав примеры, вы будете знать, зачем нужны хеш-значения сообщений и как их вычислять.

В этой главе рассмотрены следующие вопросы:

- что такое хеш-значения (хеши) сообщений и криптографические функции хеширования;
- зачем нужны хеши сообщений;
- оценка стойкости криптографических функций хеширования;
- обзор криптографических функций хеширования, поддерживаемых OpenSSL;
- вычисление хеша сообщения в командной строке;
- вычисление хеша сообщения из программы.

Технические требования

В этой главе встречаются команды, запускаемые из командной строки, и C-код, который можно откомпилировать и выполнить. Для запуска команд понадобятся программа `openssl` и соответствующие динамические библио-

теки OpenSSL. Для сборки кода на C будут нужны динамические или статические библиотеки OpenSSL, заголовки библиотек, компилятор C и компоновщик.

Мы реализуем демонстрационную программу, чтобы применить полученные знания на практике. Ее полный исходный код находится по адресу https://github.com/PacktPublishing/Demystifying-Cryptography-with-OpenSSL-3/tree/main/Chapter03.

Что такое хеш-значение сообщения и криптографическая функция хеширования?

Сообщением в криптографии называется любой фрагмент данных, большой или маленький, который обрабатывается **криптографическим алгоритмом**.

Криптографической функцией хеширования (хеш-функцией) называется алгоритм, который отображает сообщение произвольного размера в относительно короткий (скажем, 256 бит) массив битов фиксированного размера. Этот массив называется **хеш-значением сообщения**, или **криптографическим хешем**.

Иными словами, хеш-значение сообщения – это результат криптографической функции хеширования. Как отмечалось в предыдущей главе, можно сказать, что хеш сообщения – это криптографически стойкая контрольная сумма.

Хорошая криптографическая функция хеширования обладает следующими свойствами:

- она *детерминирована*, т. е. обработка одного и того сообщения всегда дает одно и то же хеш-значение;
- она *необратима*, т. е. невозможно или очень трудно восстановить исходное сообщение, зная его хеш. Единственным способом обратить хеш должен быть *полный перебор*, и его выполнение должно быть вычислительно неосуществимым;
- должно быть вычислительно неосуществимо найти два разных сообщения с одинаковым хеш-значением. Если найдено два или более сообщений с одинаковым хешем, то говорят о **коллизии хеш-функции**, или просто **коллизии**;
- любое изменение сообщения, большое или малое, должно приводить к значительному изменению хеш-значения – настолько значительному, что два хеш-значения невозможно связать друг с другом. Такая сильная реакция даже на малое изменение называется **лавинным эффектом**.

Нетрудно понять, что такое хеш-значение сообщения, но *зачем они нужны* и *почему так важны*?

Зачем нужны хеши сообщений?

У хешей сообщений много применений. Самое очевидное – **проверка целостности данных**.

Проверка целостности данных

При скачивании программ из интернета рядом со ссылкой на скачивание часто встречается хеш-значение дистрибутивного пакета. Вот, например, фрагмент страницы скачивания OpenSSL по адресу https://www.openssl.org/source/:

KBytes	Date	File
14627	2021-Sep-07 12:00:26	openssl-3.0.0.tar.gz (SHA256) (PGP sign) (SHA1)
9603	2021-Aug-24 13:46:31	openssl-1.1.1l.tar.gz (SHA256) (PGP sign) (SHA1)
1457	2017-May-24 18:01:01	openssl-fips-2.0.16.tar.gz (SHA256) (PGP sign) (SHA1)
1437	2017-May-24 18:01:01	openssl-fips-ecp-2.0.16.tar.gz (SHA256) (PGP sign) (SHA1)

Рис. 3.1 ❖ Фрагмент страницы скачивания OpenSSL

Рядом со ссылкой на скачивание мы видим ссылки `SHA256` и `SHA1`. Они содержат хеш-значения соответствующих `tar.gz`-файлов. После загрузки нужного `tar.gz`-файла можно вычислить его хеш-значение и сравнить с опубликованным. Если они совпадают, то файл не был изменен в процессе передачи и сохранения. Если файл был изменен, то из-за *лавинного эффекта* вычисленное хеш-значение будет сильно отличаться от ожидаемого. В нашем случае дистрибутивный пакет OpenSSL не особенно велик, официальный сайт OpenSSL является надежным источником, поэтому вероятность повреждения файла очень мала. Но представьте, что вы загружаете что-то по-настоящему большое, например установочный образ Linux, или получили дистрибутив из неофициального источника и хотите проверить, что он не подвергался манипуляциям. Тогда можно получить правильное хеш-значение на официальном сайте и сравнить.

Хеши сообщений можно применять также к хранимым данным. Если какие-то данные хранятся в течение длительного времени, то возможны повреждения из-за ошибок носителя или других неожиданных изменений. Верификация посредством сравнения хешей – надежный способ удостовериться в отсутствии повреждений.

Хеш-значения как основа HMAC

Хеш-значения сообщений лежат в основе **имитовставок** (или **кодов аутентичности сообщений**) **на основе функций хеширования** (Hash-based

Message Authentication Codes – **HMAC**). HMAC – это **имитовставка** (**MAC**), объединяющая секретный ключ и криптографическую хеш-функцию с целью аутентификации данных. HMAC будут подробно рассмотрены в главе 4.

Цифровые подписи

При цифровом подписании сообщения подписывается хеш-значение сообщения. Отметим, что цифровые подписи используются не только для документов, но также для подписания драйверов, приложений и даже криптографических объектов, таких как **сертификаты X.509**, используемые в протоколе TLS. Цифровые подписи будут рассмотрены подробнее в главе 7.

Сетевые протоколы

В безопасных сетевых протоколах, например TLS и SSH, HMAC и цифровые подписи передаваемых данных используются на разных этапах протокола. Выше уже упоминалось, что для HMAC и цифровых подписей необходимы криптографические функции хеширования, поэтому они необходимы и для безопасных сетевых протоколов.

Проверка пароля

В большинстве современных операционных систем и на веб-сайтах пароли не хранятся в открытом виде. Часто вместо пароля хранится его «подсоленный» хеш. Слово «подсоленный» означает, что перед хешированием к паролю добавлен модификатор – «соль», т. е. небольшой фрагмент случайных данных, длиной, скажем, 2 байта, который дописывается в начало или в конец пароля, из-за чего одинаковые пароли будут иметь разные хеши и восстановить пароль по известному хешу будет труднее. Соль сама по себе не является секретом и обычно хранится вместе с хешем. Во время проверки пароль конкатенируется с солью и результат хешируется. Проверка считается успешной, если вычисленное хеш-значение совпадает с хранимым.

Идентификатор содержимого

Поскольку одно из свойств хорошей криптографической функции хеширования – чрезвычайно малая вероятность коллизии, хеши сообщений можно использовать для идентификации объектов. Рассмотрим несколько примеров того, как криптографические хеши применяются для идентификации.

При верификации сертификата X.509 сертификат издателя ищут по имени и криптографическому хешу открытого ключа.

В современных системах **управления исходным кодом** (Source Code Management – **SCM**), например Git и Mercurial, криптографические хеши используются для однозначной идентификации хранимых объектов: файлов, фиксаций, ветвей и тегов. Такая схема идентификации заодно гарантирует целостность репозитория кода.

В некоторых одноранговых файлообменниках, например eDonkey, хеши сообщений используются для идентификации разделяемых файлов.

Блокчейн и криптовалюты

В **блокчейне** блоки связываются друг с другом с помощью криптографических хешей. Новые блоки ссылаются на старые по их хешам. Такая система гарантирует целостность цепочки и невозможность изменить старый блок, потому что при любом изменении блока изменится и его хеш. А это значит, что изменится идентификатор блока и он перестанет быть частью блокчейна.

Самое известное применение технологии блокчейна – криптовалюты. Но не единственное. В вышеупомянутых системах управления исходным кодом **Git** и **Mercurial** используется такой же вид ссылок между хранимыми объектами, так что они тоже основаны на технологии блокчейна. У блокчейна много и других применений: в торговле ценными бумагами, медицине и т. д. Блокчейн был бы невозможен без криптографических функций хеширования и хешей сообщений.

Доказательство выполнения работы

Система **доказательства выполнения работы** предлагает клиенту криптографическую задачу, с которой он может справиться, только выполнив определенный объем вычислений. Одна такая задача – **частичное обращение хеша**; ее смысл в том, что клиент должен найти фрагмент данных, для которого частичное хеш-значение совпадает с указанным в задаче. Если используется хорошая криптографическая функция хеширования, то клиент должен выполнить перебор, пусть и неполный. Найденные данные станут свидетельством того, что клиент выполнил работу.

Системы доказательства выполнения работы могут использоваться для майнинга криптовалют, обработки криптовалютных транзакций и предотвращения атак типа «**отказ в обслуживании**» или **спама**.

Объясним идею на примере спама. Допустим, что почтовый клиент подключается к почтовому серверу и хочет отправить сообщение на один или несколько адресов. Сервер выдает клиенту задачи – по одной для каждого целевого адреса. Если клиент хочет отправить сообщение только на один или на небольшое число адресов, то ему придется выполнить немного вычислений, что не станет для него большой проблемой. Но если спамер захочет разослать сообщения на миллион адресов, то объем вычислений окажется неподъемным.

Важное свойство системы доказательства выполнения работы заключается в том, что проверить факт выполнения работы гораздо проще, чем выполнить ее.

Почему в вышеупомянутой задаче нужно вычислять только частичное обращение хеша? Потому что для хорошей криптографической функции обратить хеш целиком слишком дорого, на решение этой задачи могут уйти миллиарды лет. А цель заключается в том, чтобы предотвратить злоупотребление сервисом, а не блокировать его вовсе.

Как видим, криптографические функции хеширования и хеши сообщений очень важны, и у них есть много полезных применений. Но польза от них имеется только тогда, когда они стойкие. Поговорим о популярных атаках на функции вычисления хеша сообщений, о том, как измеряется их стойкость, и о том, какая стойкость хеш-функции считается достаточной.

Оценка стойкости криптографических функций хеширования

Два самых важных типа атак на криптографические функции хеширования – коллизионные атаки и атаки нахождения прообраза. Идея коллизионной атаки – попытаться найти два сообщения с совпадающими хеш-значениями, а атаки нахождения прообраза – найти сообщение с заданным хеш-значением. Коллизионная атака проще атаки нахождения прообраза для той же функции хеширования, потому что поиск не ограничен одним конкретным хеш-значением.

Уровень стойкости криптографической хеш-функции – это вычислительная сложность коллизионной атаки. Уровень стойкости изменяется в битах, как и стойкость алгоритмов симметричного шифрования. Например, если для выполнения коллизионной атаки нужно вычислить 2^{128} хешей, то ее стойкость равна 128 бит.

Стойкость криптографической функции хеширования зависит от размера вычисляемого ей хеша сообщения. Если размер хеша равен n бит, то максимальная сложность атаки будет равна $2^{n/2}$ для коллизионной атаки и 2^n для атаки нахождения прообраза. Добиться, чтобы сложность коллизионной атаки была выше $2^{n/2}$, невозможно, потому что атака «дней рождения», основанная на парадоксе дней рождения, всегда сможет найти коллизии за время $2^{n/2}$.

Например, функция **SHA-256** порождает хеши размером 256 бит, поэтому сложность коллизионной атаки равна 2^{128}, сложность атаки нахождения прообраза равна 2^{256}, а уровень стойкости равен 128 бит. Говорят также, что стойкость к коллизиям равна 128 бит, а стойкость к нахождению прообраза равна 256 бит.

Сколько битов стойкости достаточно? Ответ такой же, как для симметричного шифрования:

- 112 бит должно быть достаточно до 2030 года;

- 128 бит должно быть достаточно до следующего революционного прорыва в технологиях или в математике.

Далее рассмотрим, какие криптографические функции хеширования поддерживаются OpenSSL и насколько они стойкие.

Обзор криптографических функций хеширования, поддерживаемых OpenSSL

В этом разделе мы рассмотрим криптографические функции хеширования, поддерживаемые OpenSSL, и их свойства, в частности стойкость, быстродействие и размер хеш-значения сообщения. Мы также немного поговорим об истории этих функций.

Семейство функций хеширования SHA-2

В семейство **SHA-2** входит наиболее популярная на сегодня криптографическая функция хеширования, SHA-256. Она порождает 256-битовое хеш-значение, а ее стойкость к коллизиям составляет 128 бит.

SHA-256 широко применяется. В настоящее время она используется по умолчанию в протоколе TLS, а также является функцией подписания по умолчанию для сертификатов X.509 и ключей SSH. В нескольких криптовалютах, включая биткойн, SHA-256 применяется для проверки транзакций и доказательства выполнения работы. Популярная система управления исходным кодом Git переходит на хеши SHA-256 в своей реализации блокчейна и для идентификации объектов. SHA-256 используется во многих протоколах и программах, связанных с безопасностью, в т. ч. SSH, IPsec, DNSSEC, PGP и др. SHA-256 также популярна как средство хеширования паролей в системах Unix.

Помимо SHA-256, в семейство SHA-2 входят следующие функции хеширования:

- SHA-224, модификация SHA-256 с *измененным вектором инициализации* и *усеченным выходом* (224 бита). Ее стойкость к коллизиям равна 112 бит;
- SHA-512 – алгоритм похож на SHA-256, но оперирует 64-битовыми, а не 32-битовыми словами. Порождает 512-битовое хеш-значение, стойкость к коллизиям равна 256 бит;
- SHA-384, SHA-512/256 и SHA-512/224 – модификации SHA-512 с измененными векторами инициализации и усеченным выходом. Функции порождают хеши длиной 384, 256 и 224 бита, а их стойкость к коллизиям равна 192, 128 и 112 бит соответственно.

Функции из семейства SHA-2 разработаны **Агентством национальной безопасности** (**АНБ**) США и впервые опубликованы **Национальным институтом стандартов и технологий** (**NIST**) в 2001 году в качестве федераль-

ного стандарта. Алгоритмы SHA-2 запатентованы, но лицензия свободна от отчислений.

Акроним **SHA** означает **Secure Hash Algorithm** (стойкий алгоритм хеширования). В настоящее время функции из семейства SHA-2 считаются стойкими, однако криптоаналитики медленно, но неуклонно развивают атаки на SHA-2. Но, несмотря на определенный прогресс, пока неизвестно об успешных атаках на полные версии SHA-2, есть лишь успехи на версии с уменьшенным числом раундов. Таким образом, на данный момент все функции SHA-2 имеют стойкость к коллизиям, равную половине числа битов в порождаемом хеш-значении, т. е. максимально возможную.

SHA-256 и ее модификации оперируют *32-битовыми словами*, а **SHA-512** со своими модификациями – *64-битовыми словами*. Поэтому вычисление SHA-256 в 2–4 раза быстрее, чем SHA-512, на 32-разрядных процессорах, тогда как на 64-разрядных SHA-512 работает примерно в 1,5 раза быстрее, чем SHA-256.

Быстродействие функций из семейства SHA-2 на современном оборудовании приемлемо, но все же ниже, чем у предшественницы – функции SHA-1. Например, SHA-256 в два раза медленнее, чем SHA-1.

Современные процессоры x86_64 производства Intel и AMD поддерживают **расширения Intel SHA**, ускоряющие процесс вычисления SHA-256 и SHA-1. Некоторые процессоры серии ARMv8, например Cortex A53, поддерживают **криптографические расширения ARMv8**, которые ускоряют процесс вычисления SHA-256 и SHA-1, а также SHA-224.

На момент написания книги криптографические функции из семейства SHA-2 наиболее популярны, но на смену им уже идет *семейство SHA-3*.

Семейство криптографических функций SHA-3

В отличие от разработки SHA-2 силами одного ведомства, алгоритмы для включения в **SHA-3** выбирались по итогам конкурса. Так же в свое время был выбран алгоритм **Advanced Encryption Standard** (**AES**). NIST организовал такой конкурс, потому что имелись успешные атаки на предшественников SHA-2 – алгоритмы **SHA-1**, **SHA-0** и **MD5**, построенные на тех же принципах, что SHA-2; возникла нужда в функции хеширования, основанной на иных принципах. Выбор SHA-3 затянулся надолго. NIST приступил к организации конкурса в 2006 году, а официально объявил о нем в 2007-м. Участники могли подавать предложения до 2009 года. Затем, в 2009 и в 2010 годах, состоялись два раунда конкурса. По их итогам в 2012 году был выбран победитель, в 2014 году опубликован стандарт, а в 2015-м окончательно утвержден.

Отметим, что алгоритмы из семейства SHA-2 не объявлены нерекомендуемыми в связи со стандартизацией SHA-3. Цель SHA-3 – повысить разнообразие хороших алгоритмов хеширования сообщений и запастить готовой заменой SHA-2 на случай, если возникнет такая необходимость.

Победителем конкурса SHA-3 стал алгоритм **Keccak**, созданный группой бельгийских криптографов. Одним из них был Йоан Даймен, соавтор алгоритма **Rijndael** (AES).

В состав семейства SHA-3 входят следующие функции:

- SHA3-224;
- SHA3-256;
- SHA3-384;
- SHA3-512;
- SHAKE128;
- SHAKE256.

Первые четыре – основные функции SHA-3, они похожи на соответствующие функции из семейства SHA-2 и порождают хеш-значения размером 224, 256, 384 и 512 бит соответственно. Как и в случае SHA-2, стойкость этих функций к коллизиям равна половине размера хеша, т. е. 112, 128, 192 и 256 бит соответственно, а стойкость к нахождению прообраза совпадает с размером хеша и равна 224, 256, 384 и 512 бит.

Как видим, *в настоящее время* эти четыре функции SHA-3 имеют такой же уровень стойкости, что и функции SHA-2. Но атаки на SHA-2 продолжаются, и в один прекрасный день это равенство может пасть их жертвой. На данный момент атаки на SHA-3 продвинулись не так далеко, как на SHA-2.

Алгоритм SHA-3 Keccak медленнее SHA-2, потому что содержит больше последовательных операций. Функция SHA3-256 примерно в 1,5 раза медленнее, чем SHA-256, а SHA3- 512 – в 2–4 раза медленнее SHA-512, в зависимости от процессора. Расширения системы команды, такие как **расширения Intel SHA** и **криптографические расширения ARMv8**, поддерживают только SHA-2, но не SHA-3. Однако для некоторых процессоров разработаны расширения, способные ускорить процедуру вычисления SHA-3, например **MMX**, **SSE**, **BMI2** и **AVX2/AVX-512/AVX-512VL** – для **x86/x86_64**, **SVE/SVE2** и **ARMv8.2-SHA** – для **ARM** и **Power ISA** – для процессоров **POWER8**. Отметим, что версия ARMv8.4-A для процессоров ARMv8 также поддерживает ускорение SHA-3.

В качестве компенсации замедления SHA-3 авторы разработали заметно более быстрые функции **SHAKE128** и **SHAKE256**. Строго говоря, это не функции хеширования, а **функции с удлиняемым результатом** (Extendable Output Function – **XOF**). Такие функции могут генерировать хеш-значения произвольной длины и служить основой для функций хеширования. Чтобы получить функцию хеширования из XOF, нужно выбрать конкретный размер хеша. Числа в именах SHAKE128 и SHAKE256 представляют максимальную стойкость функции к коллизиям. Отметим, что в силу парадокса дней рождения переменная длина хеша должна быть достаточной для достижения максимального уровня стойкости, т. е. не менее 256 бит для SHAKE128 и 512 бит для SHAKE256. Функция SHAKE128 работает приблизительно с такой же скоростью, как SHA-256, тогда как SHAKE256 в 1,5 раза медленнее SHA-512 на современных процессорах x86_64.

Впоследствии те же авторы представили функцию с удлиняемым результатом **KangarooTwelve** (**K12**), основанную на Keccak, но с уменьшенным числом раундов (12 вместо 24), которая к тому же может вычисляться параллельно на многоядерных процессорах. Авторы утверждают, что KangarooTwelve приблизительно в 13 раз быстрее SHA3-256 (0,51 цикла/байт вместо 6,4 цик-

ла/байт на процессоре Intel Skylake-X), но при этом сохраняет стойкость 128 бит. Однако независимые тесты производительности для практических реализаций показывают, что KangarooTwelve всего в 2–3 раза быстрее, чем SHA3-256. До сих пор утверждение о стойкости не опровергнуто, потому что самая известная атака на алгоритм SHA-3 применима только к версии Keccak с уменьшенным до 6 числом раундов. Однако у KangarooTwelve запас стойкости меньше, чем у полной SHA-3 с 24 раундами. Будучи более поздней разработкой, KangarooTwelve не считается частью семейства SHA-3 и не поддерживается OpenSSL.

Но хотя программные реализации SHA-3 медленнее SHA-2, некоторые аппаратные реализации работают быстрее, чем SHA-2 и даже SHA-1. А на современных процессорах быстродействие SHA-3 приемлемо.

В настоящее время семейство SHA-3 не так популярно, как SHA-2. Поддержка SHA-3 криптографическими библиотеками уже хорошая, но не так много приложений начали использовать хеш-функции из этого семейства. Алгоритмы SHA-3 еще не стандартизованы для использования в **TLS** или **X.509** и не применяются в таких популярных криптографических программах, как **OpenSSH** и **GnuPG**. Из заметных примеров использования SHA-3 отметим криптовалюту **Ethereum**, где она применяется для доказательства выполнения работы. Отметим также, что в настоящее время SHA-3 проходит процедуру стандартизации на предмет применения подписи и обмена ключами в постквантовой криптографии, определенную NIST. Переход с SHA-2 на SHA-3 происходит медленно, как, впрочем, было и с переходом на SHA-2 с предшествовавшей ей криптографической функции SHA-1.

Функции хеширования SHA-1 и SHA-0

SHA-1 – криптографическая функция хеширования, разработанная АНБ в 1990-х годах. Спецификация SHA-1 впервые была опубликована в 1993 году. Однако вскоре после того спецификация была отозвана, потому что в алгоритме обнаружились дефекты, влияющие на его стойкость. Пересмотренная спецификация SHA-1 была опубликована в 1995 году, а алгоритм 1993 года получил неофициальное название SHA-0.

SHA-1 порождает 160-битовое хеш-значение. Алгоритм работает быстро – примерно в два раза быстрее **SHA-256**. Он почти такой же быстрый, как предшествующий ему алгоритм **MD5**, а на современных процессорах даже быстрее благодаря аппаратному ускорению.

К сожалению, SHA-1 больше не считается стойким. Изначально было объявлено, что стойкость SHA-1 равна 80 бит, что по современным стандартам уже небезопасно. А лучшая из известных атак еще и снижает его стойкость до 61 бита. В 2017 году была опубликована первая найденная коллизия SHA-1. Голландский исследовательский центр **Centrum Wiskunde & Informatica** и Google опубликовали два PDF-документа с разным содержимым, но одинаковым хеш-значением SHA-1. Если бы вам удалось получить основанную на SHA-1 цифровую подпись одного документа, то ею можно было бы подписать и второй. Это лишь один пример того, почему коллизии криптогра-

фических функций хеширования опасны. В 2020 году стоимость нахождения коллизий SHA-1 уже была ниже 50 000 долларов, что вполне могут позволить себе крупные компании, разведывательные агентства и даже отдельные богатые люди.

SHA-1 был самым популярным криптографическим алгоритмом хеширования до того, как на первое место вышел SHA-256. Он использовался в протоколах SSL, TLS, SSH и IPsec, сертификатах X.509, реализациях почтовых систем **Pretty Good Privacy** (**PGP**) и **Secure Multipurpose Mail Extension** (**S/MIME**), алгоритме цифровой подписи **Digital Signature Algorithm** (**DSA**) и других областях. Такие системы управления исходным кодом, как Git и Mercurial, до сих пор используют SHA-1 для проверки целостности и идентификации объектов, хотя одновременно ведут работу по переходу на другие алгоритмы хеширования.

Переход с SHA-1 на SHA-256 занял у ИТ-индустрии много времени. Знаменитый криптограф *Брюс Шнайер* рекомендовал отказаться от SHA-1 еще в 2005 году. NIST официально объявил функцию SHA-1 нерекомендуемой в 2011 году, а основные браузеры перестали принимать основанные на SHA-1 сертификаты только в 2017 году. Microsoft прекратила поддержку основанных на SHA-1 подписей в обновлениях Windows в 2020 году. А многие не столь значимые приложения не завершили процесс перехода до сих пор.

Предшественницей SHA-1 была функция хеширования MD5 из семейства MD.

Семейство функций хеширования MD

Семейство MD включает четыре функции: **MD2**, **MD4**, **MD5** и **MD6**. Функций **MD1** и **MD3** не существует. Официальной информации о том, куда они делись, нет, но ходят слухи, что алгоритм MD1 был патентованным и широкой публике никогда не был доступен, а алгоритм MD3 был экспериментальным, и в конечном итоге автор от него отказался.

Акроним **MD** означает **Message Digest** (хеш-значение сообщения). Забавно, что **MD5** (равно как **SHA-1** и **SHA-2**) основан на **построении Меркла–Дамгора**, которое тоже можно сократить до MD.

MD-функции спроектировал знаменитый криптограф Рональд Ривест, тот самый, что изобрел симметричные шифры **RC – RC2**, **RC4** и **RC5**, рассмотренные в главе 2.

OpenSSL поддерживает хеш-функции MD2, MD4 и MD5. Все они порождают хеш-значение длиной 128 бит, а их начальная стойкость равна 64 бит. К сожалению, эти функции больше не считаются стойкими, и использовать их не рекомендуется. Лучшие известные атаки снижают стойкость MD2, MD4 и MD5 соответственно до 63, 1 (да, всего одного бита!) и 18 бит. Коллизионную атаку на MD4 или MD5 можно провести на типичном ПК за долю секунды.

MD5 – самая известная функция из семейства MD. MD5 предшествует SHA-1 и разделяет многие математические идеи с SHA-1 и SHA-2. MD5 была очень широко распространена, пока ей на смену не пришла SHA-1. Она исполь-

зовалась во всех популярных криптографических протоколах, стандартах и программах. Она также повсеместно применялась для **хеширования паролей** в Unix и вебе. Интересно, что в протоколах **SSL 3.0**, **TLS 1.0** и **TLS 1.1** использовалась комбинация **MD5** и **SHA-1**, потому что трудно найти блок данных, для которого имеет место коллизия **MD5** и **SHA-1** одновременно.

Функция **MD4** известна тем, что использовалась для хеширования паролей в Windows NT, 2000 и XP. По умолчанию применение MD4 для этой цели отключено начиная с Windows Vista, но ее все еще можно включить, даже в Windows 10.

Семейство функций хеширования BLAKE2

Семейство **BLAKE2** включает несколько функций хеширования, две из которых, **BLAKE2s** и **BLAKE2b**, поддержаны OpenSSL. Функция **BLAKE2s порождает** 256-битовое хеш-значение, а **BLAKE2b** – 512-битовое. Их стойкость к коллизиям до сих пор равна 128 и 256 бит соответственно. Запас стойкости у функций **BLAKE2** больше, чем у функций **SHA-2**, и аналогичен функциям **SHA-3**.

Функции **BLAKE2** известны своим быстродействием. **BLAKE2s** и **BLAKE2b** быстрее **MD5**, **SHA-1**, **SHA-2** и **SHA-3** на старых процессорах без аппаратного ускорения. На новых процессорах с аппаратным ускорением, например Intel Skylake или новее, **SHA-1** самая быстрая, но **BLAKE2s** и **BLAKE2b** все равно быстрее функций из семейства **SHA-2**, а именно **SHA-256** и **SHA-512**. **BLAKE2s** и **BLAKE2b** также заметно быстрее **SHA3-256** и **SHA3-512** и обладают похожим запасом стойкости.

Семейство криптографических функций BLAKE2 разработано международной группой криптографов в 2012 году. Оно основано на семействе функций BLAKE, которое было опубликовано в 2008 году, участвовало в конкурсе на звание SHA-3 и даже дошло до финальной стадии, но все же уступило алгоритму Keccak. Семейство BLAKE основано на потоковом шифре **ChaCha**, который рассматривался в предыдущей главе.

У семейства функций **BLAKE2** есть последователь: функция **BLAKE3**, обнародованная в 2020 году. Она поддерживает хеш-значения произвольного размера, большего или равного 256 битам. **BLAKE3** в несколько раз быстрее **BLAKE2** и допускает массовое распараллеливание, потому что основана на **дереве Меркла**. Что касается стойкости BLAKE3, то авторы утверждают, что ее стойкость к коллизионным атакам и атакам нахождения прообраза равна 128 бит. Но BLAKE3 пока не подвергалась такому скрупулезному криптоанализу, как другие функции хеширования, поэтому трудно сказать, насколько она действительно стойкая. BLAKE3 не включена в OpenSSL 3.0.

Хотя **BLAKE2** не стандартизована для использования в *TLS*, *SSH* или *PGP*, некоторые разработчики обратили внимание на эту хеш-функцию и решили включить ее в свои программы. Из популярных программ, в которых применяются алгоритмы BLAKE2, упомянем WhatsApp, 7-zip, WinRAR, Rsync, Chef, WireGuard криптовалюты Zcash и NANO.

Менее популярные функции хеширования, поддерживаемые OpenSSL

OpenSSL поддерживает и другие алгоритмы вычисления хеш-значений, не столь популярные, как рассмотренные выше. К их числу относятся национальные стандарты, принятые в отдельных странах.

Национальные криптографические функции хеширования

ГОСТ94[1] – российский стандарт криптографической функции хеширования, с которого гриф секретности был снят в 1994 году. Он основан на блочном шифре **ГОСТ89** и порождает 256-битовое хеш-значение. Изначально стойкость ГОСТ94 считалась равной 128 бит. Существует теоретическая атака, снижающая стойкость до 105 бит ценой практически невыполнимых требований к объему памяти. На смену ГОСТ94 пришла функция хеширования **Стрибог**, опубликованная в 2012 году и известная также под названием **ГОСТ2012**, или **ГОСТ12**[2]. И ГОСТ94, и Стрибог являются национальными стандартами Российской Федерации.

SM3 – китайская криптографическая функция хеширования, опубликованная в 2010 году. С точки зрения структуры и стойкости она похожа на SHA-256. Ее криптостойкость равна 128 бит. SM3 является национальным стандартом КНР с 2016 года.

Другие криптографические хеш-функции

Whirlpool – криптографическая функция, основанная на блочном шифре AES. Ее спроектировали Винсент Рэймен (один из соавторов AES) и Пауло Баррето. Функция генерирует 512-битовое хеш-значение, а ее криптостойкость равна 256 бит. Whirlpool – не быстрая функция хеширования; она работает приблизительно на 10 % медленнее SHA3-512. Whirlpool была опубликована в 2000 году, но с тех пор не снискала большой популярности. Из заметных применений Whirlpool упомянем программу шифрования диска **TrueCrypt** и пришедшую ей на смену **VeraCrypt**, которая поддерживает Whirlpool в качестве одного из алгоритмов хеширования.

RIPEMD-160 – функция хеширования, спроектированная группой бельгийских криптографов в 1996 году, наиболее популярная из **семейства RIPEMD**. Основана на оригинальной функции RIPEMD 1992 года, которая, в свою очередь, основана на MD4. RIPEMD-160 генерирует 160-битовое хеш-значение, ее стойкость равна 80 бит, чего, по современным стандартам, недостаточно. Поэтому использовать ее не рекомендуется.

MDC-2 – функция хеширования, в которой для порождения 128-битового хеш-значения используется симметричный блочный шифр с размером блока 64 бит. MDC-2 означает **Modification Detection Code 2** (код обнаружения модификаций 2). В OpenSSL реализация MDC-2 основана на шифре DES.

---

[1] Правильное название ГОСТ Р 34.11–94. – *Прим. перев.*

[2] Правильное название ГОСТ Р 34.11–2012. – *Прим. перев.*

К сожалению, стойкость 128-битовых хеш-значений не превышает 64 бит, а по нынешним временам этого недостаточно. Следовательно, MDC-2 не соответствует стандартам безопасности, и использовать ее не рекомендуется. Она считается устаревшей и по умолчанию не компилируется при сборке OpenSSL, хотя ее исходный код остается в библиотеке.

Какую криптографическую функцию хеширования выбрать?

Так какую же функцию вычисления хеш-значения сообщения стоит использовать в новом проекте? Если нет никаких специальных требований, то выбирайте **SHA3-256**. Она обеспечивает высокую стойкость, хорошо поддержана библиотеками, имеет приемлемую производительность и создана с заделом на будущее. Некоторые криптографы уже рекомендуют переходить с **SHA-2** на **SHA-3**. Ожидается, что функции семейства **SHA-3** получат улучшенную поддержку в будущих версиях стандартов и протоколов безопасности, таких как **TLS**, **SSH**, **PGP** и **X.509**, а также в популярных программах, где используется криптография, – браузерах, PGP и GnuPG, различных программах цифровой подписи и подсистемах хеширования паролей в операционных системах и сайтах. Также ожидается, что в новых процессорах и специализированных криптографических микросхемах алгоритмы **SHA-3** получат дополнительное ускорение.

Если совместимость и интероперабельность с имеющимся программным обеспечением нужна здесь и сейчас, то выбирайте функцию **SHA-256** из семейства **SHA-2**. Но будьте готовы перейти на **SHA-3**, если стойкость **SHA-2** окажется под угрозой.

Если требуются повышенная скорость и криптостойкость, а интероперабельность не важна, то выбирайте функцию **BLAKE2b** с 512-битовым хеш-значением. Она может похвастаться отличной стойкостью и заметно быстрее функций **BLAKE2s**, **SHA-2** и **SHA-3** на 64-битовых процессорах.

Но довольно теории! Займемся практическим вычислением хеш-значений сообщений.

Вычисление хеш-значения в командной строке

Вычислить хеш-значение сообщения в командной строке просто. Нужно лишь использовать подкоманду `openssl dgst`, документированную на странице руководства `openssl-dgst`:

```
$ man openssl-dgst
```

Сгенерируем тестовый файл для последующих вычислений:

```
$ seq 20000 >somefile.txt
```

Чтобы проверить, какие алгоритмы вычисления хеш-значения поддерживает ваша версия OpenSSL, воспользуйтесь флагом `-list`:

```
$ openssl dgst -list
Supported digests:
-blake2b512              -blake2s256              -md4
-md5                     -md5-sha1                -ripemd
-ripemd160               -rmd160                  -sha1
-sha224                  -sha256                  -sha3-224
-sha3-256                -sha3-384                -sha3-512
-sha384                  -sha512                  -sha512-224
-sha512-256              -shake128                -shake256
-sm3                     -ssl3-md5                -ssl3-sha1
-whirlpool
```

Вычислим хеш-значение SHA3-256:

```
$ openssl dgst -sha3-256 somefile.txt
SHA3-256(somefile.txt)= 658656e129914052546af527ba8cf573ab27f-
b47551a0682ffcf00eeaf56d32b
```

Как вы помните, вычисление хеш-значения – детерминированная операция. Сколько бы раз мы его ни вычисляли, будет получаться одно и то же.

Теперь посмотрим, как то же самое хеш-значение вычислить в программе на C.

Вычисление хеш-значения из программы

Мы напишем программу, которая будет вычислять хеш-значение файла с помощью функции SHA3-256. Наша программа будет принимать только один аргумент: имя входного файла.

Как и в предыдущей главе, составим план действий.

1. Сначала необходимо инициализировать контекст вычисления хеш-значения.
2. Затем нужно помещать в контексте данные порциями, читая их из входного файла.
3. Далее нужно завершить вычисление и получить хеш-значение.
4. Наконец, необходимо вывести вычисленное хеш-значение на *stdout*.

Как и при симметричном шифровании, OpenSSL включает новый **EVP API** и старый низкоуровневый, объявленный нерекомендуемым API вычисления хеш-значений. Имена функций из EVP API начинаются префиксом `EVP_`, а имена старого API – префиксами, описывающими функцию хеширования, например `MD5_` или `SHA256_`. Недостатки старого API мы обсуждали в предыдущей главе и повторяться не будем. А для вычисления хеш-значений воспользуемся EVP API. Но даже если бы мы захотели использовать низкоуровневый API, то все равно не могли бы этого сделать для функции SHA3-256, потому

что алгоритмы из семейства SHA-3 сравнительно новые, и старый API их не поддерживает.

Реализация программы digest

1. Сначала создадим и инициализируем контекст вычисления хеш-значений:

```
EVP_MD_CTX* ctx = EVP_MD_CTX_new();
EVP_DigestInit(ctx, EVP_sha3_256());
```

2. Далее будем помещать в контекст данные из входного файла:

```
while (!feof(in_file)) {
    size_t in_nbytes = fread(in_buf, 1, BUF_SIZE, in_file);
    EVP_DigestUpdate(ctx, in_buf, in_nbytes);
}
```

3. После того как все данные прочитаны и переданы в контекст, нужно завершить вычисление и получить хеш-значение в массив типа `unsigned char`:

```
const size_t DIGEST_LENGTH = 256 / 8;
unsigned char md[DIGEST_LENGTH];
EVP_DigestFinal(ctx, md, NULL);
```

4. После получения хеш-значения объект `EVP_MD_CTX` больше не нужен. Его необходимо освободить, чтобы не создавать утечек памяти:

```
EVP_MD_CTX_free(ctx);
```

5. Осталось только напечатать вычисленное хеш-значение. Выведем его в таком же формате, как подкоманда `openssl dgst`:

```
printf("SHA3-256(%s)= ", in_fname);
for (size_t i = 0; i < DIGEST_LENGTH; ++i) {
    printf("%02x", md[i]);
}
printf("\n");
```

По сравнению с симметричным шифрованием, программа вычисления хеш-значения принимает меньше параметров; ей не нужен ключ, вектор инициализации и тип дополнения. Требуются только тип функции хеширования и входные данные. К формату входных данных не предъявляется никаких требований, хешировать можно любой поток битов. Поэтому сделать ошибку при вычислении хеш-значения трудно. Если это все-таки произошло, то наиболее вероятные причины – ошибки ввода-вывода или инициализации, например функция хеширования не откомпилирована в вашей версии OpenSSL, из-за чего невозможно инициализировать контекст.

Полный исходный код программы находится по адресу https://github.com/PacktPublishing/Demystifying-Cryptography-with-OpenSSL-3/blob/main/Chapter03/digest.c.

Выполнение программы digest

Применим нашу программу вычисления хеш-значения к тому же файлу somefile.txt, что в предыдущем разделе:

```
$ ./digest somefile.txt
SHA3-256(somefile.txt)= 658656e129914052546af527ba8cf573ab27f-
b47551a0682ffcf00eeaf56d32b
```

Как видим, вычисленное хеш-значение совпадает с тем, что было вычислено подкомандой openssl dgst. Так что программа работает правильно.

Резюме

В этой главе мы узнали о криптографических функциях хеширования и о хеш-значениях сообщений. Мы также узнали о типичных атаках на функции хеширования и о том, как оценивается их криптостойкость. Затем мы дали обзор хеш-функций, поддерживаемых OpenSSL, и их стойкости. И завершили рассказ рекомендациями о том, какие хеш-функции в каких ситуациях применять.

В практической части главы мы видели, как вычислить хеш-значение с помощью подкоманды openssl dgst. И наконец, мы написали на C программу, которая пользуется библиотекой OpenSSL для вычисления хеш-значений. Сравнив ее результат с тем, что был вычислен подкомандой openssl dgst, мы убедились, что программа работает правильно.

В следующей главе мы поговорим о **кодах аутентичности сообщений**, или **имитовставках** (**MAC**) и **имитовставках на основе функций хеширования** (**HMAC**). HMAC основаны на криптографических функциях хеширования, именно поэтому и нужно было рассказать о функциях хеширования и хеш-значениях в этой главе.

Глава 4

MAC и HMAC

В этой главе мы будем говорить об **имитовставках**, или **кодах аутентичности сообщения** (message authentication code – **MAC**), которые иногда называют также **аутентификационными жетонами**. MAC'и используются в популярных безопасных сетевых протоколах, например *TLS*, *SSH* и *IPsec*, для гарантирования целостности и подлинности передаваемых данных. Применяются они и в патентованных протоколах, например в финансовых программах, для той же цели. Кроме того, имитовставки можно использовать для шифрования с аутентификацией без сети, как было показано в главе 2. Еще одно применение они находят как основа некоторых функций формирования ключа, например PBKDF2. Формирование ключа будет подробно рассмотрено в главе 5. Мы узнаем, как вычислять MAC в командной строке и в программе на C.

В этой главе рассмотрены следующие вопросы:

- что такое имитовставка;
- стойкость функции вычисления MAC;
- HMAC – MAC на основе функции хеширования;
- MAC, шифрование и принцип криптографической обреченности;
- вычисление HMAC в командной строке;
- вычисление HMAC из программы.

Технические требования

В этой главе встречаются команды, запускаемые из командной строки, и C-код, который можно откомпилировать и выполнить. Для запуска команд понадобятся программа `openssl` и соответствующие динамические библиотеки OpenSSL. Для сборки кода на C будут нужны динамические или статические библиотеки OpenSSL, заголовки библиотек, компилятор C и компоновщик.

Мы реализуем демонстрационную программу, чтобы применить полученные знания на практике. Ее полный исходный код находится по адресу https://github.com/PacktPublishing/Demystifying-Cryptography-with-OpenSSL-3/tree/main/Chapter04.

Что такое имитовставка?

В двух словах имитовставка (MAC) – это короткий массив битов длиной, скажем, 256, который **аутентифицирует** сообщение. Это означает, что получатель может удостовериться, что сообщение поступило от декларируемого отправителя и не было изменено в процессе передачи. Чтобы сгенерировать MAC, отправителю нужны сообщение и **секретный ключ**. Для проверки MAC получателю нужны сообщение и тот же самый ключ. Имитовставка порождается функцией **вычисления MAC**.

Разница между имитовставкой и хеш-значением сообщения заключается в том, что хеш-значение не защищено от подделки, тогда как MAC обеспечивает такую защиту. Если сообщение и его хеш-значение передаются по незащищенной сети, то противник может изменить сообщение и пересчитать его хеш-значение, так чтобы они соответствовали друг другу. С другой стороны, если вместе с сообщением передается его имитовставка, то противник не сможет пересчитать MAC измененного сообщения, т. к. не располагает секретным ключом. Поэтому общепризнано, что хеш-значения гарантируют только **целостность** сообщения, а имитовставки – **целостность** и **аутентичность**.

MAC также отличается от **цифровой подписи**. В цифровых подписях используется асимметричная криптография, т. е. подписывающая и проверяющая стороны пользуются разными ключами из одной и той же пары ключей. Создать подпись может только подписывающая сторона, потому что только она обладает **закрытым ключом**. Таким образом, цифровая подпись обеспечивает **неотрицаемость**, т. е. подписывающая сторона не сможет впоследствии отрицать, что располагала информацией, которую подписала. При использовании MAC отправитель и получатель используют один и тот же секретный ключ. Поэтому обе стороны могут сгенерировать MAC для любой информации, которой обладают, и третьей стороне трудно определить, кто именно это сделал. Поэтому MAC, в отличие от цифровой подписи, не обеспечивает неотрицаемости. Дополнительные сведения о цифровых подписях см. в главе 7.

Поскольку функции вычисления MAC отличны и от функций хеширования, и от цифровых подписей, к их криптостойкости предъявляются другие требования.

Стойкость функции вычисления MAC

Функция вычисления MAC считается стойкой, если она неуязвима для атак с подлогом, перечисленных ниже в порядке убывания трудности. Во всех описанных атаках предполагается, что противник не знает секретного ключа.

- **Универсальная атака с подлогом** успешна, если противник может создать корректную MAC для любого сообщения.
- Цель **избирательной атаки с подлогом** – создать корректную MAC для конкретного сообщения. Сообщение выбирается перед атакой,

и обычно имитация его подлинности имеет некоторую ценность для противника.

- **Экзистенциальная атака с подлогом** считается успешной, если удалось найти пару, состоящую из любого сообщения и соответствующей ему MAC.
- В **экзистенциальной атаке с подлогом на основе подобранного сообщения** предполагается, что существует оракул, который может сгенерировать MAC для любого сообщения, выбранного противником. Противник может подавать сообщения оракулу, получать имитовставки и анализировать поведение оракула и исполняемой им функции вычисления MAC. Атака считается успешной, если противник может найти какую-нибудь пару, состоящую из сообщения, которое не подавалось оракулу, и соответствующей ему имитовставки.

Как всегда, атака на MAC может считаться успешной, только если ее выполнение не требует практически недоступных вычислительных ресурсов.

Стойкость функции вычисления MAC измеряется в тех же единицах, что стойкость любых других криптографических функций, – в битах. Важно отметить, что обычно функция вычисления MAC пользуется какой-то другой функцией, имеющей собственные свойства безопасности. Стойкость функции вычисления MAC зависит от стойкости этой базовой функции и секретного ключа.

Существуют разные типы функций вычисления MAC, в сетевых протоколах чаще всего используется HMAC.

HMAC – MAC на основе функции хеширования

Имитовставка на основе функции хеширования (Hash-based Message Authentication Code – **HMAC**) – это имитовставка, которая генерируется **функцией вычисления HMAC**.

В функции вычисления HMAC используются криптографическая функция хеширования и секретный ключ. Функция не особенно сложная и доступна пониманию людей, не занимающихся криптографией профессионально. Определяется она следующим образом:

```
HMAC(K, message) = H(K’ XOR opad || H(K’ XOR ipad || message))
```

где

- `message`: сообщение, подлежащее аутентификации;
- `H`: функция хеширования, например SHA3-256;
- `K`: секретный ключ;
- `K’`: ключ длины, равной размеру блока, сформированный на основе `K` и зависящий от длины внутреннего блока функции хеширования, `B`;
- `ipad`: внутреннее дополнение, состоящее из байта `0x36`, повторенного `B` раз;

- opad: внешнее дополнение, состоящее из байта 0x5C, повторенного B раз;
- ||: символ конкатенации.

K' формируется из K следующим образом:

- если длина K меньше или равна B, то K' совпадает с K, дополненным байтами 0x00 до длины B;
- если длина K больше B, то K' совпадает с H(K), дополненным байтами 0x00 до длины B.

Отметим, что размер внутреннего блока функции хеширования, B, не совпадает с длиной генерируемого хеша. Как правило, длина хеша меньше B. Например, длина хеша функции SHA-256 равна 256 бит, тогда как размер ее внутреннего блока равен 1088 бит.

Отметим еще, что имя функции вычисления HMAC, пользующейся некоторой функцией хеширования, часто образуется из префикса *HMAC-* и имени этой функции. Например, функция вычисления HMAC, основанная на функции SHA-256, называется *HMAC-SHA-256*.

Функция вычисления HMAC порождает значение HMAC той же длины, что длина хеша, порождаемого базовой функцией хеширования. Так, функция HMAC-SHA-256 порождает 256-битовую HMAC.

У любопытного читателя может возникнуть вопрос: зачем усложнять функцию вычисления HMAC, включая вложенное хеширование и дополнение? Почему бы просто не предложить такую реализацию HMAC:

```
H(K || message)
```

Краткий ответ – все эти вычисления нужны, чтобы противостоять различным атакам на HMAC и прежде всего **атаке с удлинением сообщения**. Цель этой атаки – породить правильную MAC, дописав контролируемое противником сообщение в конец исходного, не зная ни исходного сообщения, ни секретного ключа. Многие криптографические функции хеширования, в т. ч. SHA-256, уязвимы перед атакой удлинением сообщения. Более современные семейства хеш-функций, в т. ч. SHA3, не боятся этой атаки, так что их можно использовать в сочетании с более простыми функциями вычисления MAC. Например, для функций из семейства SHA-3 существует собственная функция вычисления MAC: **KECCAK Message Authentication Code** (**KMAC**).

Что до криптостойкости, то для нескомпрометированной функции вычисления HMAC она вычисляется по формуле

```
SecurityLevel = min(K_bits, L)
```

где

- K_bits: стойкость секретного ключа K в битах;
- L: длина хеша, вырабатываемого базовой функцией хеширования, в битах.

Например, если используется 256-битовый секретный ключ, то функция HMAC-SHA-256 достигает предела своей стойкости: 256 бит.

Так какова же криптостойкость секретного ключа? По сути дела, это его энтропия. Если ключ генерируется криптостойким генератором случайных

чисел, то его стойкость равна длине. Это означает, что 256-битовый ключ имеет стойкость 256 бит. Если есть подозрение, что у используемого генератора случайных чисел степень случайности недостаточна, например на каждые 2 сгенерированных бита приходится только 1 бит энтропии, то для получения нужной стойкости (256 бит) имеет смысл генерировать ключ в два раза большей длины (512 бит). Но что, если ключ сформирован на основе низкоэнтропийного источника данных, например пароля? Увы, ключ не может иметь большую энтропию, чем источник, и стойкость всей функции вычисления HMAC с таким ключом уменьшается до уровня энтропии, содержащейся в пароле. Дополнительные сведения о формировании ключей из паролей приведены в главе 5.

Функция вычисления HMAC может принимать ключ любой длины. Однако использовать ключи длины, меньшей `L`, с учетом того, что они получены от криптостойкого генератора случайных чисел, настоятельно не рекомендуется, потому что они уменьшают стойкость функции вычисления HMAC. Ключи длины, большей `L`, использовать можно, но увеличение длины ключа не сильно повышает стойкость функции вычисления HMAC, поскольку длина ее результата – все равно `L`. При применении HMAC в протоколе TLS OpenSSL пользуется секретными ключами длины `L`.

Угадать секретный ключ функции вычисления HMAC должно быть столь же невозможно на практике, как угадать ключ симметричного шифра. Это значит, что ключ должен содержать достаточно энтропии. Один из способов добиться этого – использовать криптостойкий генератор псевдослучайных чисел.

Как уже отмечалось, стойкость HMAC зависит от стойкости базовой функции хеширования. Однако функция вычисления HMAC часто бывает значительно более стойкой, чем базовая хеш-функция, если последняя скомпрометирована. Например, хотя функция *MD5* серьезно скомпрометирована и на сегодня ее стойкость составляет 18 бит, считается, что стойкость функции *HMAC-MD5* равна 97 бит. Но хотя функция вычисления HMAC и является более стойкой, чем ее хеш-функция, разумно будет избегать ненужных рисков и использовать для вычисления HMAC функцию, основанную на заведомо стойкой функции хеширования, например HMAC-SHA-256.

Практическая ценность HMAC заключается, в частности, в том, что этот тип MAC используется в протоколах TLS, SSH и IPsec. В следующем разделе мы объясним, как MAC сочетается с шифрованием и применяется в безопасных сетевых протоколах.

МАС, шифрование и принцип криптографической обреченности

Для комбинирования MAC с шифрованием используется одна из следующих схем:

- **шифрование, затем MAC** (Encrypt-then-MAC – **EtM**): открытый текст сначала шифруется, после чего для *шифртекста* вычисляется MAC и отправляется вместе с шифртекстом;
- **шифрование и MAC** (Encrypt-and-MAC – **E&M**): открытый текст шифруется, но MAC вычисляется для *открытого текста*, а не для шифртекста. Затем шифртекст и MAC отправляются вместе;
- **MAC, затем шифрование** (MAC-then-Encrypt – **MtE**): MAC вычисляется для *открытого текста*. Затем шифруются конкатенация открытого текста и MAC.

На принимающей стороне схема EtM позволяет проверить аутентичность зашифрованного сообщения еще до операции расшифрования. Две другие схемы предполагают расшифрование открытого текста до проверки MAC.

Криптоаналитики считают EtM самой безопасной схемой, при условии что используется стойкая, не допускающая подлога функция вычисления MAC. С точки зрения безопасности, имеет смысл проверять аутентичность данных и отбрасывать плохие сообщения как можно скорее, особенно если сообщение поступило по ненадежному каналу, например через интернет. Если расшифрование производится до проверки аутентичности сообщения получателем, то противник может подменить любые сообщения при передаче и атаковать шифрование, прежде чем получатель отбросит сообщение.

В 2011 году известный исследователь безопасности Мокси Марлинспайк сформулировал *принцип криптографической обреченности*. Проще говоря, если вы вынуждены выполнить *любую* криптографическую операцию до проверки имитовставки полученного сообщения, то это *так или иначе*, но неизбежно приведет к роковому концу. Затем Мокси приводит несколько примеров, когда игнорирование принципа криптографической обреченности позволяло организовать успешные атаки на шифрование.

Формулировка принципа видится чрезмерно категоричной, особенно слова *любую* и *неизбежно*. На мой взгляд, его следует рассматривать скорее как рекомендацию, нежели непреложное правило.

С учетом сказанного, хотя EtM является самой безопасной схемой, две другие тоже можно реализовать безопасно, если принять необходимые меры предосторожности.

Еще несколько лет назад аутентификацию (вычисление MAC) и шифрование можно было применять только в виде двух отдельных операций. Но недавно стали популярны режимы шифрования с аутентификацией, например AES-GCM или ChaCha20-Poly1305, которые постепенно вытесняют режимы шифрования без аутентификации. В этом случае аутентификация встроена непосредственно в режим и не требует отдельной операции, например вычисления HMAC открытого или шифртекста. Используя шифрование с аутентификацией, не нужно выбирать между EtM и MtE. Специального термина для такой схемы не существует, но можно было бы сказать, что производится вычисление *MAC в процессе шифрования*.

Как видим, есть разные способы комбинирования аутентификации с шифрованием. А какие из них используются в наиболее популярных протоколах?

Исторически *TLS* использовался совместно со схемой MtE. Протокол TLS основан на протоколе SSL, изобретенном в 1995 году, когда превосходство

схемы EtM еще не было доказано. Впоследствии было внедрено **EtM-расширение протокола TLS**, которое позволяло клиенту и серверу TLS согласовать применение EtM вместо MtE. В версии TLS 1.2 в протокол был добавлены шифры с аутентификацией. Наконец, в версии TLS 1.3 допустимы только шифры с аутентификацией. Важно помнить, что хотя для шифров с аутентификацией отдельная операция вычисления MAC не нужна, HMAC все равно используется в TLS 1.3 как основа для **псевдослучайной функции** (Pseudorandom Function – **PRF**), необходимой для обмена ключами.

Протокол *SSH* тоже старый, как и TLS. Текущая версия 2 была опубликована в 1998 году. Исторически в SSH использовалась схема E&M. Однако схема аутентификации в SSH зависит от алгоритма вычисления MAC, а в новых алгоритмах используется схема EtM. Более современные алгоритмы со схемой EtM предпочтительнее старых со схемой E&M. Кроме того, новые реализации SSH поддерживают шифры с аутентификацией, не требующие отдельного вычисления MAC.

В протоколе *IPsec*, созданном в 1990-х годах, схема EtM использовалась с самого начала. Впоследствии в IPsec были добавлены шифры с аутентификацией, как в TLS и SSH.

Мы узнали, что такое MAC и HMAC и как они используются. Теперь перейдем к практике и посмотрим, как вычислить HMAC в командной строке.

Вычисление HMAC в командной строке

Для вычисления HMAC в командной строке необходимо сообщить программе `openssl` о типе функции вычисления MAC и о базовой функции хеширования. В качестве первой укажем `HMAC`, а в качестве второй выберем функцию `SHA-256`.

Программа `openssl` предлагает два метода вычисления HMAC:

- подкоманда `openssl dgst` – та же, которой мы пользовались при вычислении хеш-значения сообщения в предыдущей главе;
- подкоманда `openssl mac`. Это новая подкоманда, появившаяся в версии OpenSSL 3.0.

Мы рассмотрим оба метода.

Как обычно, сначала сгенерируем тестовый файл, для которого будем вычислять HMAC:

```
$ seq 20000 >somefile.txt
```

Для вычисления HMAC нужен секретный ключ. Сгенерируем его:

```
$ openssl rand -hex 32
df036c471b612f8ad099078d8e3bd9c64339e7aeab56ec75e2222c415db113de
```

Теперь вычислим HMAC командой `openssl dgst`. Для этого нужно добавить в командную строку флаг `-mac HMAC` и указать секретный ключ в шестнадцатеричном виде с помощью флага `-macopt hexkey`:

```
$ openssl dgst -sha-256 -mac HMAC -macopt \
hexkey:df036c471b612f8ad099078d8e3bd9c64339e7aeab56ec75e2222c415db113de \
somefile.txt
HMAC-SHA256(somefile.txt)= 55e18ba91be755133ab0f4dbca5d06f2e7d-
f0b6bb4cd5f16f9f2d2f7cf83372c
```

Как видим, вычислить HMAC в командной строке почти так же просто, как вычислить хеш-значение сообщения.

Ниже показано, как сделать то же самое с помощью новой подкоманды `openssl mac`:

```
$ openssl mac -digest SHA-256 -macopt \ hexkey:df036c471b-
612f8ad099078d8e3bd9c64339e7aeab56ec75e2222c415db113de \
-in somefile.txt \
HMAC 55E18BA91BE755133AB0F4DBCA5D06F2E7DF0B6BB4CD5F16F9F2D2F7CF83372C
```

Отметим, что получилось точно такое значение HMAC, но немного в другом формате.

Вычисление HMAC из программы

Напишем программу `hmac`, которая будет вычислять HMAC-SHA-256. Программа будет принимать два аргумента в командной строке.

1. Имя входного файла.
2. Секретный ключ в шестнадцатеричном виде.

Что касается API для вычисления HMAC, так OpenSSL 3.0 предлагает аж целых три:

- устаревший и нерекомендуемый низкоуровневый API, состоящий из функций с префиксом `HMAC_`. Забавно, что хотя это ни разу не `EVP` API, в нем используется `EVP_MD` API для доступа к базовым функциям вычисления хеш-значений сообщений;
- `EVP_DigestSign` API. Этот API предназначен в основном для цифровых подписей. Его можно использовать и для вычисления MAC, но такое применение плохо согласуется с интуицией;
- `EVP_MAC` API. Это новый API, специально созданный для вычисления MAC в версии OpenSSL 3.0. Мы будем использовать именно его.

Официальная документация по `EVP_MAC` API приведена на одноименной странице руководства:

```
$ man EVP_MAC
```

Интерес могут представлять также следующие страницы руководства:

```
$ man EVP_MAC-HMAC
$ man OSSL_PARAM
$ man OSSL_PARAM_int
```

Как и в предыдущих главах, сначала составим общий план.

1. Выбрать реализацию алгоритма вычисления HMAC в виде объекта `EVP_MAC`.
2. Задать параметры вычисления MAC в массиве типа `OSSL_PARAM`.
3. Создать и инициализировать контекст вычисления MAC, `EVP_MAC_CTX`.
4. Читать данные из входного файла порциями и помещать их в контекст вычисления MAC.
5. Завершить вычисление и получить HMAC.
6. Вывести HMAC на `stdout`.

Приступим же к кодированию!

Реализация программы hmac

1. Первым делом получим описание алгоритма вычисления HMAC от поставщика алгоритма по умолчанию:

   ```
   EVP_MAC* mac = EVP_MAC_fetch(NULL, OSSL_MAC_NAME_HMAC, NULL);
   ```

   Обратите внимание на параметр `OSSL_MAC_NAME_HMAC`, переданный функции. Так задается именно алгоритм вычисления HMAC, а не какой-то другой алгоритм вычисления имитовставки, например CMAC или GMAC.
2. Зададим параметры в контексте вычисления MAC:

   ```
   OSSL_PARAM params[] = {
       OSSL_PARAM_construct_utf8_string(
           OSSL_MAC_PARAM_DIGEST,
           OSSL_DIGEST_NAME_SHA2_256,
           0),
       OSSL_PARAM_construct_end()
   };
   ```

   Обратите внимание на значения `OSSL_MAC_PARAM_DIGEST` и `OSSL_DIGEST_NAME_SHA2_256`. Так мы указываем, что хотим использовать функцию хеширования SHA-256 в качестве базовой функции вычисления хеш-значения.
3. Мы выбрали алгоритм и задали параметры; теперь воспользуемся ими, чтобы создать и инициализировать контекст вычисления MAC:

   ```
   EVP_MAC_CTX* ctx = EVP_MAC_CTX_new(mac);
   EVP_MAC_init(ctx, key, key_length, params);
   ```

   Мы также поместили в контекст секретный ключ и его длину. Ключ мы сгенерировали в командной строке и перевели его в десятичную форму.
4. С инициализацией покончено. Теперь нужно читать данные из входного файла и помещать их в контекст.

```
while (!feof(in_file)) {
    size_t in_nbytes = fread(in_buf, 1, BUF_SIZE, in_file);
    EVP_MAC_update(ctx, in_buf, in_nbytes);
}
```

5. После того как все данные прошли через контекст, завершим вычисление и получим результирующее значение HMAC:

```
const size_t HMAC_LENGTH = 256 / 8;
unsigned char hmac[HMAC_LENGTH];
size_t out_nbytes = 0;
EVP_MAC_final(ctx, hmac, &out_nbytes, HMAC_LENGTH);
```

6. Больше объекты `EVP_MAC` и `EVP_MAC_CTX` не понадобятся. Освободим их, чтобы избежать утечек памяти.

```
EVP_MAC_CTX_free(ctx);
EVP_MAC_free(mac);
```

7. Наконец, напечатаем вычисленное HMAC в том формате, в каком это делает подкоманда `openssl mac`:

```
for (size_t i = 0; i < HMAC_LENGTH; ++i) {
    printf("%02X", hmac[i]);
}
printf("\n");
```

Полный исходный код программы `hmac` можно найти на GitHub по адресу https://github.com/PacktPublishing/Demystifying-Cryptography-with-OpenSSL-3/blob/main/Chapter04/hmac.c.

Выполнение программы hmac

Запустим программу вычисления HMAC и посмотрим, работает ли она:

```
$ ./hmac somefile.txt df036c471b612f8ad099078d8e3bd9c
64339e7aeab56ec75e2222c415db113de
55E18BA91BE755133AB0F4DBCA5D06F2E7DF0B6BB4CD5F16F9F2D2F7CF83372C
```

Как видим, наша программа вычислила тот же HMAC, что и программа `openssl` в предыдущем разделе. Стало быть, она работает правильно.

Резюме

В этой главе мы узнали, что такое имитовставки и чем они отличаются от хеш-значений сообщений и от цифровых подписей. Мы также познакомились с криптостойкостью MAC и атаками, для которых хорошая функция вычисления MAC должна быть неуязвима. Далее мы выяснили, что такое

функция вычисления HMAC и от чего зависит ее стойкость. И завершили теоретическую часть обзором методов объединения функций вычисления MAC с шифрованием, пояснили, какой метод лучший, и обсудили принцип криптографической обреченности.

В практической части мы рассмотрели два метода вычисления HMAC в командной строке. А затем показали, как вычислить HMAC в программе на языке C. Сравнив результаты вычисления HMAC всеми методами, мы, к своему удовлетворению, убедились, что все они дают одинаковый результат.

В следующей главе будем говорить о **функциях формирования ключа** и о том, как получить ключ шифрования из пароля.

Глава 5

Формирование ключа шифрования из пароля

В этой главе мы узнаем о формировании **ключей симметричного шифрования** из **паролей** или **парольных фраз**. В главе 2 мы говорили, что алгоритмы симметричного шифрования шифруют не паролем, а ключом. Поэтому чтобы использовать пароль для шифрования, мы должны предварительно сформировать из него ключ.

Мы дадим обзор функций формирования ключа, поддерживаемых OpenSSL. В практической части главы мы покажем, как можно сформировать ключ из пароля в командной строке и в программе на C.

В этой главе рассмотрены следующие вопросы:

- в чем разница между паролем и ключом шифрования;
- что такое функция формирования ключа;
- обзор функций формирования ключа, поддерживаемых OpenSSL;
- формирование ключа из пароля в командной строке;
- формирование ключа из пароля из программы.

Технические требования

В этой главе встречаются команды, запускаемые из командной строки, и C-код, который можно откомпилировать и выполнить. Для запуска команд понадобятся программа `openssl` и соответствующие динамические библиотеки OpenSSL. Для сборки кода на C будут нужны динамические или статические библиотеки OpenSSL, заголовки библиотек, компилятор C и компоновщик.

Мы реализуем демонстрационную программу, чтобы применить полученные знания на практике. Ее полный исходный код находится по адресу https://github.com/PacktPublishing/Demystifying-Cryptography-with-OpenSSL-3/tree/main/Chapter05.

В чем разница между паролем и ключом шифрования?

Ключ симметричного шифрования – это секретный массив битов, который непосредственно используется алгоритмом шифрования. Как правило, для алгоритма шифрования требуется ключ определенной длины, например 256 бит. В некоторых не особенно популярных шифрах допускаются ключи переменной длины, но это скорее исключение, чем правило. Ключ шифрования не предназначен для восприятия человеком: он выглядит как набор случайных данных (а зачастую они и есть случайные), слишком длинный для чтения и записи, запомнить его может разве что супермен.

С другой стороны, пароль или парольная фраза гораздо удобнее. Обычно они ориентированы на чтение человеком – особенно в фильмах, где пароли всегда простые, короткие и удобочитаемые. Неудивительно, что люди предпочитают 8-символьные пароли или 4-словные парольные фразы 256-битовым ключам, а популярные программы шифрования дают возможность шифровать данные паролями вместо таких неудобных ключей шифрования.

Однако, несмотря на все удобства, пароль непригоден для прямого использования в алгоритме шифрования. Мы должны сначала сформировать из пароля ключ, а затем воспользоваться этим ключом для шифрования. Но как это сделать? С помощью функции формирования ключа!

Что такое функция формирования ключа?

Функция формирования ключа (Key Derivation Function – **KDF**) строит секретный ключ желаемой длины из другого, тоже секретного материала, например пароля, парольной фразы, иного разделяемого секрета или комбинации открытого и закрытого ключей асимметричного шифрования. Секретный материал называется также **входным ключевым материалом** (Input Key Material – **IKM**), а порожденный на его основе секретный ключ – **выходным ключевым материалом** (Output Key Material – **OKM**). Длины IKM и OKM часто различны. На внутреннем уровне KDF, как правило, используется криптографическая функция хеширования или блочный шифр.

Функция формирования ключа на основе пароля (Password-Based Key Derivation Function – **PBKDF**) – это KDF, предназначенная для порождения секретных ключей из низкоэнтропийного IKM, например паролей. Эти секретные ключи можно использовать для симметричного шифрования. Другое популярное применение PBKDF – хеширование паролей. PBKDF порождают хеш, более стойкий к полному перебору, чем просто криптографические функции хеширования.

Некоторые функции формирования ключей используются при обмене ключами в безопасных сетевых протоколах. Но эти KDF недостаточно стойкие для формирования ключей из низкоэнтропийного IKM, поэтому мы не

станем говорить о них в этой главе, а сосредоточимся только на формировании ключей из паролей.

Типичная KDF принимает следующие параметры:

- IKM, например пароль;
- **соль** – случайно сгенерированные данные. Используется, чтобы внести дополнительную случайность в процедуру формирования ключа и препятствовать атакам с предварительно вычисленными хешами, известными также как атаки на основе радужных таблиц. Технически возможно подать на вход KDF пустую соль, но это не рекомендуется. NIST рекомендует брать соль длиной не менее 128 бит. Генерирование соли не должно зависеть от IKM. Соль может быть открытой или секретной;
- **информация**, зависящая от приложения. Она не увеличивает стойкость, но может быть полезна для связывания ключевого материала с его применением. Например, это может быть версия протокола, идентификатор алгоритма, пользователя или объекта. Использование разных значений «информации» для разных целей называется **разделением доменов**;
- базовая **псевдослучайная функция** (Pseudorandom Function – **PRF**), например функция вычисления HMAC или блочный шифр;
- зависящие от функции параметры, стойкие к полному перебору. Для многих KDF это просто счетчик итераций, но некоторые функции, например *scrypt*, принимают несколько параметров, влияющих на потребление процессора и памяти;
- желаемая длина OKM.

Не все параметры поддерживаются всеми KDF. Длины IKM и OKM поддерживает любая KDF. Хорошие PBKDF поддерживают также соль и параметры, стойкие к полному перебору. Некоторые KDF позволяют параметризовать PRF и добавлять информацию.

Из всех вышеупомянутых параметров только IKM (пароль) считается секретным. Прочие параметры несекретные и либо хранятся в открытом виде, либо подразумеваются решением, частью которого является KDF, например модулем аутентификации или программным обеспечением шифрования.

Стойкая PBKDF обладает следующими свойствами:

- она *детерминирована*, т. е. при одних и тех же входных параметрах всегда порождает один и тот же секретный ключ;
- она *необратима*, т. е. вычислительно невозможно получить исходный IKM, зная OKM. Это свойство особенно важно для применения KDF к хешированию паролей, потому что восстановление паролей по OKM ни в коем случае не должно быть простым. Но необратимость важна и для генерирования ключей шифрования, потому что если один и тот же пароль использовался для генерирования нескольких ключей и один из этих ключей оказался скомпрометирован, то крайне нежелательно, чтобы пароль можно было восстановить и использовать для восстановления других ключей;

- она должна быть стойкой к *полному перебору*, т. е. выполнение KDF должно потреблять много процессорного времени и памяти.

Остановимся подробнее на стойкости к полному перебору. Важно понимать, что типичный пароль обладает гораздо меньшей энтропией, чем случайно сгенерированный секретный ключ. Если используется хороший генератор случайных чисел, то случайно сгенерированный ключ имеет столько битов энтропии, какова его длина, обычно 128 или 256 бит.

С паролями дело обстоит иначе. В настоящее время большинство сайтов требуют, чтобы длина пароля была не меньше 8 символов, а некоторые даже начали требовать длины как минимум 12 символов. Многие сайты требуют, чтобы пароль содержал строчные и заглавные латинские буквы, цифры и специальные символы. Это дает нам приблизительно 80 возможных символов. Предположим, что вероятности всех символов пароля одинаковы. Тогда энтропию пароля можно вычислить по формуле

$$\text{Entropy} = \log_2(\text{nchar}^{\text{plen}})$$

где

- `nchar` – количество возможных символов, в нашем случае 80;
- `plen` – длина пароля.

Согласно этой формуле, энтропия 8-символьного пароля приближенно равна 51 бит. Если увеличить длину пароля до 12 символов, то энтропия составит примерно 76 бит.

Важно понимать, что PBKDF не может дать больше энтропии, чем содержится в ее входе. Если PBKDF формирует 256-битовый ключ из пароля, содержащего 76 бит энтропии, и соль открыта, то выходной ключ все равно будет содержать только 76 бит энтропии.

Как видим, типичный пароль содержит не так много энтропии. Поэтому PBKDF не может полагаться только на энтропию пароля, чтобы эффективно противостоять атакам с полным перебором. Основанные на пароле KDF должны еще потреблять много вычислительных ресурсов. Многие PBKDF решают эту проблему, выполняя большое число итераций базовой PRF. В то же время PBKDF не должны быть слишком медленными, поскольку формирование ключа часто выполняется в процессе взаимодействия с человеком, например при проверке введенного пароля. Так что желательно, чтобы PBKDF считала доли секунды.

Как видим, криптостойкость необязательно совпадает с количеством битов энтропии. Стойкость PBKDF можно увеличить, выполняя много итераций базовой PRF. Отметим, однако, что стойкость зависит от числа итераций не линейно, а логарифмически. Например, чтобы увеличить стойкость на 8 бит, понадобится всего 256 (2^8) итераций, но чтобы увеличить ее на 16 бит, нужно уже 65 536 (2^{16}) итераций – довольно много.

Правда, в последнее время стало модно взламывать пароли с помощью массового распараллеливания на **графических процессорах** (**GPU**), **интегральных схемах специального назначения** (Application-Specific Integration Circuit – **ASIC**) или **программируемых пользователем вентильных**

матрицах (**ППВМ**), которые могут очень быстро выполнять простые операции. Поэтому одной вычислительной трудоемкости еще недостаточно для хорошей PBKDF. Стойкая PBKDF должна также потреблять много памяти, тогда распараллеливание взлома пароля будет ограничено объемом доступной памяти, так что задействовать вычислительную мощь GPU, ASIC или ППВМ в полной мере не удастся.

В следующем разделе мы расскажем, какие PBKDF поддерживает OpenSSL.

Обзор функций формирования ключа, поддерживаемых OpenSSL

OpenSSL 3.0 поддерживает несколько функций формирования ключа, но только две из них пригодны для формирования ключей из паролей, а именно *scrypt* и *PBKDF2*.

PBKDF2 – популярная PBKDF-функция, описанная и рекомендованная стандартом PKCS #5. В качестве PRF в ней используется функция вычисления HMAC, например HMAC-SHA-256. PBKDF2 поддерживает настраиваемое число итераций, от нее можно добиться высокого потребления процессора, но не памяти. В 2021 году проект **Open Web Application Security Project** (**OWASP**, открытый проект по безопасности веб-приложений) рекомендовал 310 000 итераций для PBKDF2 с HMAC-SHA-256 в качестве псевдослучайной функции.

Scrypt – лучшая из имеющихся PBKDF в OpenSSL 3.0. Она может быть настроена не только на высокую вычислительную трудоемкость, но и на высокое потребление памяти. На внутреннем уровне в scrypt используется PBKDF2 с HMAC-SHA-256. Scrypt позволяет настраивать объем вычислений, потребление памяти и степень параллелизма. В 2021 году OWASP рекомендовал для scrypt следующие параметры, стойкие к полному перебору: `N` = 65 536, `r` = 8, `p` = 1. Хотя scrypt можно использовать для порождения ключей симметричного шифрования, она также применяется в методах хеширования паролей в современных версиях GNU/Linux. Еще одно применение scrypt – доказательство выполнения работы в некоторых криптовалютах, например Litecoin и Dogecoin.

Одношаговые KDF и HKDF не годятся в качестве PBKDF, поскольку не являются вычислительно трудоемкими. Кроме того, они не принимают стойких к полному перебору параметров.

Псевдослучайные функции из стандартов ANSI X9.42, ANSI X9.63 и TLS1 не являются вычислительно трудоемкими и требуют специфических для TLS параметров.

KDF, применяемая в SSH, также не является вычислительно трудоемкой и требует специфических для SSH параметров.

Так что scrypt – лучший из доступных нам вариантов PBKDF.

Формирование ключа из пароля в командной строке

Для порождения ключа шифрования из пароля применяется новая подкоманда `openssl kdf`, добавленная в версии OpenSSL 3.0. Она документирована на странице руководства `openssl-kdf`:

```
$ man openssl-kdf
```

Прежде чем формировать ключ, сгенерируем 128-битовую соль:

```
$ openssl rand -hex 16
cf0e0acf943629ecffea41c87bab94d4
```

Теперь можно сформировать 256-битовый ключ симметричного шифрования. Воспользуемся KDF *scrypt* с рекомендуемыми OWASP параметрами, стойкими к полному перебору, и паролем `SuperPa$$w0rd`:

```
$ openssl kdf \
  -keylen 32 \
  -kdfopt 'pass:SuperPa$$w0rd' \
  -kdfopt hexsalt:cf0e0acf943629ecffea41c87bab94d4 \
  -kdfopt n:65536 -kdfopt r:8 -kdfopt p:1 \
  SCRYPT
D0:3D:31:A1:A2:2A:F6:68:99:B3:02:22:60:3B:D7:21:5B:15:5B:80:2B: 85:33:36:E6:3B:AB:F9:EE:8F
:FE:C7
```

Обратите внимание, что аргумент командной строки, содержащий пароль, заключен в одиночные кавычки. Это сделано для того, чтобы избежать специальной интерпретации символов `$` оболочкой. Другой метод задания специальных символов – передать пароль в шестнадцатеричном виде с помощью аргумента `-kdfopt hexpass:` вместо `-kdfopt pass:`.

Формирование ключа из пароля из программы

Мы напишем программу `kdf`, которая будет формировать ключ из пароля. Программа будет принимать два аргумента:

1) пароль;
2) соль в шестнадцатеричном виде.

Мы не станем принимать N, r и стойкие к полному перебору параметры scrypt, потому что хотим упростить пример и его использование. Поэтому просто зададим параметры, рекомендуемые OWASP.

OpenSSL 3.0 предлагает следующие API для формирования ключа:

- унаследованные функции `PKCS5_PBKDF2_HMAC()`, `PKCS5_PBKDF2_HMAC_SHA1()` и `EVP_PBE_scrypt()`, предназначенные для конкретных KDF;

- API EVP_PKEY, предназначенный для использования совместно с асимметричной криптографией и состоящий из одной функции EVP_PKEY_derive(). Эта функция рассчитана прежде всего на формирование ключа без пароля в методе обмена ключами в безопасных сетевых протоколах, например **Диффи–Хеллмана**, но поддерживает также формирование ключа и пароля с помощью алгоритма *scrypt*;
- новый API EVP_KDF, специально созданный для KDF OpenSSL 3.0. Именно он рекомендуется, и мы им воспользуемся.

API EVP_KDF и его использование совместно с *scrypt* документированы на следующих страницах руководства:

```
$ man EVP_KDF
$ man EVP_KDF-SCRYPT
```

Как обычно, начнем с составления общего плана.

1. Выбрать реализацию алгоритма формирования ключа в форме объекта EVP_KDF.
2. Создать параметры формирования ключа в виде массива OSSL_PARAM.
3. Создать контекст формирования ключа, EVP_KDF_CTX.
4. Сформировать ключ.
5. Напечатать ключ на stdout.

Теперь воплотим этот план в коде.

Реализация программы kdf

1. Сначала получим описание алгоритма формирования ключа *scrypt* от поставщика алгоритма по умолчанию:

```
EVP_KDF* kdf = EVP_KDF_fetch(
    NULL, OSSL_KDF_NAME_SCRYPT, NULL);
```

2. Затем создадим параметры формирования ключа:

```
uint64_t scrypt_n = 65536;
uint32_t scrypt_r = 8;
uint32_t scrypt_p = 1;
OSSL_PARAM params[] = {
    OSSL_PARAM_construct_octet_string(
        OSSL_KDF_PARAM_PASSWORD,
        (char*)password, strlen(password)),
    OSSL_PARAM_construct_octet_string(
        OSSL_KDF_PARAM_SALT, (char*)salt, salt_length),
    OSSL_PARAM_construct_uint64(
        OSSL_KDF_PARAM_SCRYPT_N, &scrypt_n),
    OSSL_PARAM_construct_uint32(
        OSSL_KDF_PARAM_SCRYPT_R, &scrypt_r),
```

```
    OSSL_PARAM_construct_uint32(
        OSSL_KDF_PARAM_SCRYPT_P, &scrypt_p),
    OSSL_PARAM_construct_end()
};
```

К сожалению, функции OSSL_PARAM_construct_*() принимают только неконстантные указатели, поэтому необходимо приводить типы указателей на пароль и соль и создавать переменные scrypt_n, scrypt_r и scrypt_p как неконстантные.

3. Далее создаем контекст формирования ключа:

```
EVP_KDF_CTX* ctx = EVP_KDF_CTX_new(kdf);
```

4. Теперь все части на месте и можно сформировать ключ:

```
EVP_KDF_derive(ctx, key, KEY_LENGTH, params);
```

5. Объекты EVP_KDF и EVP_KDF_CTX больше не нужны, поэтому освободим их, чтобы избежать утечек памяти:

```
EVP_KDF_CTX_free(ctx);
EVP_KDF_free(kdf);
```

6. Ключ сформирован. Напечатаем его в таком же формате, как подкоманда openssl kdf:

```
for (size_t i = 0; i < KEY_LENGTH; ++i) {
    if (i != 0)
        printf(“:”);
    printf(“%02X”, key[i]);
}
printf(“\n”);
```

Полный исходный код программы kdf находится по адресу https://github.com/PacktPublishing/Demystifying-Cryptography-with-OpenSSL-3/blob/main/Chapter05/kdf.c.

Выполнение программы kdf

Запустим нашу программу формирования ключа и посмотрим, как она работает:

```
$ ./kdf ‘SuperPa$$w0rd’ cf0e0acf943629ecffea41c87bab94d4
D0:3D:31:A1:A2:2A:F6:68:99:B3:02:22:60:3B:D7:21:5B:15:5B:80:2B: 85:33:36:E6:3B:AB:F9:EE:8F
:FE:C7
```

Как видим, наша программа kdf сформировала тот же ключ, что подкоманда openssl kdf в предыдущем разделе. Это позволяет надеяться, что наша программа работает правильно.

Резюме

В этой главе мы узнали об идее формирования ключа шифрования из пароля. Затем поговорили о том, что такое функция формирования ключа и какие требования предъявляются к хорошей PBKDF. Теоретическую часть мы закончили обзором функций формирования ключа, поддерживаемых OpenSSL, и рекомендациями по выбору KDF для формирования ключа из пароля.

В практической части мы узнали, как формировать ключ симметричного шифрования из пароля в командной строке и из программы на C. Мы сравнили результаты и убедились, что оба метода формируют один и тот же ключ.

В следующей главе мы начнем изучать асимметричную криптографию.

Часть III

АСИММЕТРИЧНАЯ КРИПТОГРАФИЯ И СЕРТИФИКАТЫ

В этой части мы узнаем об асимметричной криптографии, открытых и закрытых ключах шифрования, цифровых подписях и их проверке, а также о сертификатах X.509. Мы сравним разные асимметричные шифры, в т. ч. RSA и эллиптические кривые. Все рассматриваемые технологии иллюстрируются примерами вызова из командной строки и кода на C.

Эта часть включает следующие главы:

- главу 6 «Асимметричное шифрование и расшифрование»;
- главу 7 «Цифровые подписи и их проверка»;
- главу 8 «Сертификаты X.509 и инфраструктура открытых ключей».

Глава 6

Асимметричное шифрование и расшифрование

В этой главе мы узнаем об **асимметричном шифровании**: как оно работает и как открытый и закрытый ключи используются при шифровании и расшифровании. Будут рассмотрены следующие темы:

- что такое асимметричное шифрование;
- что такое атака с человеком посередине;
- какие виды асимметричного шифрования доступны в OpenSSL;
- сеансовый ключ;
- криптостойкость RSA;
- генерирование пары ключей RSA;
- шифрование и расшифрование с помощью RSA в командной строке;
- шифрование с помощью RSA из программы;
- что такое очередь ошибок в OpenSSL;
- расшифрование с помощью RSA из программы.

Технические требования

В этой главе встречаются команды, запускаемые из командной строки, и C-код, который можно откомпилировать и выполнить. Для запуска команд понадобятся программа `openssl` и соответствующие динамические библиотеки OpenSSL. Для сборки кода на C будут нужны динамические или статические библиотеки OpenSSL, заголовки библиотек, компилятор C и компоновщик.

Мы реализуем демонстрационную программу, чтобы применить полученные знания на практике. Ее полный исходный код находится по адресу https://github.com/PacktPublishing/Demystifying-Cryptography-with-OpenSSL-3/tree/main/Chapter06.

Что такое асимметричное шифрование

В главе 2 мы видели, что в алгоритмах симметричного шифрования для шифрования и расшифрования используется один и тот же ключ. А в **алгоритмах асимметричного шифрования** ключа два: **открытый** и **закрытый**. Открытый и соответствующий ему закрытый ключ называются **парой ключей**. Открытый ключ применяется для шифрования, а закрытый – для расшифрования.

Зачем нужно два ключа? Почему нельзя всегда использовать симметричное шифрование с одним ключом? Если коротко, то асимметричное шифрование необходимо, когда трудно или невозможно отправить секретный ключ и гарантировать, что его никто не перехватит по дороге. Представим, что Алиса хочет отправить Бобу сообщение по небезопасному каналу, например через интернет. Промежуточные серверы, через которые транспортируется сообщение, могут прослушивать проходящий трафик и даже изменять его. Алиса не хочет, чтобы кто-то прослушал ее сообщение, поэтому решает его зашифровать. Но как передать ключ шифрования? Если бы между Алисой и Бобом был безопасный канал связи для отправки ключа, то они могли бы просто послать сообщение по этому каналу без всякого шифрования. Поэтому Алиса и Боб решают воспользоваться асимметричной криптографией.

1. Боб генерирует пару ключей и отправляет свой открытый ключ Алисе.
2. Алиса шифрует сообщение открытым ключом Боба и отправляет ему зашифрованное сообщение.
3. Боб расшифровывает сообщение своим закрытым ключом.

Если противник перехватит трафик и заполучит открытый ключ Боба и сообщение, то расшифровать сообщение он все равно не сможет. Это можно сделать только закрытым ключом Боба. Поэтому Боб должен держать свой закрытый ключ в секрете и никому его не показывать.

Все это работает, если потенциальный противник может только пассивно прослушивать трафик, но не может его изменять. А если может? Тогда он мог бы попытаться организовать атаку с человеком посередине.

Что такое атака с человеком посередине

Атакой с человеком посередине, или атакой с посредником (Man in the Middle – **MITM**), называется атака, в которой противник прослушивает транзитный трафик и изменяет его, пытаясь выдать себя за получателя – отправителю и за отправителя – получателю.

Продемонстрируем возможную атаку для рассмотренного выше сценария. Предположим, что Мэллори играет роль человека посередине, пытающегося восстановить открытый текст по зашифрованному сообщению от Алисы Бобу. Атака развивается следующим образом:

1) Боб генерирует пару ключей и отправляет свой открытый ключ Алисе;
2) Мэллори генерирует собственную пару ключей. Она перехватывает открытый ключ, отправленный Алисе, и сохраняет его для использования

в будущем. Вместо открытого ключа Боба Мэллори отправляет Алисе свой открытый ключ, выдавая его за ключ Боба;
3) Алиса шифрует свое сообщение открытым ключом Мэллори, думая, что это открытый ключ Боба. Затем Алиса отправляет зашифрованное сообщение Бобу;
4) Мэллори перехватывает сообщение Алисы. Она расшифровывает его своим закрытым ключом, а затем перешифровывает открытым ключом Боба и отправляет зашифрованное сообщение Бобу;
5) в итоге Алиса думает, что отправила сообщение Бобу и больше его никто не читал. А Боб думает, что получил от Алисы сообщение, больше никем не прочитанное. Мэллори же получила секретную информацию и осталась незамеченной.

Как защититься от атаки с человеком посередине? Есть разные методы, но все они требуют надежного канала связи, доступного хотя бы в течение короткого времени до начала основного взаимодействия.

Личная встреча

Обе стороны (Алиса и Боб) могут встретиться лично и обменяться своими открытыми ключами или проверить их. Метод тривиальный, но не практичный, особенно если Алиса и Боб живут далеко друг от друга. Впрочем, иногда и он оказывается полезным. Например, на некоторых ИТ-конференциях предусмотрены **встречи для подписания ключей**. Люди, прибывающие на конференцию из разных частей света, собираются и предъявляют друг другу свои идентификаторы, например паспорта или водительские права, и свои цифровые отпечатки ключей в системе **Pretty Good Privacy** (**PGP**). Участники встречи могут также подписывать ключи друг друга, отсюда и название мероприятия.

Проверка цифрового отпечатка ключа по телефону

Открытый цифровой отпечаток ключа – это, по существу, полное или частичное криптографическое хеш-значение, представленное в шестнадцатеричном виде или в коде base64. После обмена открытыми ключами через интернет – например, по электронной почте – Алиса и Боб созваниваются и проверяют отпечатки друг друга. В таком случае формой аутентификации служат голоса Алисы и Боба. К сожалению, в наши дни такой метод проверки ключей стал не так практичен и надежен, как раньше. Быстродействующие процессоры и развитые методы криптоанализа заставляют Алису и Боба использовать более длинные отпечатки, а прогресс технологий глубоких подделок[1] открывает возможность фальсифицировать человеческий голос

[1] Специально для не признающих русского языка – «дипфейк». – *Прим. перев.*

в режиме реального времени. Но даже при таких условиях верификация по голосу остается неплохим методом аутентификации. Чтобы фальсифицировать обмен ключами, противнику необходимо провести атаку с человеком посередине на два разных канала связи и воспользоваться продвинутой технологией глубоких подделок.

Разделение ключа

Ключ можно разделить на несколько частей и отправить их получателю разными методами, например по электронной почте, в мессенджере, опубликовать на файлообменном сервере и отправить флешку обычной почтой. Идея в том, что противнику будет трудно перехватить и фальсифицировать сообщения на всех каналах. После того как получатель соберет ключ, его отпечаток можно дополнительно проверить по телефону.

Подписание ключа доверенной третьей стороной

Третья сторона – назовем ее Томас, – которой Алиса и Боб доверяют, может заверить открытый ключ Боба своей цифровой подписью, подтвердив, что он принадлежит именно Бобу. Затем Алиса сможет проверить эту подпись. Этот метод требует, чтобы Алиса сначала получила открытый ключ Томаса по какому-то надежному каналу. Похожая, но более сложная модель верификации ключа используется в **сети доверия** PGP и в **инфраструктуре открытых ключей** (Public Key Infrastructure – **PKI**), основанной на сертификатах X.509. Дополнительные сведения о цифровых подписях см. в главе 7, а о сертификатах X.509 – в главе 8.

Теперь Алиса и Боб знают, как безопасно обменяться открытыми ключами. Но какой алгоритм асимметричного шифрования использовать, чтобы зашифровать сообщения, которые они хотят отправлять друг другу? Этому вопросу посвящен следующий раздел.

Какие виды асимметричного шифрования доступны в OpenSSL

В библиотеке OpenSSL реализовано несколько алгоритмов асимметричной криптографии, но лишь один из них позволяет *непосредственно* шифровать данные. Это алгоритм **Ривеста–Шамира–Адлемана** (**RSA**).

Другие доступные алгоритмы асимметричной криптографии, например **Digital Signature Algorithm** (**DSA**) и **Elliptic Curve Digital Signature Algorithm** (**ECDSA**), используются для цифровых подписей. Алгоритмы **Диффи–Хелммана** (**DH**) и **Диффи–Хеллмана на эллиптических кривых** (Elliptic

Curve Diffie–Hellman – **ECDH**) используются для обмена ключами в протоколе TLS.

Как мы знаем, у симметричных криптографических ключей нет структуры; это просто массивы случайных битов. Напротив, в асимметричной криптографии ключи структурированы, т. е. ключ может состоять из нескольких компонентов, к которым предъявляются определенные требования, например это должно быть простое число, а не случайный битовый массив. У каждого алгоритма асимметричной криптографии своя структура открытого и закрытого ключей. Потому-то и форматы асимметричных ключей разные, например есть ключи RSA и ключи ECDSA.

Почему выше мы сказали, что RSA – единственный алгоритм асимметричной криптографии, способный *непосредственно* шифровать данные? Что значит «непосредственно»? Нельзя ли использовать другие алгоритмы для *опосредованного* шифрования? Можно. Существует алгоритм **ElGamal**, в котором для выполнения асимметричного шифрования используется алгоритм DH с ключами DSA или алгоритм ECDH с ключами ECDSA. В ElGamal создается эфемерный (временный) ключ DSA/ECDSA в дополнение к ключу DSA/ECDSA, используемому для шифрования. При этом для выработки разделяемого секрета применяется обмен ключами DH/ECDH, затем из разделяемого секрета формируется ключ симметричного шифрования, и, наконец, этим ключом зашифровываются данные. Как видим, это сложнее, чем использование RSA. Существуют и другие схемы формирования симметричного ключа на основе определенного и эфемерного асимметричного ключа, например **интегрированная схема шифрования** (Integrated Encryption Scheme – **IES**) и **гибридное шифрование открытым ключом** (Hybrid Public Key Encryption – **HPKE**). В OpenSSL алгоритмы ElGamal, IES, HPKE и им подобные не реализованы. Поэтому в примерах асимметричного шифрования в этой главе будет использоваться RSA.

Ключ симметричного шифрования в алгоритме ElGamal называется **сеансовым ключом**. Идея использования сеансового ключа встречается не только в ElGamal. Он используется также в RSA и в таких безопасных сетевых протоколах, как TLS, SSH и IPsec.

Что такое сеансовый ключ

Важно понимать, как асимметричное шифрование используется на практике. Асимметричные шифры, тот же RSA, гораздо медленнее симметричных, например AES. Поэтому обычно фактически отправляемые данные шифруются не с помощью RSA. Вместо этого отправитель генерирует симметричный сеансовый ключ. Затем фактические данные зашифровываются симметричным алгоритмом, к примеру AES, с этим сеансовым ключом, который сам зашифрован RSA. Получатель сначала расшифровывает сеансовый ключ, применяя RSA, а затем – фактические данные, применяя AES

с сеансовым ключом. Такая схема часто называется **гибридной схемой шифрования**.

В безопасных сетевых протоколах, таких как TLS, SSH и IPsec, сеанс связи начинается операцией **квитирования**, частью которой является **обмен ключами**. В старых версиях протоколов обмен ключами включал в себя генерирование сеансового ключа одной стороной, шифрование его с помощью RSA и отправку зашифрованного ключа другой стороне. В современных версиях стороны пользуются для обмена ключами методами DH или ECDH, позволяющими обеим сторонам выработать один и тот же ключ, который они затем используют для шифрования полезных данных. Дополнительные сведения о квитировании в TLS см. в главе 9.

Чтобы гибридное шифрование было стойким, таковыми должны быть симметричное шифрование сеансовым ключом и асимметричное шифрование RSA-ключом. В следующем разделе описывается криптостойкость алгоритма RSA.

Криптостойкость RSA

Криптостойкость RSA зависит от трудности задачи факторизации целого числа. Напомним, что в этой задаче требуется разложить положительное целое число в произведение простых множителей. В настоящее время неизвестен алгоритм ее решения за полиномиальное время, который можно было бы выполнить на классическом компьютере, поэтому задача остается вычислительно очень трудоемкой. А насколько большие числа используются в RSA? В типичном случае они состоят из сотен или тысяч десятичных цифр, т. е. тысяч или десятков тысяч двоичных цифр. Очевидно, что обычные целые типы языка C недостаточно широки, чтобы над ними можно было выполнять математические операции, требуемые RSA. Поэтому в состав OpenSSL входит подбиблиотека для работы с большими целыми числами. Типы и функции, принадлежащие этой подбиблиотеке, имеют префикс `BN_`.

И открытый, и закрытый ключи RSA состоят из нескольких больших или малых целых чисел, которые называются модулем и показателем степени, а также необязательных простых чисел и коэффициентов. Размером ключа RSA считается размер модуля. Например, если ключ RSA включает 2048-битовый модуль, то размер ключа будет равен 2048 бит.

Важно иметь в виду, что стойкость ключа RSA гораздо меньше размера ключа. Так, стойкость 2048-битового ключа составляет всего 112 бит. Стойкость ключа RSA зависит от размера нелинейно. Чтобы увеличить стойкость на несколько битов, размер ключа придется увеличить существенно.

В таблице ниже приведены оценки стойкости ключей RSA разного размера, полученные Национальным институтом стандартов и технологий США (NIST).

Таблица 6.1. Стойкость ключей RSA разных размеров

Размер ключа RSA в битах	Стойкость в битах
1024	80
2048	112
3072	128
4096	152
7680	192
8192	200
15 360	256

Как видим, для достижения такой же стойкости, как у шифра AES-256, необходим ключ RSA длиной 15 360 бит! Есть ли какие-нибудь недостатки у длинных ключей RSA? К сожалению, есть.

- Более низкая производительность шифрования и расшифрования, подписания и проверки подписи. С ростом длины ключа производительность падает быстрее, чем линейно. При удвоении длины ключа в диапазоне 1024–4096 бит – например, с 1024 до 2048 бит – шифрование и подписание замедляются в 7 раз, а расшифрование и проверка подписи – в 3 раза. Чтобы помочь вам составить представление о быстродействии RSA, скажем, что при длине ключа 4096 бит процессор Intel i5 может выполнить тысячи операций шифрования или сотни операций расшифрования в секунду. Как видно, шифрование в RSA гораздо быстрее, чем расшифрование. Если вам любопытно, проверьте быстродействие на своей машине с помощью команды `openssl speed rsa`.
- RSA выводит зашифрованные данные блоками, длина которых равна длине ключа. Учитывая, что RSA обычно применяется для шифрования симметричного ключа и, возможно, вектора инициализации, получается, что RSA-шифрование увеличивает размер входных данных в несколько раз. Например, 256-битовый симметричный ключ и 128-битовый вектор инициализации после шифрования 2048-битовым ключом RSA порождают шифртекст длиной 2048 бит. Чем длиннее ключ RSA, тем выше удлинение в результате шифрования. Технически результатом RSA-шифрования является очень длинное целое число, поэтому можно удалить начальные нули и немного уменьшить длину шифртекста. Но такая игра не стоит свеч, и ни одна популярная программа шифрования этим не занимается.
- Размер RSA-подписи в стандартном формате PKCS #1 также равен размеру ключа RSA. Длинные подписи неудобны.
- Чем длиннее ключ RSA, тем дольше его генерировать, например на ПК с процессором Intel i5 4096-битовый ключ RSA генерируется меньше секунды, но генерирование 15 360-битового ключа займет 2 минуты.

Как видим, налицо компромисс между безопасностью и производительностью. Объем памяти, занимаемой ключом RSA, зашифрованным блоком или подписью, – это еще меньшее из зол. Настоящая беда в том, что для получения нескольких лишних битов стойкости приходится увеличивать длину

ключа на гораздо большее число битов и расплачиваться за это заметным снижением производительности. К счастью, процессоры, в т. ч. установленные в мобильных и встраиваемых устройствах, становятся все быстрее.

Но каков приемлемый компромисс? Какую длину ключа RSA выбрать? NIST рекомендует следующие размеры ключей:

- не менее 2048 бит до 2030 года;
- не менее 3072 бит после 2030 года.

На момент написания книги 2048 бит – самый популярный и минимально приемлемый размер ключей RSA. В большинстве основанных на RSA TLS-сертификатах веб-серверов используются 2048-битовые ключи. Следующим шагом, скорее всего, будут 4096-битовые сертификаты. Во многих старых статьях в интернете отстаивалась идея о компромиссном размере, 3072 бита, но в итоге она была отвергнута. В современных статьях обычно сравниваются 2048- и 4096-битовые сертификаты.

Лично я думаю, что лучше иметь какой-то запас стойкости, поэтому предпочитаю ключи длиной 4096 бит. Почему не меньше? Потому что компьютеры, с которыми я имею дело, достаточно быстро работают с такими ключами. Почему не больше? Потому что 4096 бит дают достаточный запас стойкости, а некоторые программы, особенно старые, не поддерживают ключи RSA длиннее 4096 бит. Например, даже современные версии GnuPG не позволяют создавать пару ключей RSA длиннее 4096 бит. Да и помимо всего прочего, 4096 – хорошее число, потому что является степенью двойки, в отличие от 3072.

Я также полагаю, что еще лучше использовать **криптографию на эллиптических кривых** (Elliptic Curve Cryptography – **ECC**) вместо RSA в тех случаях, когда это так же просто, как использование RSA, например в TLS-сертификатах и ключах SSH. Но для шифрования RSA использовать гораздо проще, чем ECC. Мы еще вернемся к ECC в главе 7.

Что станется со стойкостью RSA после пришествия **квантовых вычислений**? К сожалению, асимметричная криптография в том виде, в каком мы ее знаем, включая RSA и ECC, будет полностью уничтожена квантовыми компьютерами, и в настоящее время еще нет стандартизованной и прошедшей проверку временем замены. Хорошая же новость заключается в том, что многие криптографы активно работают над постквантовыми криптографическими алгоритмами. На момент написания книги NIST проводит работу над стандартизацией постквантовой криптографии в форме конкурса квантово-стойких алгоритмов по аналогии с конкурсами AES и SHA-3. Победители конкурса станут постквантовым стандартом. Самый оптимистический прогноз говорит, что квантовые компьютеры, способные вскрывать современную асимметричную криптографию, будут созданы в 2030-х годах. Так что есть надежда, что постквантовые криптоалгоритмы будут стандартизованы и включены в популярные криптографические библиотеки типа OpenSSL еще до того, как мощные квантовые компьютеры станут доступны крупным игрокам – правительствам и корпорациям.

Но все это – дело будущего. А что мы можем сделать уже сейчас? Начнем с генерирования пары ключей RSA.

Генерирование пары ключей RSA

Программа openssl предоставляет две подкоманды для генерирования пары ключей RSA: genrsa и genpkey. Первая умеет генерировать только пару ключей RSA, вторая – более общая, она может генерировать любую пару ключей, поддерживаемую OpenSSL. genrsa объявлена нерекомендуемой начиная с OpenSSL 3.0, поэтому будет использовать genpkey.

Подкоманда openssl genpkey документирована на странице руководства openssl-genpkey:

```
man openssl-genpkey
```

Откуда взялось имя genpkey? В OpenSSL есть понятие **открытого или закрытого ключа** (Public or Private Key – **PKEY**). Тут важно прояснить одно недоразумение. В документации по OpenSSL встречаются упоминания об открытых и закрытых ключах. Очень часто, говоря о закрытом ключе, документация на самом деле имеет в виду пару ключей. Это относится к документации как по командным инструментам, так и по OpenSSL API. Например, в описательной части страницы руководства по openssl-genpkey сказано: «Команда genpkey генерирует закрытый ключ». Но если генерируется только закрытый ключ, то как сгенерировать соответствующий ему открытый? В документации говорится, что открытый ключ можно извлечь из закрытого, что для неопытных пользователей звучит странно. Если быть точным, то, что в OpenSSL и в стандартах PKCS обычно называется закрытым ключом, на самом деле является структурой данных, в которой хранится информация, достаточная для построения закрытого и открытого ключей. Часть этой информации является общей для обоих ключей, в т. ч. модуль RSA. Под открытым ключом понимается похожая структура данных, содержащая только информацию об открытом ключе. Поэтому для простоты мы можем предположить, что любое упоминание о закрытом ключе в OpenSSL означает пару ключей, а при упоминании открытого ключа имеется в виду ровно это – открытый ключ.

Надеюсь, что развеял больше тумана, чем нагнал! А теперь сгенерируем свою первую пару ключей RSA:

```
$ openssl genpkey \
      -algorithm RSA \
      -pkeyopt rsa_keygen_bits:4096 \
      -out rsa_keypair.pem
```

С помощью параметра -pkeyopt rsa_keygen_bits:4096 мы указали, что хотим сгенерировать 4096-битовую пару ключей. Сгенерированная пара будет записана в файл rsa_keypair.pem.

Открыв файл с парой ключей в текстовом редакторе, мы увидим что-то вроде

```
-----BEGIN PRIVATE KEY-----
MIIJQwIBADANBgkqhkiG9w0BAQEFAASCCS0wggkpAgEAAoICAQDC2/ sxiM72yGgy
... много данных в коде base64 ...
```

```
FZTCpfZK4ecBXkHHaVnHBmdS10EyngU=
-----END PRIVATE KEY-----
```

Пара ключей хранится в формате **почты с повышенной секретностью** (Privacy Enhanced Mail – **PEM**). PEM – формат, применяемый в OpenSSL по умолчанию для хранения ключей и сертификатов. Хотя в названии упомянута «почта», этот формат используется не только для почты. На самом деле PEM – обертка двоичных данных кодом Base64, содержащая текстовый заголовок (строка `BEGIN`) и текстовый концевик (строка `END`). Если удалить заголовок и концевик из PEM-файла и декодировать содержимое, то получится пара ключей в формате **особых правил кодирования** (Distinguished Encoding Rules – **DER**). DER – еще один популярный формат хранения ключей и сертификатов, поддерживаемый OpenSSL. Это двоичное представление структур данных, описанных в **абстрактной синтаксической нотации версии 1** (Abstract Syntax Notation One – **ASN.1**). ASN.1 – язык, позволяющий описывать подлежащие сериализации структуры данных платформенно-независимым способом, например на этом языке пара ключей RSA описывается как ASN.1-последовательность ASN.1-целых чисел.

Заглянуть внутрь структуры сгенерированной пары ключей позволяет подкоманда `openssl pkey`, документированная на странице руководства `openssl-pkey`:

```
man openssl-pkey
```

Имеется также похожая подкоманда, относящаяся только к RSA, `openssl rsa`. Она объявлена нерекомендуемой начиная с версии OpenSSL 3.0, наряду с `openssl genrsa`, `openssl rsautl` и другими подкомандами для конкретных типов ключей.

Вот как выглядит распечатка структуры пары ключей:

```
$ openssl pkey -in rsa_keypair.pem -noout -text
Private-Key: (4096 bit, 2 primes)
modulus:
...
publicExponent: 65537 (0x10001)
privateExponent:
...
prime1:
...
prime2:
...
exponent1:
...
exponent2:
...
coefficient:
...
```

Очень хорошо, что мы сгенерировали пару ключей. А теперь нужно извлечь из нее открытый ключ, чтобы передать его той стороне, которая будет шифровать для нас данные. Делиться всей парой ключей ни в коем случае нельзя – только открытым ключом.

Вот как извлекается открытый ключ из пары:

```
$ openssl pkey \
    -in rsa_keypair.pem \
    -pubout \
    -out rsa_public_key.pem
```

Обратите внимание на флаг `-pubout`. Так мы указываем, что хотим видеть на выходе только открытый ключ, а не всю пару.

Результирующий файл `rsa_public_key.pem` не содержит никакой секретной информации, им можно безбоязненно поделиться. Можно заглянуть внутрь файла с открытым ключом:

```
$ openssl pkey -pubin -in rsa_public_key.pem -noout -text
Public-Key: (4096 bit)
Modulus:
...
Exponent: 65537 (0x10001)
```

Обратите внимание на флаг `-pubin`. Так мы указываем, что подкоманда `openssl pkey` должна рассматривать вход как открытый ключ, а не как пару ключей.

Легко видеть, что в файле открытого ключа гораздо меньше информации, чем в файле пары ключей, – только модуль и открытый показатель степени.

Шифрование и расшифрование с помощью RSA в командной строке

В программе `openssl` есть две подкоманды для RSA-шифрования: `pkeyutl` и объявленная нерекомендуемой подкоманда `rsautl`, относящаяся только к RSA. Разумеется, мы будем использовать `pkeyutl`. Она документирована на странице руководства `openssl-pkeyutl`:

```
man openssl-pkeyutl
```

Как уже было сказано, RSA обычно применяется для шифрования сеансового ключа, который затем используется для шифрования полезных данных. Сгенерируем 256-битовый сеансовый ключ:

```
$ openssl rand -out session_key.bin 32
```

Воспользуемся подкомандой `openssl pkeyutl` с нашим открытым ключом RSA для шифрования сеансового ключа:

```
$ openssl pkeyutl \
    -encrypt \
    -in session_key.bin \
    -out session_key.bin.encrypted \
```

```
    -pubin \
    -inkey rsa_public_key.pem \
    -pkeyopt rsa_padding_mode:oaep
```

Обратите внимание на параметр `-pkeyopt rsa_padding_mode:oaep`. Он означает, что при RSA-шифровании нужно использовать **оптимальное дополнение асимметричного шифрования** (Optimal Asymmetric Encryption Padding – **OAEP**), определенное в стандарте PKCS #1 v2.0. Настоятельно рекомендуется использовать именно этот тип дополнения, потому что он считается самым стойким. Дополнение типа PKCS #1 v1.5 может сделать ваш шифртекст уязвимым для атаки с оракулом дополнения Блейхенбахера.

Посмотрим на созданные нами файлы:

```
$ cksum session_key.bin*
186975932 32 session_key.bin
2324953448 512 session_key.bin.encrypted
```

Отметим, что длина входного сеансового ключа равна всего 32 байта (256 бит), но в зашифрованном виде он занимает уже 512 байт (4096 бит), поскольку размер блока выходного шифртекста RSA равен длине ключа.

Попробуйте зашифровать сеансовый ключ несколько раз и убедитесь, что размер зашифрованного файла всегда один и тот же, но контрольная сумма всякий раз разная. Хотя само RSA-шифрование детерминировано, дополнение OAEP таковым не является, в результате чего шифртекст получается недетерминированным.

Мы успешно зашифровали небольшой файл. А что, если попытаться зашифровать файл побольше? Попробуем. Сначала сгенерируем файл нужного размера:

```
$ seq 20000 >somefile.txt
```

Получился файл примерно на 100 КБ. Зашифруем его:

```
$ openssl pkeyutl \
    -encrypt \
    -in somefile.txt \
    -out somefile.txt.encrypted \
    -pubin \
    -inkey rsa_public_key.pem \
    -pkeyopt rsa_padding_mode:oaep
Public Key operation error
C011FAE8677F0000:error:0200006E:
rsa routines:ossl_rsa_padding_add_PKCS1_OAEP_mgf1_ex:
data too large for key size:crypto/rsa/rsa_oaep.c:87:
```

Мы получили ошибку, означающую, что входные данные слишком велики для ключа такого размера. Дело в том, что RSA не умеет шифровать данные длиннее ключа за одну операцию. А если используется дополнение, то максимально допустимая длина открытого текста еще меньше. При использовании дополнения типа OAEP максимальная длина входных данных равна длине ключа минус 42 (в нашем случае 4096 – 42 = 4054 байта). Если нужно зашиф-

ровать данные большей длины, то придется использовать симметричный сеансовый ключ.

Настало время расшифровать то, что мы зашифровали, и сравнить с оригиналом. Для расшифрования нужно передать `openssl pkeyutl` пару ключей, а не просто открытый ключ:

```
$ openssl pkeyutl \
    -decrypt \
    -in session_key.bin.encrypted \
    -out session_key.bin.decrypted \
    -inkey rsa_keypair.pem \
    -pkeyopt rsa_padding_mode:oaep
```

Проверим контрольную сумму:

```
$ cksum session_key.bin*
186975932 32 session_key.bin
186975932 32 session_key.bin.decrypted
2324953448 512 session_key.bin.encrypted
```

Как видим, размеры и контрольные суммы файлов `session_key.bin` и `session_key.bin.decrypted` одинаковы, т. е. расшифрованный файл совпадает с исходным. Замечательно, что расшифрование всегда детерминировано!

Шифрование с помощью RSA из программы

OpenSSL 3.0 предлагает следующие API для RSA-шифрования:

- унаследованный API с префиксом `RSA_` и функцией `RSA_public_encrypt()`. В версии OpenSSL 3.0 он объявлен нерекомендуемым, и мы его использовать не будем;
- API `EVP_PKEY`, конкретно функция `EVP_PKEY_encrypt()`. Именно им мы и воспользуемся;
- API `EVP_Seal`. Это API гибридного шифрования, который генерирует сеансовый ключ, зашифровывает его с помощью RSA, а затем шифрует пользовательские данные сеансовым ключом. Этот API включает функции `EVP_SealInit()`, `EVP_SealUpdate()` и `EVP_SealFinal()`, работающие по аналогии с `EVP_EncryptInit()`, `EVP_EncryptUpdate()` и `EVP_EncryptFinal()`. `EVP_SealUpdate()` – это просто макрос `#define` для `EVP_EncryptUpdate()`. Существуют соответственные функции `EVP_Open` для расшифрования *пломб*: `EVP_OpenInit()`, `EVP_OpenUpdate()` и `EVP_OpenFinal()`. К сожалению, API `EVP_Seal` не особенно гибкий. Он поддерживает только RSA-шифрование сеансового ключа. API не предоставляет метода, позволяющего задать дополнение или другие свойства RSA-шифрования: всегда используется дополнение PKCS #1 v1.5, а не OAEP. Довольно трудно понять, как следует вызывать функцию `EVP_SealInit()`; нужно несколько минут изучать документацию, пока не дойдет. Мне кажется, что API `EVP_Seal` плохо спроектирован и должен быть заменен. Поэтому я не рекомендую им пользоваться.

Мы напишем программу `rsa-encrypt`, которая будет шифровать небольшую порцию данных, например сеансовый ключ, с помощью RSA, подобно тому, как мы шифровали командой `openssl pkeyutl` в предыдущем разделе. Наша программа будет интероперабельна с `pkeyutl`, т. е. `pkeyutl` можно будет использовать для расшифрования порожденного нашей программой шифртекста.

Итак, мы будем использовать API `EVP_PKEY`. Ниже приведен перечень страниц руководства для интересующих нас функций:

```
$ man PEM_read_PUBKEY
$ man EVP_PKEY_CTX_new_from_pkey
$ man EVP_PKEY_encrypt_init
$ man EVP_PKEY_CTX_set_rsa_padding
$ man EVP_PKEY_get_size
$ man EVP_PKEY_encrypt
$ man provider-asym_cipher
```

Наша программа будет принимать три аргумента в командной строке:

1) имя входного файла;
2) имя выходного файла;
3) имя файла с открытым ключом RSA.

Составим общий план реализации, как обычно делаем. Программа должна выполнить следующие шаги:

1) загрузить открытый ключ RSA из файла;
2) создать контекст `EVP_PKEY` с этим ключом;
3) инициализировать контекст `EVP_PKEY` для шифрования и установить режим дополнения OAEP;
4) прочитать открытый текст из входного файла;
5) зашифровать открытый текст;
6) записать шифртекст в выходной файл.

Реализация программы rsa-encrypt

1. Сначала загрузим открытый ключ RSA:

```
const char* pkey_fname = argv[3];
FILE* pkey_file = fopen(pkey_fname, "rb");
EVP_PKEY* pkey = PEM_read_PUBKEY(
    pkey_file, NULL, NULL, NULL);
```

2. Затем создадим контекст `EVP_PKEY` с этим ключом:

```
EVP_PKEY_CTX* ctx = EVP_PKEY_CTX_new_from_pkey(
    NULL, pkey, NULL);
```

3. Далее инициализируем `EVP_PKEY_CTX` и установим режим дополнения:

```
EVP_PKEY_encrypt_init(ctx);
EVP_PKEY_CTX_set_rsa_padding(ctx, RSA_PKCS1_OAEP_PADDING);
```

Можно также инициализировать в новом стиле OpenSSL 3.0, воспользовавшись массивом параметров типа `OSSL_PARAM`:

```
int rsa_padding = RSA_PKCS1_OAEP_PADDING;
OSSL_PARAM params[] = {
    OSSL_PARAM_construct_int(
        OSSL_ASYM_CIPHER_PARAM_PAD_MODE, &rsa_padding),
    OSSL_PARAM_construct_end()
};
EVP_PKEY_encrypt_init_ex(ctx, params);
```

Интересно, что в документации по OpenSSL 3.0 (на странице руководства `provider-asym_cipher`) параметр `OSSL_ASYM_CIPHER_PARAM_PAD_MODE` определен как целое, а в исходном коде OpenSSL (в файле `rsa_enc.c`) тот же параметр определен как строка в кодировке UTF-8. Код принимает как целое, так и строку и подставляет целое значение `legacy pad mode number`. Если режим дополнения задается строкой, то код инициализации контекста будет выглядеть так:

```
OSSL_PARAM params[] = {
    OSSL_PARAM_construct_utf8_string(
        OSSL_ASYM_CIPHER_PARAM_PAD_MODE,
        OSSL_PKEY_RSA_PAD_MODE_OAEP,
        0),
    OSSL_PARAM_construct_end()
};
EVP_PKEY_encrypt_init_ex(ctx, params);
```

Я предпочитаю старый стиль инициализации за его удобочитаемость, краткость и типобезопасность, пусть даже он менее общий. Сама OpenSSL также предпочитает старый стиль. В исходном коде OpenSSL 3.0 есть несколько вызовов `EVP_PKEY_CTX_set_rsa_padding()`, но ни одного вызова, содержащего `OSSL_ASYM_CIPHER_PARAM_PAD_MODE`, если не считать кода реализации.

4. Следующий шаг – прочитать входные данные. Но сколько данных нужно читать? Мы знаем, что в режиме с дополнением количество байтов, шифруемых RSA за одну операцию, чуть меньше длины ключа. Можно рискнуть и зашифровать байтов столько, чему равна длина ключа. Если операция не пройдет из-за слишком длинного входа, то мы узнаем об этом по коду ошибки.
Определим длину ключа, выделим память для входного и выходного буферов и прочитаем входные данные:

```
size_t pkey_size = EVP_PKEY_get_size(pkey);
unsigned char* in_buf = malloc(pkey_size);
unsigned char* out_buf = malloc(pkey_size);
size_t in_nbytes = fread(in_buf, 1, pkey_size, in_file);
```

5. Асимметричное шифрование входных данных выполняет функция `EVP_PKEY_encrypt()`:

```
size_t out_nbytes = pkey_size;
EVP_PKEY_encrypt(
ctx, out_buf, &out_nbytes, in_buf, in_nbytes);
```

6. Зашифрованные данные нужно записать в выходной файл:

```
fwrite(out_buf, 1, out_nbytes, out_file);
```

7. Закончив работу, освободим буферы и объекты:

```
free(out_buf);
free(in_buf);
EVP_PKEY_CTX_free(ctx);
EVP_PKEY_free(pkey);
```

Полный исходный код программы `rsa-encrypt` находится на GitHub по адресу https://github.com/PacktPublishing/Demystifying-Cryptography-with-OpenSSL-3/blob/main/Chapter06/rsa-encrypt.c.

Выполнение программы rsa-encrypt

Выполним программу `rsa-encrypt` и зашифруем созданный ранее 32-байтовый сеансовый ключ:

```
$ ./rsa-encrypt \
    session_key.bin \
    session_key.bin.encrypted \
    rsa_public_key.pem
Encryption succeeded
```

Программа сообщила об успешном завершении, проверим файлы:

```
$ cksum session_key.bin*
186975932 32 session_key.bin
390548196 512 session_key.bin.encrypted
```

Пока все хорошо – программа создала файл `session_key.bin.encrypted` правильного размера. Я обещал, что программа будет интероперабельной с `openssl pkeyutl`. Проверим:

```
$ openssl pkeyutl \
    -decrypt \
    -in session_key.bin.encrypted \
    -out session_key.bin.decrypted \
    -inkey rsa_keypair.pem \
    -pkeyopt rsa_padding_mode:oaep
```

Ошибок нет. Проверим файлы и их контрольные суммы:

```
$ cksum session_key.bin*
186975932 32 session_key.bin
```

```
186975932 32 session_key.bin.decrypted
2324953448 512 session_key.bin.encrypted
```

Как видим, размеры и контрольные суммы файлов `session_key.bin` и `session_key.bin.decrypted` одинаковы, т. е. `openssl pkeyutl` успешно расшифровывает файл, зашифрованный программой `rsa-encrypt`.

Я упоминал об обработке ошибок, но пока не показал такого кода. Просто я не хотел, чтобы код обработки ошибок загромождал фрагменты в тексте книги, хотя в полном исходном коде на GitHub он присутствует. Обработка ошибок в OpenSSL – отдельная тема, заслуживающая целого раздела. Вот к ней мы сейчас и перейдем.

Что такое очередь ошибок в OpenSSL

При выполнении программы `rsa-encrypt` многое может пойти не так, например:

- файл с открытым ключом может быть поврежден или не содержать ключа. В таком случае загрузка ключа завершится ошибкой;
- файл с открытым ключом может содержать ключ, который не является ключом RSA. В таком случае загрузка ключа завершится успешно, но при попытке шифрования возникнет ошибка;
- входной файл может быть слишком велик. В таком случае шифрование тоже завершится ошибкой, но по другой причине.

Как обрабатывать такие ошибки? Те функции OpenSSL, которые могут завершаться ошибкой, обычно сообщают о ней, возвращая `NULL`, `0` или отрицательное число. В случае успешного завершения обычно возвращается 1. Некоторые функции также добавляют ошибку в очередь ошибок OpenSSL в случае неудачи.

Очередь OpenSSL – это контейнер для ошибок, о которых хочет сообщить библиотека. *У каждого потока в процессе имеется своя очередь ошибок.* Очередь не нуждается ни в инициализации, ни в очистке; OpenSSL берет это на себя. В начале работы каждого потока очередь пуста. Ошибки хранятся в очереди, пока не будут извлечены из нее или до момента завершения потока. Поэтому следует проверять очередь ошибок, обрабатывать ошибки и при необходимости очищать очередь после обращений к OpenSSL.

Не все функции помещают ошибки в очередь; например, я не находил там ошибок при неуспешном завершении симметричного шифрования или инициализации функции вычисления HMAC. Но те функции, которые отвечают за асимметричную криптографию, сертификаты X.509 или TLS, обычно помещают ошибки в очередь. Некоторые функции, особенно те, что проверяют сертификаты X.509, могут помещать сразу несколько ошибок в результате одного обращения.

Ниже перечислены популярные функции для взаимодействия с очередью ошибок OpenSSL:

- `ERR_get_error()`: получить код самой старой ошибки в очереди и удалить эту ошибку из очереди;
- `ERR_peek_error()`: получить код самой старой ошибки в очереди, но оставить эту ошибку в очереди;
- `ERR_GET_LIB()` и `ERR_GET_REASON()`: получить компоненты кода, возвращенного `ERR_get_error()` или `ERR_peek_error()`;
- `ERR_error_string_n()`: получить понятную человеку строку, соответствующую коду ошибки;
- `ERR_clear_error()`: очистить очередь, удалив из нее все ошибки;
- `ERR_print_errors_fp()`: вывести очередь ошибок в поток `FILE` и очистить очередь. Отметим, что хотя эта функция очень удобна для отладки, она не сообщает об ошибке вывода очереди, например если поток `FILE` связан с файловой системой, а диск заполнился. Функция `ERR_print_errors_fp()` не возвращает код ошибки и не помещает ошибку в очередь, потому что в ее задачу входит также очистка очереди. Ситуация, когда невозможно получить информацию об ошибке от функции обработки ошибок, заставляет меня иронично улыбнуться, хотя я и нахожу ее поведение вполне логичным.

Важно понимать, что коды, возвращаемые функциями OpenSSL, не то же самое, что коды ошибок, хранящиеся в составе ошибок в очереди; например, если на вход функции RSA-шифрования поданы слишком длинные данные, то функция `EVP_PKEY_encrypt()` возвращает код `0` и помещает в очередь ошибку с кодом `0x200006E`, которая состоит из следующих компонентов:

- номер библиотеки, возвращаемый функцией `ERR_GET_LIB()`: 4, т. е. `ERR_R_RSA_LIB`;
- код причины, возвращаемый функцией `ERR_GET_REASON()`: `0x6E`, или 110 – значение символической константы `RSA_R_DATA_TOO_LARGE_FOR_KEY_SIZE`.

В коде ниже показано, как получить самую старую ошибку из очереди и напечатать информацию о ней:

```
unsigned long error_code = ERR_get_error();
printf("Код ошибки: %lX\n", error_code);
printf("Номер библиотеки: %i\n", ERR_GET_LIB(error_code));
printf("Код причины: %X\n", ERR_GET_REASON(error_code));
```

Здесь номер библиотеки и код причины печатаются в виде чисел. Но где найти их символические представления, `ERR_R_RSA_LIB` и `RSA_R_DATA_TOO_LARGE_FOR_KEY_SIZE`? К сожалению, я не смог найти ответ в документации по OpenSSL, но он есть в заголовочных файлах библиотеки. Список номеров библиотек находится в заголовке `err.h`, а коды причин – в заголовках конкретных библиотек, например подбиблиотеке RSA соответствует заголовочный файл `rsaerr.h`, содержащий коды ошибок.

Функции `ERR_get_error()`, `ERR_peek_error()`, `ERR_GET_LIB()` и `ERR_GET_REASON()` дают удобный способ обработки ошибок OpenSSL. Но что, если нужно что-то простое и быстрое для отладки? Тогда можно воспользоваться функцией `ERR_print_errors_fp()`. Если написать

```
ERR_print_errors_fp(stderr);
```

то очередь ошибок будет выведена на stderr и очищена.

Вот как функция ERR_print_errors_fp() используется в исходном коде нашей программы rsa-encrypt:

```
if (ERR_peek_error()) {
    exit_code = 1;
    if (error_stream) {
        fprintf(
            error_stream,
            "Errors from the OpenSSL error queue:\n");
        ERR_print_errors_fp(error_stream);
    }
}
```

Попробуем запустить rsa-encrypt, задав слишком длинный вход:

```
$ seq 20000 >somefile.txt
$ ./rsa-encrypt \
    somefile.txt \
    somefile.txt.encrypted \
    rsa_public_key.pem
EVP_API error
Errors from the OpenSSL error queue:
806B3E2A797F0000:error:
    0200006E:rsa routines:
    ossl_rsa_padding_add_PKCS1_OAEP_mgf1_ex:
    data too large for key size:
    crypto/rsa/rsa_oaep.c:87:
Encryption failed
```

Мы получили такую же распечатку очереди ошибок OpenSSL, какую выдала команда openssl pkeyutl в аналогичной ситуации. Многовато информации для одной ошибки, но часть этой длинной строки вполне понятна:

```
data too large for key size
```

Дополнительные сведения об обработке ошибок в OpenSSL можно найти на следующих страницах руководства:

```
$ man ERR_get_error
$ man ERR_GET_LIB
$ man ERR_error_string_n
$ man ERR_print_errors_fp
$ man ERR_clear_error
```

Конечно, вам решать, как именно обрабатывать ошибки после вызовов OpenSSL. Но, будучи ответственным программистом, вы не должны забывать обрабатывать ошибки и затем очищать очередь ошибок.

Представьте, что вы выполнили несколько операций OpenSSL. В большинстве из них вам было достаточно кодов возврата, и вы забыли очистить оче-

редь ошибок, но для последней операции вам захотелось получить ошибку из очереди. Однако теперь очередь содержит ошибки всех ранее выполненных операций. Как найти среди них те, что соответствуют последней операции? Хорошего способа нет, поэтому последнюю операцию следовало начинать с пустой очередью. Здесь операция может состоять не из одного обращения к OpenSSL, а из нескольких. Главное, чтобы она представляла некоторый законченный логический кусок работы, для которого все ошибки имеет смысл обрабатывать сразу.

Когда лучше очищать очередь ошибок – до или после операции? На этот счет есть разные мнения. Одни считают, что очищать следует после операции, потому что ответственный программист должен подчищать за собой и не давать ошибкам *утекать*. Другие полагают, что очищать до операции лучше, потому что тогда вы гарантированно получите пустую очередь до начала операции. Лично я предпочитаю очищать очередь до и после – после, потому что это ответственное поведение, а до, потому что в сложных проектах участвует много людей, и кто-то из них мог забыть почистить за собой очередь. Людям свойственно ошибаться; это печальная правда жизни и разработки ПО.

Но довольно философии. Пришло время заняться расшифрованием с помощью RSA из программы на C.

Расшифрование с помощью RSA из программы

В этом разделе мы разработаем небольшую программу `rsa-decrypt`, чтобы узнать, как производится RSA-расшифрование.

Наша программа будет принимать почти такие же аргументы, как `rsa-encrypt`, только третьим будет имя файла, содержащего пару ключей, а не просто открытый ключ:

1) имя входного файла;
2) имя выходного файла;
3) имя файла с парой ключей.

Разумеется, теперь входной файл должен содержать шифртекст, который будет расшифрован, а получившийся открытый текст записан в выходной файл.

План реализации `rsa-decrypt` содержит список действий, противоположных тем, что мы выполняли в `rsa-encrypt`.

1. Загрузить пару ключей RSA из файла. Для расшифрования нам понадобится закрытый ключ, открытого недостаточно.
2. Создать контекст `EVP_PKEY` с этим ключом.
3. Инициализировать контекст `EVP_PKEY` для расшифрования и установить режим дополнения OAEP.
4. Прочитать шифртекст из входного файла.

5. Расшифровать шифртекст.
6. Записать открытый текст в выходной файл.

Реализация программы rsa-decrypt

1. Первым делом загружаем пару ключей RSA:

   ```
   const char* pkey_fname = argv[3];
   FILE* pkey_file = fopen(pkey_fname, "rb");
   EVP_PKEY* pkey = PEM_read_PrivateKey(
       pkey_file, NULL, NULL, NULL);
   ```

   Отметим, что и для открытого ключа, и для пары ключей используется один и тот же тип, `EVP_PKEY`.
2. Далее создаем контекст `EVP_PKEY` из загруженной пары ключей:

   ```
   EVP_PKEY_CTX* ctx = EVP_PKEY_CTX_new_from_pkey(
       NULL, pkey, NULL);
   ```

3. Затем инициализируем `EVP_PKEY_CTX` и устанавливаем режим дополнения:

   ```
   EVP_PKEY_decrypt_init(ctx);
   EVP_PKEY_CTX_set_rsa_padding(ctx, RSA_PKCS1_OAEP_PADDING);
   ```

4. Определяем длину ключа, выделяем память для входного и выходного буферов и читаем входные данные:

   ```
   size_t pkey_size = EVP_PKEY_get_size(pkey);
   unsigned char* in_buf = malloc(pkey_size);
   unsigned char* out_buf = malloc(pkey_size);
   size_t in_nbytes = fread(in_buf, 1, pkey_size, in_file);
   ```

5. Расшифровываем полученный шифртекст функцией `EVP_PKEY_decrypt()`:

   ```
   size_t out_nbytes = pkey_size;
   EVP_PKEY_decrypt(
       ctx, out_buf, &out_nbytes, in_buf, in_nbytes);
   ```

6. Записываем расшифрованные данные в выходной файл:

   ```
   fwrite(out_buf, 1, out_nbytes, out_file);
   ```

7. Закончив работу, освобождаем буферы и объекты:

   ```
   free(out_buf);
   free(in_buf);
   EVP_PKEY_CTX_free(ctx);
   EVP_PKEY_free(pkey);
   ```

Полный исходный код программы `rsa-encrypt` находится на GitHub по адресу https://github.com/PacktPublishing/Demystifying-Cryptography-with-OpenSSL-3/blob/main/Chapter06/rsa-decrypt.c.

Выполнение программы rsa-decrypt

Выполним программу `rsa-decrypt` и расшифруем созданный ранее файл с зашифрованным сеансовым ключом:

```
$ ./rsa-decrypt \
session_key.bin.encrypted \
session_key.bin.decrypted \
rsa_keypair.pem
Decryption succeeded
```

Программа сообщила об успешном завершении, проверим файлы:

```
$ cksum session_key.bin*
186975932 32 session_key.bin
186975932 32 session_key.bin.decrypted
2324953448 512 session_key.bin.encrypted
```

Как видим, размеры и контрольные суммы файлов `session_key.bin` и `session_key.bin.decrypted` одинаковы, т. е. наша программа `rsa-decrypt` успешно расшифровала файл с сеансовым ключом.

Резюме

В этой главе мы узнали о том, что такое асимметричная криптография. Затем мы поговорили об атаке с человеком посередине и о том, как ей противостоять. Далее мы узнали, что такое сеансовый ключ. Теоретическую часть мы закончили рассказом о криптостойкости RSA, компромиссе между стойкостью и производительностью и о том, как выбирать размер ключа RSA.

В практической части мы узнали о RSA-шифровании и расшифровании в командной строке, а также из программы на C. Мы рассказали об очереди ошибок OpenSSL и о проверке ошибок.

В следующей главе продолжим тему асимметричной криптографии и поговорим о цифровых подписях и их проверке.

Глава 7

Цифровые подписи и их проверка

В этой главе мы узнаем о **цифровых подписях**, о том, как подписать сообщение и затем проверить подпись. У цифровых подписей много важных практических применений, например подписание документов, сертификатов, программ, сетевых запросов и ответов, включая финансовые транзакции. Цифровые подписи важны также для безопасности криптовалют – биткойна и других. Алгоритмы цифровой подписи – необходимый элемент проверки подлинности клиента и сервера в безопасных сетевых протоколах, таких как *TLS*, *SSH* и *IPsec*. Подписи используются в процедуре квитирования *TLS* и для подписания **сертификатов X.509**, которые нужны для идентификации и верификации в *TLS*. Дополнительные сведения о сертификатах X.509 см. в главе 8. Протокол TLS рассматривается в части IV. Цифровые подписи также находят применение в стандартах безопасного обмена сообщениями, в частности **Pretty Good Privacy** (**PGP**) и **Secure Multipurpose Mail Extensions** (**S/MIME**). Например, все выпуски OpenSSL подписаны цифровыми подписями *PGP*.

Мы дадим обзор алгоритмов цифровых подписей, поддерживаемых OpenSSL, и порекомендуем, какие схемы использовать, а каких лучше избегать. В практической части главы мы покажем, как подписать содержимое и проверить подпись в командной строке и из программы на C.

Будут рассмотрены следующие темы:

- что такое цифровая подпись;
- обзор алгоритмов цифровых подписей, поддерживаемых OpenSSL;
- криптостойкость RSA;
- генерирование пары ключей эллиптического шифрования;
- подписание и проверка подписи в командной строке;
- подписание из программы;
- проверка подписи из программы.

Технические требования

В этой главе встречаются команды, запускаемые из командной строки, и C-код, который можно откомпилировать и выполнить. Для запуска команд понадобятся программа `openssl` и соответствующие динамические библиотеки OpenSSL. Для сборки кода на C будут нужны динамические или статические библиотеки OpenSSL, заголовки библиотек, компилятор C и компоновщик.

Мы реализуем демонстрационные программы, чтобы применить полученные знания на практике. Их полный исходный код находится по адресу https://github.com/PacktPublishing/Demystifying-Cryptography-with-OpenSSL-3/tree/main/Chapter07.

Что такое цифровая подпись

Цифровая подпись – это массив битов, дающих криптографически строгие гарантии аутентичности, целостности и неотрицаемости цифрового сообщения. Объясним, что это такое.

- **Аутентичность** означает, что сообщение отправлено именно объявленным отправителем, при условии что этот отправитель владеет закрытым ключом, использованным для порождения подписи.
- **Целостность** означает, что сообщение не было изменено третьей стороной, например в процессе передачи.
- **Неотрицаемость** означает, что отправитель не может отказаться от подписи, при условии что никто, кроме него, не имеет доступа к закрытому ключу, использованному для ее порождения.

Цифровая подпись порождается *закрытым ключом*, который можно проверить с помощью соответствующего ему *открытого ключа*. Поэтому алгоритмы цифровой подписи и ее проверки считаются **алгоритмами асимметричной криптографии**, хотя они и не являются **алгоритмами асимметричного шифрования**.

Важно понимать, что обычно при подписании сообщения цифровая подпись применяется не к самому сообщению, а к его *хеш-значению*, созданному какой-то криптографической функцией хеширования, например SHA-256. Криптографы говорят, что такие схемы основаны на парадигме **хеширования и подписания**.

Заметным, хотя и спорным исключением является алгоритм подписания EdDSA. Заметный он потому, что это единственный популярный алгоритм, который принимает на входе сообщение целиком, а не его хеш-значение. А спорный, потому что внутри сообщение все равно хешируется, пусть и несколько более замысловато, чем в других алгоритмах цифровой подписи.

Так почему же подписывается хеш сообщения, а не оно само? По тем же причинам, по которым в асимметричном шифровании используется сеан-

совый ключ. Асимметричные алгоритмы сравнительно медленные – гораздо медленнее, чем криптографические функции хеширования. Быстрый алгоритм хеширования преобразует потенциально очень длинное входное сообщение в короткое хеш-значение известной длины, поэтому медленному алгоритму цифровой подписи не придется обрабатывать длинный вход. Такое разделение обязанностей повышает производительность. Другая причина – то, что алгоритм цифровой подписи может подписать только блок данных ограниченного размера. Неудобно было бы разбивать сообщение на несколько блоков, подписывать каждый по отдельности и изобретать безопасный метод зацепления подписей, стойкий к различным атакам с изменением порядка.

Обратите внимание, как *эффективная и производительная* асимметричная криптография строится поверх симметричной криптографии. В асимметричном шифровании используются симметричные сеансовые ключи, а в цифровых подписях – хеши сообщений. Поэтому так важно изучить *симметричную* криптографию, прежде чем приступать к асимметричной.

Различие между цифровыми подписями и имитовставками

Как видите, между цифровыми подписями и имитовставками (**MAC**) есть что-то общее. Те и другие обеспечивают аутентификацию и целостность. Однако есть и важные различия.

- Цифровые подписи создаются с помощью асимметричных закрытых ключей и проверяются с помощью открытых ключей. Имитовставки пользуются одним и тем же симметричным ключом как для создания аутентификационного жетона, так и для его проверки.
- Цифровую подпись можно проверить, не требуя от подписавшего предъявить свой секретный закрытый ключ. Любая сторона, знающая открытый ключ, может проверить подпись. В случае MAC один и тот же секретный ключ должен быть известен подписавшему и проверяющему. Только проверяющий может удостоверить, что сообщение пришло от объявленного отправителя и только если проверяющий не аутентифицировал то же самое сообщение. Третья сторона не может проверить сообщение. Если проверяющих несколько, то верифицировать источник сообщения становится еще труднее, не только из-за объявленного отправителя, но и потому, что любой проверяющий, обладающий секретным ключом, мог бы аутентифицировать сообщение.
- В отличие от цифровых подписей, имитовставки не обеспечивают неотрицаемости. Для любого сообщения отправитель может заявить, что аутентифицировал его не он, а проверяющий.

Теперь, когда мы понимаем, что такое цифровая подпись, посмотрим, какие алгоритмы цифровой подписи поддерживаются OpenSSL.

Обзор алгоритмов цифровых подписей, поддерживаемых OpenSSL

В этом разделе мы дадим обзор алгоритмов цифровых подписей, поддерживаемых OpenSSL, и порекомендуем, какие из них использовать.

Алгоритм RSA

Алгоритм **Ривеста–Шамира–Адлемана** (**RSA**) изобретен Рональдом Ривестом, Ади Шамиром и Леонардом Адлеманом. Рональд Ривест – тот самый человек, который придумал алгоритмы симметричного шифрования из семейства **RC** и алгоритмы вычисления хеш-значений из семейства **MD**. Алгоритм RSA был впервые опубликован в 1977 году. Это был первый популярный алгоритм асимметричной криптографии.

RSA – универсальный алгоритм, потому что позволяет как шифровать, так и подписывать сообщения. Это самый медленный алгоритм подписания, но самый быстрый алгоритм проверки подписи из всех алгоритмов с такой же криптостойкостью, предлагаемых OpenSSL.

Скорость подписания и проверки зависит от длины ключа. Скорость подписания быстро уменьшается с ростом длины ключа. Скорость проверки тоже меняется, но не так быстро. Проверка занимает в 30–70 раз меньше времени, чем подписание, в зависимости от длины ключа.

RSA порождает самые длинные подписи. Длина RSA-подписи совпадает с длиной ключа, использованного для ее создания. Например, подпись, созданная с помощью 4096-битового ключа, будет иметь длину 4096 бит, или 512 байт. Другие алгоритмы порождают гораздо более короткие подписи.

Еще один недостаток RSA заключается в том, что его ключи гораздо длиннее ключей более современных алгоритмов на основе **эллиптических кривых** (**EC**). Кроме того, чтобы получить несколько лишних битов стойкости, длину ключа RSA придется увеличить значительно.

RSA – старый алгоритм. OpenSSL поддерживает еще один старый алгоритм, DSA.

Алгоритм DSA

Алгоритм цифровой подписи (Digital Signature Algorithm – **DSA**) изобрел Дэвид У. Кравиц, когда работал в **Агентстве национальной безопасности** (**АНБ**) США. Впервые DSA был опубликован в 1991 году, когда NIST предложил использовать DSA в своем **стандарте цифровой подписи** (Digital Signature Standard – **DSS**).

Стойкость DSA опирается на вычислительную трудность проблемы дискретного логарифмирования. Как и в RSA, в DSA используются длинные ключи, например 2048-битовые. Как и для RSA, стойкость ключа DSA гораздо

меньше его длины. Так получилось, что стойкость ключей DSA и RSA одинаковой длины совпадает. Поэтому таблица соответствия между длиной ключа и его стойкостью, приведенная в главе 6, относится и к DSA. Так, 2048-битовый ключ DSA будет иметь стойкость 112 бит.

Первый стандарт DSS, одобренный под названием FIPS-186 в 1994 году, позволял использовать DSA только в сочетании с функцией хеширования SHA-1 и с длиной ключа не более 1024 бит. Более поздний DSS, FIPS 186-4 от 2013 года, допускает использование DSA совместно с функциями хеширования SHA-224 и SHA-256 и с длиной ключа до 3072 бит. OpenSSL поддерживает DSA со всеми тремя упомянутыми функциями хеширования и очень длинными ключами. Но на мой взгляд, использовать ключ DSA длиной более 4096 бит не стоит.

Хотя ключи DSA, как и ключи RSA, длинные, порождаемые ими подписи гораздо короче, чем длина ключа. В битах длина подписи DSA составляет *приблизительно* `4*K_bits`, где `K_bits` – стойкость использованного ключа DSA. Например, 4096-битовый ключ RSA порождает 512-байтовую подпись, а ключ DSA той же длины – 70- или 71-байтовую подпись.

В настоящее время алгоритм DSA не особенно популярен, его вытесняет ECDSA.

Алгоритм ECDSA

Алгоритм цифровой подписи на эллиптических кривых (Elliptic Curve Digital Signature Algorithm – **ECDSA**) – вариант алгоритма DSA, в котором используется **криптография на эллиптических кривых** (Elliptic Curve Cryptography – **ECC**). ECC – форма криптографии с открытым ключом, основанная на алгебраической структуре эллиптических кривых над конечным полем. ECC была предложена Нилом Коблицем и Виктором Миллером в 1985 году. ECDSA был предложен Скоттом Ванстоуном в 1992 году в виде комментария к первому стандарту DSS.

Преимущество ECDSA перед DSA заключается в том, что для достижения той же стойкости нужен ключ меньшей длины. Стойкость эллиптического ключа равна примерно половине его длины. Например, у 224-битового ключа стойкость 112 бит. Для сравнения ключ DSA такой же стойкости должен насчитывать 2048 бит. Как и в случае DSA, длина ECDSA-подписи в битах равна *приблизительно* `4*K_bits`, где `K_bits` – длина ключа ECDSA.

Для генерирования ключа ECDSA нужно выбрать кривую. У кривой есть имя, определяющее длину эллиптического ключа, сгенерированного на основе этой кривой. Так, ключ, сгенерированный на основе кривой NIST P-256, будет иметь длину 256 бит. OpenSSL поддерживает кривые NIST и кривые Brainpool. Кривые NIST разработаны АНБ и стандартизованы NIST. Кривые Brainpool предложены рабочей группой Brainpool, созданной криптографами, недовольными кривыми NIST ввиду отсутствия проверяемой гарантии их случайного генерирования, из-за чего возможно намеренное или случайное ослабление криптостойкости. Кривые Brainpool стандартизованы Федеральным управлением информационной безопасности Германии (Bundesamt

für Sicherheit in der Informationstechnik, BSI). Рабочая группа Brainpool определила метод проверяемой гарантии случайного генерирования эллиптических кривых и заявляет, что сгенерировала кривые Brainpool именно этим методом. Однако исследовательская группа под руководством Дэниэла Дж. Бернштейна попыталась верифицировать метод рабочей группы Brainpool и не смогла сгенерировать те же самые кривые. Впоследствии группа Бернштейна предложила свои эллиптические кривые и свой алгоритм цифровой подписи под названием EdDSA. Дополнительные сведения о кривых EdDSA будут приведены в следующем разделе.

Несмотря на все сомнения, кривые NIST остаются наиболее популярными кривыми для ECDSA. В набор NIST включены две очень быстрые кривые: P-256 и P-224. Кривая P-256 в 16 раз быстрее кривой Brainpool P256r1 при подписании и в 5 раз быстрее при проверке подписи. Кривая P-224, сравнимая по стойкости с 2048-битовым DSA, приблизительно в пять раз быстрее его при подписании и в два раза быстрее при проверке. Другие кривые NIST по скорости сопоставимы с кривыми Brainpool аналогичной стойкости, т. е. иногда быстрее оказываются кривые NIST, а иногда кривые Brainpool.

Важно отметить, что стандартная реализация ECDSA нуждается в хорошем **генераторе случайных чисел** (**ГСЧ**) как для генерирования ключей, так и для процедуры подписания. Существует также альтернативная реализация ECDSA, которая не генерирует случайные данные при подписании, а вместо них использует хеш закрытого ключа. Применение дефектного ГСЧ при подписании может привести к утечке закрытого ключа через подпись. При реализации ECDSA необходима сугубая осторожность, иначе возможна уязвимость для атаки с хронометражем, также способной раскрыть закрытый ключ. Когда-то в OpenSSL была реализация, уязвимая для такой атаки, но эта ошибка была исправлена в 2011 году.

ECDSA – очень популярный алгоритм, но у него есть конкурент: еще один алгоритм подписи на эллиптических кривых, EdDSA.

Алгоритм EdDSA

Алгоритм цифровой подписи на кривой Эвардса (Edwards-curve Digital Signature Algorithm – **EdDSA**) – еще один алгоритм подписи с применением эллиптических кривых. Он был опубликован в 2011 году группой криптографов под руководством Дэниэла Дж. Бернштейна, того самого, который разработал семейство симметричных потоковых шифров ChaCha и алгоритм вычисления имитовставки Poly1305.

EdDSA основан на **скрученных кривых Эдвардса**, утверждается, что он быстрее и его проще реализовать безопасно, чем ECDSA. В отличие от ECDSA, EdDSA не нуждается в ГСЧ на этапе подписания, и потому в нем невозможна утечка закрытых ключей из-за дефектного ГСЧ.

EdDSA поддерживает две кривые: 253-битовую Curve25519 и 456-битовую Curve448. При их проектировании во главу угла ставилась безопасность, поэтому трудно сделать ошибки, ослабляющие криптостойкость. Кроме того, они неуязвимы для атак с хронометражем.

Как и в случае ECDSA, стойкость EdDSA-подписей равна половине длины ключа, т. е. 126,5 бита для Curve25519 и 228 бит для Curve448. Длина подписи для Curve25519 равна 64 байта, а для Curve448 – 114 байт.

Хотя при проектировании кривых Curve25519 и Curve448 ставилась цель добиться высокой скорости, они оказались не быстрее NIST P-256, которая приблизительно в два раза быстрее Curve25519 при подписании и проверке подписи. У кривой P-224 примерно такая же скорость, как у Curve25519. Однако Curve25519 в 8–13 раз быстрее других кривых NIST и Brainpool сопоставимой стойкости при подписании и в 2–9 раз быстрее их при проверке. Кривая Curve448 приблизительно в два раза быстрее кривых NIST и Brainpool сопоставимой стойкости при подписании и в 1,5–3 раза быстрее при проверке.

EdDSA – необычный алгоритм цифровой подписи. Обычные алгоритмы принимают на входе криптографический хеш входных данных, т. е. данных, подлежащих подписанию. А EdDSA принимает входные данные целиком, но позволяет пользователю задать так называемую функцию предварительного хеширования, которая будет применена к данным до дальнейшей обработки.

В роли функции предварительного хеширования может выступать настоящая криптографическая хеш-функция, например SHA-256. Такой вариант EdDSA называется **HashEdDSA**. Он больше других похож на традиционные алгоритмы подписания, потому что передает хеш входных данных на следующую стадию обработки. Функция предварительного хеширования может быть и тождественной, т. е. возвращающей свой вход без изменения. Такой вариант EdDSA называется **PureEdDSA**.

Стоит отметить, что на внутреннем уровне алгоритма EdDSA используются функция хеширования SHA-512 для кривой Curve25519 и функция SHAKE256 для кривой Curve448. Поэтому в случае HashEdDSA входные данные хешируются дважды. Так, может быть, тогда стоит всегда использовать PureEdDSA? Нет, не все так просто – читайте дальше.

Еще одно различие между EdDSA и традиционными алгоритмами цифровой подписи заключается в том, что EdDSA – двухпроходный алгоритм, т. е. обрабатывает входные данные дважды. Это, в свою очередь, означает, что PureEdDSA не поддерживает потоковых входных данных, в отличие от традиционных алгоритмов. Если входные данные длинные и поступают из медленного хранилища или по дорогому сетевому подключению, то PureEdDSA – не лучший выбор.

OpenSSL 3.0 также поддерживает вариант PureEdDSA и требует, чтобы все данные были помещены в память для хеширования. Это может оказаться затруднительным, если входные данные очень длинные, а память ограничена. С другой стороны, хотя HashEdDSA еще не поддерживается OpenSSL непосредственно, программист может хешировать данные самостоятельно, а затем использовать PureEdDSA.

Алгоритм SM2

Shang Mi 2 (**SM2**) – еще один алгоритм цифровой подписи на основе эллиптических кривых. Это национальный стандарт КНР. Предложенный Сяоюн

Ваном с коллегами, он был стандартизован Китайским управлением коммерческой криптографии в 2012 году.

SM2 поддерживает только одну кривую: 256-битовую CurveSM2. Ее скорость при подписании и проверке подписи примерно такая же, как для 256-битовых кривых Brainpool. Длина подписи равна 71 байт.

Алгоритм SM2 обычно используется в сочетании с 256-битовой функцией хеширования SM3. OpenSSL поддерживает только это сочетание в своих высокоуровневых API.

Как видим, OpenSSL поддерживает несколько алгоритмов цифровой подписи. Какой же из них выбрать? Об этом мы поговорим в следующем разделе.

Какой алгоритм цифровой подписи выбрать?

В последнее время популярность обрели алгоритм EdDSA и кривая Curve25519. Curve25519 доверяют многие специалисты по безопасности. Если подписываемые данные всегда помещаются в память и не требуется совместимость со старыми программами, выбирайте EdDSA. Отметим, что на момент написания книги основные **удостоверяющие центры** еще не выпускали сертификатов X.509 на основе EdDSA. Поэтому если необходим TLS-сертификат для сервера в интернете, то пока придется удовлетвориться подписанным каким-то другим алгоритмом.

Если вам нужен более традиционный алгоритм подписания или требуется поддержать потоковое поступление данных, то выбирайте ECDSA, тоже очень популярный.

Если требуется интероперабельность со старыми программами или очень быстрая проверка подписи, то выбирайте RSA.

На этом теоретическая часть главы завершается. Далее мы научимся генерировать пару ключей EC и подписывать данные в командной строке.

Генерирование пары ключей эллиптического шифрования

Я хочу продемонстрировать традиционный подход к цифровым подписям, когда подписывается хеш-значение входных данных. Поэтому мы будем использовать в примерах алгоритм ECDSA, а не EdDSA.

Как вы уже знаете, пару ключей можно сгенерировать подкомандой `openssl genpkey`. Мы сгенерируем пару ключей EC максимальной длины, доступной в OpenSSL, 570 бит, основанную на кривой NIST B-571:

```
$ openssl genpkey \
    -algorithm EC \
    -pkeyopt ec_paramgen_curve:secp521r1 \
    -out ec_keypair.pem
```

Здесь мы воспользовались параметром `-pkeyopt ec_paramgen_curve:secp521r1`, чтобы задать кривую NIST B-571. Какие еще имена можно было бы указать вместо `secp521r1`? Полный список поддерживаемых кривых можно получить от команды

```
$ openssl ecparam -list_curves
```

После того как пара ключей сгенерирована и записана в файл `ec_keypair.pem`, мы можем взглянуть на ее структуру:

```
$ openssl pkey -in ec_keypair.pem -noout -text
Private-Key: (570 bit)
priv:
  ...шестнадцатеричные значения ...
pub:
  ...шестнадцатеричные значения ...
ASN1 OID: sect571r1
NIST CURVE: B-571
```

Как видим, пара ключей EC по структуре проще, чем пара ключей RSA. Это просто два длинных значения – закрытая и открытая части.

Извлечь открытый ключ из пары можно так же, как мы делали для RSA в главе 6:

```
$ openssl pkey \
    -in ec_keypair.pem \
    -pubout \
    -out ec_public_key.pem
```

Исследуем структуру извлеченного открытого ключа:

```
$ openssl pkey -in ec_public_key.pem -pubin -noout -text
Public-Key: (570 bit)
pub:
  ...шестнадцатеричные значения ...
ASN1 OID: sect571r1
NIST CURVE: B-571
```

Здесь открытый ключ содержит то же самое значение `pub`, что и в паре ключей, и, естественно, не содержит части `priv`.

Для *подписания* нам необходим *закрытый ключ*, а для *проверки* подписи – открытый ключ. Напомним, что никогда не следует никому сообщать ни закрытый ключ, ни пару ключей. Распространять можно только свой открытый ключ.

В главе 6 мы объясняли, что в документации по OpenSSL под закрытым ключом обычно понимается пара ключей. Кроме того, не существует подкоманды для извлечения закрытой части из пары ключей. Это можно сделать только программно, если знать, как ключ эллиптической криптографии строится в коде на C, и выделить нужные данные. На практике выделять закрытый ключ из пары не имеет смысла, потому что когда OpenSSL нужен

закрытый ключ, она может использовать пару. Если вам требуется подписать что-нибудь с помощью OpenSSL, в командной строке или в программе, просто передайте пару ключей и не заморачивайтесь выделением закрытого ключа.

Итак, мы сгенерировали пару эллиптических ключей. Давайте теперь что-нибудь подпишем.

Подписание и проверка подписи в командной строке

OpenSSL предоставляет несколько подкоманд для подписания и проверки подписи:

- нерекомендуемая подкоманда `openssl rsautl`, работающая только с RSA;
- подкоманда `openssl dgst`: обычно она используется для вычисления хеша сообщения, но умеет также подписывать созданные хеши. Это значит, что ее нельзя использовать для подписания в алгоритме PureEdDSA, потому что он не подписывает хеши;
- подкоманда `openssl pkeyutl`: эту подкоманду можно использовать в сочетании с любым алгоритмом подписания, поддерживаемым OpenSSL. До выхода версии OpenSSL 3.0 подкоманда `openssl pkeyutl` не поддерживала подписания длинных входных данных; пользователь должен был создать хеш сообщения до подписания. Начиная с версии OpenSSL 3.0 `openssl pkeyutl` поддерживает как «необработанный вход» (raw input), так и хеш сообщения в качестве входных данных.

В примерах ниже мы будем пользоваться подкомандой `openssl pkeyutl`. Она документирована на странице руководства `openssl-pkeyutl`:

```
man openssl-pkeyutl
```

Для проверки подписи в командной строке выполните следующие действия.

1. Сгенерировать файл с тестовыми данными для подписания:

   ```
   $ seq 20000 >somefile.txt
   ```

2. Подписать файл, используя функцию хеширования SHA3-512 и ранее сгенерированную пару ключей EC:

   ```
   $ openssl pkeyutl \
       -sign \
       -digest sha3-512 \
       -inkey ec_keypair.pem \
       -in somefile.txt \
       -rawin \
       -out somefile.txt.signature
   ```

3. Проверить размеры и контрольные суммы файлов:

```
$ cksum somefile.txt*
3231941463 108894 somefile.txt
2915754802 151 somefile.txt.signature
```

   Мы видим, что в результате операции был создан файл с подписью `somefile.txt.signature` длиной 151 байт.
4. Теперь, если отдать кому-то исходный файл `somefile.txt`, его подпись `somefile.txt.signature` и наш открытый ключ `ec_public_key.pem`, то этот кто-то сможет проверить подпись. Вот как это делается в командной строке:

```
$ openssl pkeyutl \
    -verify \
    -digest sha3-512 \
    -inkey ec_keypair.pem \
    -in somefile.txt \
    -rawin \
    -sigfile somefile.txt.signature
Signature Verified Successfully
```

   Обратите внимание, что для подписания мы указали флаг `-sign`, а для проверки – флаг `-verify`.
5. Что, если подписанные данные были изменены в процессе передачи? Смоделируем такую ситуацию, для чего допишем какие-нибудь данные в конец подписанного файла и попытаемся еще раз проверить подпись:

```
$ echo "additional data" >> somefile.txt
$ openssl pkeyutl \
    -verify \
    -digest sha3-512 \
    -inkey ec_keypair.pem \
    -in somefile.txt \
    -rawin \
    -sigfile somefile.txt.signature
Signature Verification Failure
```

   Как видим, на этот раз проверка не прошла – если данные были изменены, процедура проверки подписи обнаруживает это и сообщает об ошибке.

Подписание из программы

OpenSSL 3.0 предоставляет следующие API для цифровой подписи:

- унаследованные низкоуровневые API, содержащие функции с префиксами, зависящими от алгоритма, например `RSA_`, `DSA_` и `ECDSA_`. Эти API объявлены нерекомендуемыми начиная с версии OpenSSL 3.0;

- API EVP_PKEY: этот API нерекомендуемым не объявлен, но все равно является низкоуровневым. Удобнее использовать высокоуровневый API;
- API EVP_Sign: этот API высокоуровневый, но обладает рядом недостатков. Заданный в аргументе ключ используется только после того, как входные данные были полностью прочитаны и хешированы. Поэтому если ключ был некорректным, то обнаружится это скорее поздно, чем рано. Другой недостаток – то, что этот API негибкий и не позволяет задавать параметры подписания в контексте PKEY_CTX, даже если алгоритм их поддерживает. В документации по OpenSSL этот API использовать не рекомендуется;
- API EVP_DigestSign: это высокоуровневый API, в котором исправлены недостатки API EVP_Sign. Именно он рекомендуется в документации, поэтому им мы и воспользуемся.

Мы напишем небольшую программу ec-sign, в которой будет использоваться алгоритм ECDSA, как в подкоманде openssl pkeyutl в предыдущем разделе. Наша программа будет интероперабельна с pkeyutl, т. е. pkeyutl сможет проверить подпись, созданную нашей программой.

Ниже перечислено несколько страниц руководства, имеющих отношение к используемому нами API:

```
$ man PEM_read_PrivateKey
$ man PEM_read_PUBKEY
$ man EVP_DigestSignInit_ex
$ man EVP_DigestVerifyInit_ex
```

Наша программа будет принимать три аргумента:

1) имя входного файла, содержащего подлежащие подписанию данные;
2) имя выходного файла, в который будет записана подпись;
3) имя файла, содержащего пару ключей для подписания.

Общий план выглядит следующим образом.

1. Загрузить пару ключей из файла.
2. Создать объект типа EVP_MD_CTX, который будет служить контекстом подписания.
3. Инициализировать контекст подписания, указав функцию хеширования и загруженную пару ключей.
4. Читать подписываемые данные порциями и помещать их в контекст подписания.
5. Определить длину подписи и выделить для нее память.
6. Завершить процедуру подписания и получить подпись.
7. Записать подпись в выходной файл.

Реализация программы ec-sign

1. Первым делом загружаем пару ключей:

```
const char* pkey_fname = argv[3];
FILE* pkey_file = fopen(pkey_fname, “rb”);
```

```
EVP_PKEY* pkey = PEM_read_PrivateKey(
    pkey_file, NULL, NULL, NULL);
```

2. Создаем контекст подписания `EVP_MD_CTX` и инициализируем его функцией хеширования SHA3-512 и загруженной парой ключей:

```
EVP_MD_CTX* md_ctx = EVP_MD_CTX_new();
EVP_DigestSignInit_ex(
        md_ctx,
        NULL,
        OSSL_DIGEST_NAME_SHA3_512,
        NULL,
        NULL,
        pkey,
        NULL);
```

3. Контекст инициализации создан, поместим в него входные данные:

```
const char* in_fname = argv[1];
FILE* in_file = fopen(in_fname, "rb");
while (!feof(in_file)) {
    size_t in_nbytes = fread(in_buf, 1, BUF_SIZE, in_file);
    EVP_DigestSignUpdate(md_ctx, in_buf, in_nbytes);
}
```

После того как все данные помещены в контекст, нужно завершить процедуру подписания и получить подпись. Но мы не знаем, сколько памяти выделить под буфер подписи.

4. К счастью, функция `EVP_DigestSignFinal()` сообщает длину подписи, если передать `NULL` вместо указателя на буфер:

```
size_t sig_len = 0;
EVP_DigestSignFinal(md_ctx, NULL, &sig_len);
```

Заметим, что `EVP_DigestSignFinal()` – не единственная в OpenSSL функция, которая таким образом возвращает длину выходного буфера. Так поступают и многие другие функции, если в качестве указателя на буфер передать `NULL`.

5. Зная длину подписи, мы можем выделить для нее буфер:

```
unsigned char* sig_buf = malloc(sig_len);
```

6. Теперь можно записать подпись в буфер и, наконец, получить ее:

```
EVP_DigestSignFinal(md_ctx, sig_buf, &sig_len);
```

Заметим, что на этот раз мы передали не `NULL`, а реальный указатель на буфер `sig_buf`.

Вместо того чтобы использовать функции `EVP_DigestSignInit_ex()`, `EVP_DigestSignUpdate()` и `EVP_DigestSignFinal()`, можно было бы вызвать одну функцию `EVP_DigestSign()`, но только при условии, что все подлежащие подписанию данные помещаются в непрерывный буфер. Для ECDSA

это необязательно, но для создания подписи алгоритмом PureEdDSA другого способа в OpenSSL 3.0 нет.

7. Получив подпись, запишем ее в выходной файл:

```
const char* sig_fname = argv[2];
sig_file = fopen(sig_fname, "wb");
fwrite(sig_buf, 1, sig_len, sig_file);
```

8. Наконец, освободим буферы, чтобы избежать утечек памяти:

```
free(sig_buf);
free(in_buf);
EVP_MD_CTX_free(md_ctx);
EVP_PKEY_free(pkey);
```

Полный исходный код программы `ec-sign` находится на GitHub по адресу https://github.com/PacktPublishing/Demystifying-Cryptography-with-OpenSSL-3/blob/main/Chapter07/ec-sign.c.

Выполнение программы ec-sign

Запустим программу `ec-sign` и посмотрим, как она работает:

```
$ ./ec-sign \
    somefile.txt \
    somefile.txt.signature \
    ec_keypair.pem
Signing succeeded
```

Программа сообщила об успешном завершении. Воспользуемся подкомандой `openssl pkeyutl` для проверки того, что подпись правильная:

```
$ openssl pkeyutl \
    -verify \
    -digest sha3-512 \
    -inkey ec_keypair.pem \
    -in somefile.txt \
    -rawin \
    -sigfile somefile.txt.signature
Signature Verified Successfully
```

Замечательно, что наша программа создала правильную подпись. Но еще лучше, что мы научились писать такие программы. Следующим подвигом станет программная проверка подписи.

Проверка подписи из программы

Напишем простенькую программку `ec-verify`, которая будет проверять подписи, созданные `ec-sign` или `openssl pkeyutl`.

Наша программа будет принимать три аргумента:

1) имя входного файла, содержащего подписанные данные;
2) имя файла, содержащего подпись;
3) имя файла, содержащего открытый ключ ключей для проверки.

Общий план выглядит следующим образом.

1. Загрузить подпись.
2. Загрузить открытый ключ.
3. Создать объект типа `EVP_MD_CTX`, который будет служить контекстом верификации.
4. Инициализировать контекст подписания, указав функцию хеширования и загруженный открытый ключ.
5. Читать подписанные данные порциями и помещать их в контекст верификации.
6. Завершить процедуру верификации и выяснить, была она успешной или нет.

Приступим к реализации этого плана.

Реализация программы ec-verify

1. Сначала нужно загрузить подпись. Но мы не знаем, сколько для нее выделить памяти. Поэтому откроем файл с подписью и найдем его длину:

   ```
   const char* sig_fname = argv[2];
   FILE* sig_file = fopen(sig_fname, "rb");
   fseek(sig_file, 0, SEEK_END);
   long sig_file_len = ftell(sig_file);
   fseek(sig_file, 0, SEEK_SET);
   ```

   Такой основанный на функции `seek` метод нахождения длины, возможно, и не оптимален, но зато в нем используется только стандартная библиотека C, поэтому он в высшей степени переносим.
2. Зная длину подписи, мы можем выделить для нее память и загрузить подпись из файла:

   ```
   unsigned char* sig_buf = malloc(sig_file_len);
   size_t sig_len = fread(sig_buf, 1, sig_file_len, sig_file);
   ```

3. Загрузим открытый ключ для верификации:

   ```
   const char* pkey_fname = argv[3];
   FILE* pkey_file = fopen(pkey_fname, "rb");
   EVP_PKEY* pkey = PEM_read_PUBKEY(
       pkey_file, NULL, NULL, NULL);
   ```

4. Создадим контекст верификации и инициализируем его функцией хеширования SHA3-512 и загруженным открытым ключом:

   ```
   EVP_MD_CTX* md_ctx = EVP_MD_CTX_new();
   EVP_DigestVerifyInit_ex(
   ```

```
        md_ctx,
        NULL,
        OSSL_DIGEST_NAME_SHA3_512,
        NULL,
        NULL,
        pkey,
        NULL);
```

5. После того как контекст инициализирован, в него можно поместить подписанные данные:

    ```
    while (!feof(in_file)) {
        size_t in_nbytes = fread(in_buf, 1, BUF_SIZE, in_file);
        EVP_DigestVerifyUpdate(md_ctx, in_buf, in_nbytes);
    }
    ```

6. После того как все подписанные данные были пропущены через контекст, можно запросить состояние верификации:

    ```
    int status = EVP_DigestVerifyFinal(md_ctx, sig_buf, sig_len);
    ```

    Функция `EVP_DigestVerifyFinal()` возвращает 1, если проверка подписи была успешной. Любое другое значение означает ошибку. Как и для API подписания, для API верификации есть единая функция `EVP_DigestVerify()`, которая не читает данные порциями, но требует, чтобы все подписанные данные были помещены в непрерывный блок памяти. В версии OpenSSL 3.0 это единственный способ проверить подпись для алгоритмов, не поддерживающих потоковое чтение данных, к числу которых относится, в частности, PureEdDSA.
7. Получив состояние верификации от функции `EVP_DigestVerifyFinal()`, мы можем освободить буферы и прочие объекты:

    ```
    free(in_buf);
    EVP_MD_CTX_free(md_ctx);
    EVP_PKEY_free(pkey);
    free(sig_buf);
    ```

Полный исходный код программы `ec-verify` находится на GitHub по адресу https://github.com/PacktPublishing/Demystifying-Cryptography-with-OpenSSL-3/blob/main/Chapter07/ec-verify.c.

Выполнение программы ec-verify

Запустим программу `ec-verify` и проверим подпись, созданную ранее программой `ec-sign`:

```
$ ./ec-verify \
    somefile.txt \
    somefile.txt.signature \
    ec_public_key.pem
Verification succeeded
```

Отлично – наша программа ec-verify успешно проверила корректную подпись.

Теперь изменим подписанные данные и убедимся, что верификация не проходит:

```
$ echo "additional data" >> somefile.txt
$ ./ec-verify \
    somefile.txt \
    somefile.txt.signature ec_public_key.pem
EVP_API error
Verification failed
```

Как видим, программа ec-verify отличает корректную подпись от некорректной.

Резюме

В этой главе мы узнали о понятии цифровой подписи, о том, как в ней используется хеш-значение сообщения, и о различиях между цифровой подписью и имитовставкой. Мы также узнали, какие криптографические гарантии дают цифровые подписи. После этого привели обзор алгоритмов цифровой подписи, поддерживаемых OpenSSL, обсудили их технические достоинства и немного рассказали об их истории. И закончили теоретическую часть рекомендациями по выбору алгоритма цифровой подписи в конкретной ситуации.

В практической части мы узнали, как создавать и проверять ECDSA-подпись в командной строке и из программы на C.

В следующей главе мы поговорим о сертификатах X.509 и основанной на них **инфраструктуре открытых ключей** (Public Key Infrastructure – **PKI**).

Глава 8

Сертификаты X.509 и инфраструктура открытых ключей

В этой главе мы узнаем о **сертификатах X.509**. Сертификаты – это структуры данных, применяемые для представления и проверки пользователя. Сертификаты X.509 необходимы для функционирования протокола TLS и основанных на нем, например HTTPS, где они используются для доказательства подлинности веб-сайтов. Сертификаты используются также в стандартах безопасного обмена сообщениями, например S/MIME, в VPN-решениях, например OpenVPN, в смарт-картах, при подписании программ и т. д. Факультативно их можно использовать также в протоколе IPsec.

Мы узнаем, из чего состоят сертификаты, как устроены цепочки сертификатов и как работает **инфраструктура открытых ключей** (Public Key Infrastructure – **PKI**). В практической части главы мы научимся генерировать сертификаты и проверять их цепочки в командной строке и из программы на C.

Будут рассмотрены следующие темы:

- что такое сертификат X.509;
- цепочки сертификатов;
- как выпускаются сертификаты X.509;
- что такое расширения X509v3;
- инфраструктура открытых ключей X.509;
- генерирование самоподписанного сертификата;
- генерирование (выпуск) несамоподписанного сертификата;
- проверка сертификата в командной строке;
- проверка сертификата из программы.

Технические требования

В этой главе встречаются команды, запускаемые из командной строки, и C-код, который можно откомпилировать и выполнить. Для запуска команд понадобятся программа `openssl` и соответствующие динамические библиотеки OpenSSL. Для сборки кода на C будут нужны динамические или статические библиотеки OpenSSL, заголовки библиотек, компилятор C и компоновщик.

Мы реализуем демонстрационные программы, чтобы применить полученные знания на практике. Их полный исходный код находится по адресу https://github.com/PacktPublishing/Demystifying-Cryptography-with-OpenSSL-3/tree/main/Chapter08.

Что такое сертификат X.509?

Сертификат – это структура данных, применяемая для представления и проверки пользователя. Сертификат можно сохранить в файле или передать по сети, он также может быть частью безопасного сетевого протокола, например TLS. Сертификат X.509 связывает идентичность объекта с открытым ключом посредством цифровой подписи.

Приведем пример текстового представления сертификата X.509:

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: ... шестнадцатеричные байты ...
        Signature Algorithm: sha256WithRSAEncryption
        Issuer: C = US, O = Let's Encrypt, CN = R3
        Validity
            Not Before: Mar 4 12:43:52 2022 GMT
            Not After : Jun 2 12:43:51 2022 GMT
        Subject: CN = www.openssl.org
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                RSA Public-Key: (2048 битов)
                Modulus: ... шестнадцатеричные байты ...
                Exponent: 65537 (0x10001)
        X509v3 extensions:
          X509v3 Key Usage: critical
              Digital Signature, Key Encipherment
          X509v3 Extended Key Usage:
              TLS Web Server Authentication,
              TLS Web Client Authentication
          X509v3 Basic Constraints: critical
              CA:FALSE
          X509v3 Subject Key Identifier:
              ... шестнадцатеричные байты ...
```

```
X509v3 Authority Key Identifier:
      ... шестнадцатеричные байты ...
... дополнительные расширения X509v3 ...
```

Как видим, сертификат X.509 состоит из следующих полей:

- **X.509 version** (версия X.509): современные сертификаты X.509 имеют версию 3;
- **Serial number** (серийный номер): уникальный номер сертификата среди всех сертификатов, подписанных одним издателем;
- **Signature algorithm** (алгоритм подписи): криптографическая функция хеширования и алгоритм цифровой подписи, использованные для подписания сертификата;
- **Issuer** (издатель): юридическое лицо, подписавшее сертификат, или, в терминологии X.509, выпустившее (издавшее) его. Издатель представлен в формате **различимого имени** (Distinguished Name – **DN**), который будет описан ниже;
- **Validity** (действительность): два поля временных меток, Not Before (не раньше) и Not After (не позже), определяющие срок действия сертификата;
- **Subject** (субъект): юридическое или физическое лицо, идентифицируемое сертификатом. Это может быть человек, сайт, организация или что-то еще. Поле Subject также представлено в формате различимого имени;
- **открытый ключ сертификата**: открытый ключ важен для проверки идентичности, представленной сертификатом. Поддерживаются разные типы ключей, в т. ч. RSA, DSA, ECDSA и EdDSA. Важно помнить, что сертификат X.509 не содержит закрытого ключа, только открытый;
- **X509v3 extensions** (расширения X509v3): содержат дополнительную информацию;
- **подпись сертификата**: предоставляется издателем сертификата.

Как открытые и закрытые ключи, сертификаты X.509 кодируются в абстрактной синтаксической нотации версии 1 (**ASN.1**) и могут быть представлены в одном из двух форматов: **Distinguished Encoding Rules** (**DER**) или **Privacy-Enhanced Mail** (**PEM**).

Поля Subject и Issuer сертификатов X.509 представлены в формате различимого имени, DN. Различимое имя выглядит следующим образом: `C = US, O = Let's Encrypt, CN = R3`. Как видно, DN состоит из пар ключ-значение. У ключей имеется точно определенная семантика; например, `C` означает `Country` (страна), а `O` – `Organization` (организация). Самым важным является ключ `CN`, или `Common Name` (стандартное имя). Это основное имя лица, идентифицируемого сертификатом. Это может быть имя сайта, например `www.openssl.org`; имя человека, например `John Doe`; или если сертификат выпущен для технических целей, то имя самого сертификата, например `Technical certificate 123` или просто `R3`.

Каждому сертификату X.509 соответствует закрытый ключ. Он не включен в состав сертификата, но является парным открытому ключу, включенному в сертификат, т. е. цифровая подпись, порожденная этим закрытым ключом,

может быть проверена открытым ключом в сертификате. Этот закрытый ключ часто называют закрытым ключом сертификата. Владельцу сертификата необходим закрытый ключ, чтобы доказать, что он действительно владеет сертификатом, а не просто скопировал его. Такое доказательство владелец может предъявить, подписав какие-то данные своим закрытым ключом и попросив другую сторону проверить подпись с помощью открытого ключа, извлеченного из сертификата.

Сертификат X.509 обычно не является секретной информацией, его можно свободно распространять, как и открытый ключ. С другой стороны, закрытый ключ сертификата – секрет, который должен быть известен только его владельцу, поскольку всякий обладающий закрытым ключом может объявить себя владельцем сертификата и криптографически строго доказать это. Кража закрытого ключа может привести к краже личности. Например, если кто-то украдет закрытый ключ сертификата сайта `www.openssl.org` и при этом еще и организует атаку **отравления DNS** или **атаку с человеком посередине**, то сможет создать ложный сайт `www.openssl.org`, который пройдет проверку сертификата. Существуют способы смягчить последствия описанной кражи личности, например **списки отозванных сертификатов** (Certificate Revocation List – **CRL**) и **протокол онлайновой проверки состояния сертификата** (Online Certificate Status Protocol – **OCSP**), но у противника все равно может остаться достаточно времени для атаки, прежде чем скомпрометированный сертификат будет отозван. Другой пример: если закрытый ключ сертификата использован для подписания сообщений электронной почты, то всякий укравший ключ сможет подписывать почту от имени владельца сертификата.

Когда какие-то данные подписаны закрытым ключом сертификата, часто говорят, что они подписаны сертификатом. Сами сертификаты тоже могут быть подписаны. На самом деле любой сертификат X.509 подписан каким-то сертификатом, т. е. его закрытым ключом. Подписавший сертификат указан в поле `Issuer` в формате DN. Сертификат может быть подписан и своим собственным закрытым ключом и тогда называется самоподписанным. У самоподписанных сертификатов DN-адреса в полях `Subject` и `Issuer` одинаковы. Бывает и так, что один сертификат подписан другим, тот – третьим и т. д. В таком случае имеется цепочка сертификатов.

Цепочки сертификатов

Цепочкой подписания сертификатов, или **цепочкой проверки сертификатов**, или просто **цепочкой сертификатов**, или даже **цепочкой доверия**, называется упорядоченная последовательность сертификатов, в которой каждый сертификат подписан следующим в цепочке. Кроме, конечно, последнего. Последний сертификат является самоподписанным.

Зачем нужны цепочки сертификатов? Чтобы проверить их действительность. Любопытный читатель может спросить: «А разве закрытый ключ сертификата не решает проблему?» Нет, не все так просто. При проверке иден-

тичности с использованием сертификата X.509 мы должны удостовериться в истинности двух утверждений:

1) **лицо, предъявившее сертификат для идентификации, является его владельцем.** Это утверждение доказывается с помощью закрытого ключа сертификата;
2) **предъявленный сертификат действителен.** Это утверждение доказывается с помощью цепочки сертификатов.

Тут все обстоит так же, как при идентификации человека по паспорту. Вы можете идентифицировать себя, предъявив паспорт, но этот паспорт должен быть действителен.

Как устроена проверка сертификата? Чтобы понять это, мы должны разобраться в том, как сертификаты создаются, или, в терминологии X.509, *выпускаются*.

Сертификаты большинства сайтов в интернете выпущены так называемыми **удостоверяющими центрами** (**УЦ**). УЦ обычно взимает плату за выпуск сертификата. Некоторые УЦ, например Let's Encrypt, выпускают сертификаты бесплатно. CA представляют себя как **центры доверия** (trusted third party). Идея в том, что вы доверяете УЦ выпуск сертификатов и считаете выпущенные ими сертификаты действительными. Вы уверены, что УЦ проверяет личности заказавших сертификат и выпускает сертификат только для тех, чья личность совпадает с записанной в поле Subject сертификата.

Почему можно доверять какой-то организации, с которой вы никогда не встречались, о которой едва слышали и с сотрудниками которой незнакомы? В ответ на этот вопрос УЦ говорят, что доверие – их бизнес; если они станут выпускать поддельные сертификаты, не соответствующие личностям субъектов, то никто не будет доверять УЦ и они обанкротятся. Поэтому вы должны им доверять. Лично мне такая аргументация кажется сомнительной. Но выбора-то все равно нет. Вы (а точнее, ваша ОС или ваш браузер) либо доверяете сертификатам, выпущенным УЦ (или, по крайней мере, притворяетесь, что доверяете), либо не сможете обратиться к HTTPS-сайтам, на которых используются сертификаты, выпущенные этими УЦ, – а это большинство сайтов в интернете. Так устроена современная Всемирная паутина.

Для проверки сертификата необходимо построить цепочку, оканчивающуюся надежным сертификатом. Рассмотрим широко распространенный случай проверки сертификата веб-сайта. Первым сертификатом в цепочке будет сертификат, идентифицирующий сайт. Этот сертификат не подписывает и не выпускает никаких других сертификатов. Сертификаты, которые не подписывают других сертификатов, называются **листовыми**. Сертификат сайта подписан вторым сертификатом в цепочке. В большинстве случаев этот второй сертификат не является самоподписанным, а потому на нем цепочка не обрывается. Сертификаты, которые подписывают другие сертификаты, но не являются самоподписанными, называются **сертификатами промежуточных УЦ**. В типичной цепочке, начинающей сертификатом сайта, будет один или два сертификата промежуточных УЦ. После всех промежуточных сертификатов идет последний сертификат в цепочке, который является самоподписанным. Самоподписанные сертификаты, которые подписывают другие сертификаты, называются **корневыми**.

Цепочка сертификатов выглядит следующим образом:

Рис. 8.1 ❖ Цепочка сертификатов, расположенная горизонтально

В интернете также часто встречается вертикально расположенная цепочка сертификатов:

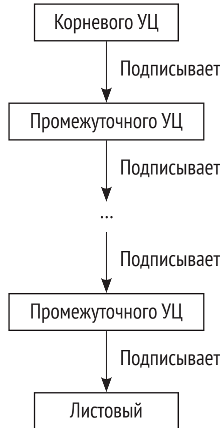


Рис. 8.2 ❖ Цепочка сертификатов, расположенная вертикально

Отметим, что горизонтальное представление выглядит как связанный список, в котором сертификат, подлежащий проверке, находится в начале, а надежный сертификат – в конце. Именно так представлены цепочки сертификатов в OpenSSL.

В наших описаниях и рисунках мы упоминали промежуточные УЦ. Но зачем они нужны? Почему не подписывать все листовые сертификаты корневыми? Тому есть несколько причин:

- если промежуточный сертификат скомпрометирован, то его можно отозвать, не отзывая корневой сертификат и прочие промежуточные сертификаты, подписанные тем же корневым. Поэтому область воздействия скомпрометированного сертификата сужается;
- многие УЦ поддерживают автоматический выпуск сертификатов в онлайновом режиме. Это означает, что закрытый ключ подписывающего сертификата должен находиться на сервере, который прямо или косвенно доступен из интернета. Это создает риски. Лучше подвергнуть риску промежуточный сертификат, чем корневой. Поскольку корневой сертификат подписывает другие не слишком часто, его закрытый ключ можно хранить в безопасном месте, не подключенном к сети, например на флеш-накопителе в стальном сейфе, в запертом и охраняемом помещении;

- списки отозванных сертификатов крупных УЦ могут быть очень велики. Мне встречались такие списки размером 50 МБ. Имея несколько промежуточных сертификатов, УЦ может разделить свой пул выпущенных сертификатов между промежуточными УЦ и публиковать меньшие по размеру CRL, по одному на каждый промежуточный УЦ.

Итак, цепочки сертификатов нужны для проверки сертификатов и могут содержать листовые (концевые), промежуточные и корневые сертификаты. Но кто проверяет эти цепочки и как проверяющая сторона получает все сертификаты? Всякий, кто может получить необходимые сертификаты и построить цепочку, может проверить сертификат. Вернемся к проверке сертификата сайта. В этом случае проверкой сертификата занимается браузер или другой HTTPS-клиент. Браузер получает сертификат сервера во время процедуры квитирования TLS. А как браузер получает сертификат корневого УЦ? Из хранилища сертификатов в браузере или в ОС. В большинстве операционных систем имеется общесистемное хранилище сертификатов, в котором они разбиты на категории: сертификаты надежных УЦ, сертификаты ненадежных УЦ или сертификаты клиентов с закрытыми ключами. Эти хранилища регулярно обновляются с помощью механизмов обновления ОС. В некоторых браузерах имеются собственные хранилища сертификатов, которые можно использовать вместе или вместо хранилища ОС. Такие хранилища обновляются самими браузерами и необязательно в момент выпуска новой версии, а периодически.

Кто решает, каким сертификатам можно доверять? Лицо, отвечающее за обслуживание хранилища сертификатов, решает, как его инициализировать и какие изменения (добавления или удаления) включать в обновления. В случае хранилища ОС обслуживание осуществляет производитель ОС, а в случае хранилища браузера – производитель браузера. Пользователь ОС и браузера также может добавлять и удалять сертификаты. Однако не так много пользователей обладают необходимыми для этого знаниями, да и потребность или желание изменить содержимое хранилища сертификатов возникает нечасто. Поэтому в большинстве случаев хранилище сертификатов содержит только те сертификаты, которые решил включить производитель ОС или браузера.

Содержимое хранилищ сертификатов в разных ОС и браузерах полностью или почти полностью совпадает. В них помещаются сертификаты хорошо известных корневых УЦ. Если вы заходите на сайт, имеющий сертификат, выпущенный одним из этих УЦ, то браузер сможет проверить сертификат сайта и покажет вам страницу. Но если вы заходите на любительский сайт, сертификат которого не выпущен хорошо известным УЦ, то браузер не сможет проверить его сертификат и покажет предупреждение, способное отпугнуть технически слабо подкованного пользователя. Чтобы избежать такого предупреждения, нужно добавить сертификат УЦ, который прямо или опосредованно подписал сертификат сайта, в хранилище ОС или браузера, сделав его тем самым надежным. Эта задача может оказаться трудной для неспециалиста. Поэтому сертификаты большинства HTTPS-сайтов в интернете выпущены хорошо известными УЦ, для которых сертификаты корневых УЦ находятся в хранилище ОС или браузера и считаются надежными. Для небольших любительских сайтов оплата услуг УЦ может оказаться неподъем-

ной. К счастью, начиная с 2015 года можно получить бесплатные сертификаты от некоммерческого УЦ Let's Encrypt. Некоторые крупные коммерческие УЦ иногда тоже выпускают бесплатные HTTPS-сертификаты, обычно для пробного использования.

Внимательный читатель, вероятно, обратил внимание, что в хранилищах сертификатов предусмотрена специальная категория для сертификатов ненадежных УЦ. Зачем она нужна? Ответ простой. Сертификаты ненадежных УЦ – это сертификаты промежуточных УЦ, которые можно использовать для построения цепочки от сертификата сайта до сертификата надежного УЦ. Однако они крайне редко встречаются в хранилищах сертификатов ОС или браузера. Как же тогда браузер получает необходимые промежуточные сертификаты? Обычно веб-сервер посылает сертификаты промежуточных УЦ вместе с сертификатом сайта.

Как строится цепочка сертификатов при проверке сертификата? Очевидно, что первым сертификатом в цепочке является проверяемый, например сертификат сайта. Но как решить, какой сертификат помещать в цепочку следующим? Найти следующий сертификат можно двумя способами. Во-первых, использовать поле `Issuer` текущего сертификата и поискать сертификат с таким же DN в поле `Subject`. Поле `Issuer` всегда заполнено в любом сертификате, но иногда можно найти несколько сертификатов с одним и тем же субъектом. Поэтому поиск по `Issuer` неоднозначен. Второй способ – воспользоваться расширением X509v3 «*Authority Key Identifier*» (идентификатор ключа УЦ), которое содержит криптографический хеш открытого ключа сертификата издателя. Однако расширения X509v3 факультативны, поэтому в конкретном сертификате расширение может отсутствовать. Кроме того, при возобновлении сертификата ключ иногда остается тем же, а срок действия изменяется. Поэтому второй метод тоже неоднозначен. Следовательно, иногда для построения правильной цепочки сертификатов необходимо анализировать несколько путей подписания. Если хотя бы один путь позволяет построить цепочку до надежного сертификата и все ограничения, в частности на срок действия и другие, налагаемые расширениями X509v3, удовлетворены, то можно считать, что сертификат успешно проверен. OpenSSL поддерживает такой анализ нескольких путей подписания при построении цепочки сертификатов.

В большинстве случаев цепочка сертификатов заканчивается самоподписанным сертификатом корневого УЦ. Но, строго говоря, это необязательно. OpenSSL поддерживает цепочки, завершающиеся несамоподписанными сертификатами. Однако использование таких цепочек считается дурным тоном.

Мы многое узнали о проверке сертификатов. Но как они выпускаются?

Как выпускаются сертификаты X.509

Процедура генерирования сертификата X.509 состоит из нескольких шагов:

1) заявитель (будущий владелец сертификата) генерирует закрытый и открытый ключи сертификата;

2) заявитель генерирует **запрос на подписание сертификата** (Certificate Signing Request – **CSR**). CSR содержит наименование субъекта, открытый ключ будущего сертификата, запрошенные заявителем расширения X509v3 и подпись CSR. CSR подписывается закрытым ключом сертификата;
3) заявитель отправляет CSR удостоверяющему центру для подписания;
4) УЦ проверяет личность заявителя;
5) УЦ изготавливает сертификат на основе информации, взятой из CSR. УЦ также добавляет в сертификат другую информацию, например наименование издателя, поля срока действия и расширения X509v3. Наконец, УЦ подписывает сертификат;
6) УЦ отправляет сертификат заявителю, который становится его владельцем, или держателем.

Отметим, что ни на каком этапе ни одна сторона не раскрывает другой свой закрытый ключ.

Скажем несколько слов о том, как УЦ проверяет личность заявителя в случае выпуска сертификата веб-сайта. Некоторые полагают, что УЦ *всегда* выполняет тщательную проверку сайта и его владельцев, сканирует сайт на предмет вредоносных программ, лично встречается с владельцами, проверяет кучу документов и т. д. Обычно ничего такого не происходит. Для большинства выпускаемых сертификатов личность проверяется автоматически в онлайновом режиме, без каких-либо контактов между владельцами сайта и сотрудниками УЦ. Не проверяются никакие документы, скажем паспорта владельцев, договоры с интернет-провайдером и на аренду DNS-имени и т. д. Что же тогда проверяется? Что заявитель действительно управляет сайтом и его доменным именем. Проверки автоматические и могут выглядеть следующим образом:

- разместить некий файл по конкретному URL-адресу, принадлежащему сайту;
- добавить DNS-запись в домен сайта;
- создать адрес электронной почты в домене сайта и получить на этот адрес пароль.

Именно поэтому HTTPS-сертификаты обычно называются сертификатами **с подтвержденным доменом** (Domain-Validated **– DV**).

Что именно удостоверяет такой DV-сертификат сайта? То, что браузер пользователя подключился к сайту с доменным именем, указанным в сертификате, например `www.openssl.org`, – не больше, не меньше. DV-сертификат не гарантирует, что сайт представляет какую-то конкретную организацию, например проект OpenSSL. Некоторые верят в миф, будто если сайт имеет сертификат, то он сертифицирован и, значит, не может представлять недостойную организацию, не может содержать вредоносного ПО, не может атаковать пользователей и что вся информация на этом сайте правдива. Увы, это не так. Сайты, занимающиеся фишингом, распространением вредоносного ПО и другой преступной деятельностью, тоже могут иметь действительные HTTPS-сертификаты, и подключения к ним браузер будет считать «безопасными». Важно понимать, что безопасно только подключение. Сам же сайт может быть вовсе не безопасен.

Почему же DV-сертификаты все-таки полезны? Потому что они защищают пользователей браузера от атак с отравлением DNS и с человеком посередине. Пользователь может с высокой вероятностью предполагать, что действительно находится на сайте, который указан в адресной строке браузера, а не на поддельном сайте, имитирующем подлинный.

Существуют ли сертификаты, подтверждающие, что сайт представляет конкретную организацию, а не только доменное имя? Да, они называются сертификатами **с расширенным подтверждением** (Extended Validation – **EV**). При выпуске EV-сертификата УЦ выполняет гораздо больше проверок запроса на подписание (CSR). УЦ проверяет, что CSR поступил из организации, владеющей доменом, что название организации в поле `Subject` запроса совпадает с истинным названием, что организация существует и надлежащим образом зарегистрирована в государственном реестре организаций, что адрес и телефон организации реальны, что человек, запросивший сертификат, действует от имени организации, и т. д. EV-сертификаты выпускаются только для юридических лиц.

EV-сертификаты стоят значительно дороже DV-сертификатов. Например, на момент написания книги один хорошо известный УЦ взимал за DV-сертификаты 8 долларов в год, а за EV-сертификаты – 90 долларов. На большинстве сайтов в интернете используются DV-сертификаты. EV-сертификаты используются только особо важными организациями, например на сайтах банков.

Об EV-статусе выпущенного сертификата свидетельствует расширение X509v3. К сожалению, на данный момент не существует стандартизованного расширения X509v3 для этой цели; разные УЦ используют разные расширения. Поэтому браузер должен поддерживать несколько расширений, указывающих на EV-статус.

Раньше тот факт, что у сайта имеется EV-сертификат, основные браузеры индицировали зеленым фоном адресной строки и демонстрацией в ней названия компании-владельца сайта. Начиная с 2019 года такая визуальная индикация была прекращена. Зеленой адресной строки больше нет, а название компании показывается, только если щелкнуть по значку замка. Большинство прочих программ и раньше никогда не индицировали наличие EV-сертификата визуально.

Существуют также сертификаты **с подтверждением юридического лица** (Organization Validation – **OV**) и **с подтверждением физического лица** (Individual Validation – **IV**). Они занимают промежуточное положение между DV- и EV-сертификатами. При выпуске таких сертификатов УЦ проверяет подлинность юридического или физического лица, а выпущенный сертификат содержит название владельца в поле `Subject`. Отличие от EV-сертификатов состоит в том, что при обработке CSR выполняется меньше проверок. OV- и IV-сертификаты не столь популярны, как EV-сертификаты. Если организация хочет получить сертификат более высокого ранга, чем DV, то обычно заказывает EV-, а не OV-сертификат.

Мы уже несколько раз упоминали расширения X509v3. Поговорим о них подробнее.

Что такое расширения X509v3?

Расширения X509v3 – это дополнительные поля, которые можно включить в сертификат X.509. Расширения могут налагать ограничения на использование сертификата или предоставлять дополнительную информацию о нем. Для примера рассмотрим расширения X509v3 в сертификате сайта www.openssl.org:

- **использование ключа и расширенное использование ключа**: ограничивают использование сертификата определенными целями. Учитывать эти ограничения или нет – дело программы, проверяющей сертификат. Если проверяющая программа не понимает или не придает значения некоторым ограничениям X509v3, то они игнорируются;
- **базовое ограничение CA:FALSE**: говорит, что этот сертификат нельзя использовать для выпуска других сертификатов, т. е. проверяющая программа не должна рассматривать его как сертификат промежуточного или корневого УЦ;
- **идентификатор ключа субъекта и идентификатор ключа УЦ**: информационные расширения, помогающие искать сертификат издателя по выпущенному сертификату;
- **доступ к информации об УЦ**: информационное расширение, помогающее находить ресурсы УЦ, например OCSP-сервер или сертификаты УЦ, полезные для проверки текущего сертификата;
- **альтернативное имя субъекта**: информационное расширение, в котором перечисляются доменные имена сайтов, которым разрешено использовать этот сертификат. Помимо www.openssl.org, могут быть включены другие имена, например mail.openssl.org или просто openssl.org;
- **подписанные временные метки (SCT) предсертификатов в журналах прозрачности сертификатов (CT)**. CT – это средство мониторинга, помогающее обнаруживать выпуск поддельных сертификатов X.509.

Расширения X509v3 – необязательные части формата сертификата X.509, но благодаря им использовать сертификаты удобнее. Кроме того, некоторые программы, проверяющие сертификаты, например браузеры, могут ожидать присутствия определенных расширений и изменять результат проверки в зависимости от содержимого этих расширений или их отсутствия.

Сертификаты X.509 полезны, но не в отрыве от окружения. Сертификаты являются частью более крупной **инфраструктуры открытых ключей**.

Инфраструктура открытых ключей X.509

Инфраструктура открытых ключей (**PKI**) X.509 – это комбинация стандартов, алгоритмов, структур данных, программного и аппаратного обеспечения, организаций и процедур, необходимых для создания, хранения, передачи,

использования, отзыва и других действий по управлению сертификатами X.509 и их ключами.

Звучит запутанно, но мы только что разобрали, как X.509 PKI работает во Всемирной паутине. УЦ выпускают сертификаты, которые используются сайтами и проверяются браузерами. Именно так миллионы пользователей веба ежедневно и автоматически проверяют подлинность сайтов. Некоторые сайты поддерживают аутентификацию клиентов с помощью клиентских сертификатов. В таких случаях не только сайт должен предъявить свой сертификат, но и браузер пользователя должен предъявить сертификат сайту, чтобы сайт мог сертифицировать, аутентифицировать и авторизовать пользователя.

X.509 PKI используется не только в вебе. Сертификаты X.509 применяются для передачи почтовых сообщений, для взаимодействия автоматизированных компьютерных систем, при подписании программ и т. д.

Некоторые думают, что PKI – это нечто таинственное и не поддающееся пониманию, но на самом деле это не так. Это всего лишь управление ключами и идентичностью.

На этом теоретическая часть закончена. Перейдем к практической части и сгенерируем несколько сертификатов.

Генерирование самоподписанного сертификата

Чтобы сгенерировать сертификат, нужно сгенерировать пару ключей, запрос на подписание сертификата и, наконец, сам сертификат. Программа `openssl` умеет генерировать самоподписанные сертификаты несколькими способами. Один из них – воспользоваться командой, которая сразу генерирует и пару ключей, и сертификат. Но мы сделаем это по отдельности, потому что это более общий способ.

Мы будем использовать следующие подкоманды `openssl`: `genpkey`, `pkey`, `req` и `x509`. Они документированы на страницах руководства:

```
$ man openssl-genpkey
$ man openssl-pkey
$ man openssl-req
$ man openssl-x509
```

Ниже описана процедура генерирования самоподписанного сертификата.

1. Для начала сгенерируем пару ключей. На этот раз выберем тип ключа `ED448`:

   ```
   $ openssl genpkey -algorithm ED448 -out root_keypair.pem
   ```

   Команда ничего не напечатала, это значит, что ошибок не было.
2. Проверим созданный ключ:

   ```
   $ openssl pkey -in root_keypair.pem -noout -text
   ED448 Private-Key:
   priv:
   ```

```
e2:62:21:f0:32:25:20:ca:84:f9:b8:4f:0a:9f:51:
51:3b:68:d0:0d:3a:91:c9:68:38:b4:2f:d0:53:af:
62:5a:06:9d:b0:f5:86:11:73:f5:be:39:9a:78:be:
ec:a2:53:d8:91:ad:8b:e5:2e:e2:b3:a3
pub:
70:3c:22:d9:9f:f8:d6:76:e0:4f:46:e8:74:7b:5f:
98:98:ee:90:49:af:07:ba:05:a4:3b:b3:2c:e3:20:
1a:00:cf:11:5c:76:93:32:0a:91:14:98:fa:dd:83:
7b:9c:00:46:c8:d3:df:67:23:ea:e1:80
```

Выглядит нормально.

3. Сгенерируем CSR-запрос. Чтобы не усложнять задачу, не станем включать много расширений X509v3. О более продвинутой процедуре генерирования сертификатов мы узнаем в главе 12.

```
$ openssl req \
    -new \
    -subj "/CN=Root CA" \
    -addext "basicConstraints=critical,CA:TRUE" \
    -key root_keypair.pem \
    -out root_csr.pem
```

Заметим, что мы добавили расширение `CA:TRUE`. Это необходимо для сертификатов УЦ.

4. Ничего не напечатано, и создан файл, `root_csr.pem`, содержащий наш CSR-запрос. Посмотрим, что в нем:

```
$ openssl req -in root_csr.pem -noout -text
Certificate Request:
    Data:
        Version: 1 (0x0)
        Subject: CN = Root CA
        Subject Public Key Info:
            Public Key Algorithm: ED448
                ED448 Public-Key:
                pub:
                    ... шестнадцатеричные байты ...
        Attributes:
            Requested Extensions:
                X509v3 Basic Constraints: critical
                    CA:TRUE
    Signature Algorithm: ED448
    Signature Value:
        ... шестнадцатеричные байты ...
```

5. Теперь сгенерируем самоподписанный сертификат, используя CSR-запрос и пару ключей. Срок действия нового сертификата будет составлять 3650 дней, т. е. около 10 лет:

```
$ openssl x509 \
    -req \
    -in root_csr.pem \
```

```
    -copy_extensions copyall \
    -key root_keypair.pem \
    -days 3650 \
    -out root_cert.pem
```

Обратите внимание на параметр `-copy_extensions copyall`. Он нужен, потому что по умолчанию команда `openssl x509` не копирует расширения X509v3 из CSR-запроса в сертификат.

6. Ничего не напечатано, и создан файл, `root_cert.pem`, содержащий самоподписанный сертификат. Посмотрим, что в нем:

```
$ openssl x509 -in root_cert.pem -noout -text
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            ... шестнадцатеричные байты ...
        Signature Algorithm: ED448
        Issuer: CN = Root CA
        Validity
            Not Before: Mar 27 22:25:17 2022 GMT
            Not After : Mar 24 22:25:17 2032 GMT
        Subject: CN = Root CA
        Subject Public Key Info:
            Public Key Algorithm: ED448
                ED448 Public-Key:
                pub:
                    ... шестнадцатеричные байты ...
        X509v3 extensions:
            X509v3 Basic Constraints: critical
                CA:TRUE
            X509v3 Subject Key Identifier:
                ... шестнадцатеричные байты ...
    Signature Algorithm: ED448
    Signature Value:
        ... шестнадцатеричные байты ...
```

Обратите внимание, что поля `Issuer` и `Subject` одинаковы. Это признак самоподписанного сертификата.

Мы успешно сгенерировали самоподписанный сертификат, который будем использовать в качестве сертификата корневого УЦ. Теперь сгенерируем два несамоподписанных сертификата.

Генерирование несамоподписанного сертификата

При генерировании самоподписанного сертификата мы воспользовались общим подходом, а не длинными комбинированными командами. Поэтому

процедура генерирования самоподписанного сертификата будет очень похожа. Мы сгенерируем два несамоподписанных сертификата. Один будет использоваться как сертификат промежуточного УЦ, а второй – как листовый сертификат.

1. Сначала сгенерируем пару ключей для сертификата промежуточного УЦ:

```
$ openssl genpkey \
    -algorithm ED448 \
    -out intermediate_keypair.pem
```

2. Затем сгенерируем CSR-запрос:

```
$ openssl req \
    -new \
    -subj "/CN=Intermediate CA" \
    -addext "basicConstraints=critical,CA:TRUE" \
    -key intermediate_keypair.pem \
    -out intermediate_csr.pem
```

   Обратите внимание, что поле Subject для сертификата промежуточного УЦ отличается от предыдущего.
3. Теперь выпустим сертификат промежуточного УЦ и подпишем его закрытым ключом корневого сертификата:

```
$ openssl x509 \
    -req \
    -in intermediate_csr.pem \
    -copy_extensions copyall \
    -CA root_cert.pem \
    -CAkey root_keypair.pem \
    -days 3650 \
    -out intermediate_cert.pem
```

   Здесь мы задали флаги `-CA` и `-CAkey` вместо `-key`. Флаг `-CA` нужен для копирования поля Subject из сертификата УЦ в поле Issuer выпущенного сертификата, а также для заимствования другой необходимой информации из сертификата выпускающего УЦ, например *идентификатора ключа УЦ*, если используется соответствующее расширение X509v3.
4. Посмотрим на выпущенный сертификат промежуточного УЦ:

```
$ openssl x509 -in intermediate_cert.pem -noout -text
Certificate:
    Data:
        Version: 1 (0x0)
        Serial Number:
            ... шестнадцатеричные байты ...
        Signature Algorithm: ED448
        Issuer: CN = Root CA
        Validity
            Not Before: Mar 27 23:11:35 2022 GMT
            Not After : Mar 24 23:11:35 2032 GMT
```

```
        Subject: CN = Intermediate CA
        Subject Public Key Info:
            Public Key Algorithm: ED448
                ED448 Public-Key:
                pub:
                    ... шестнадцатеричные байты ...
        X509v3 extensions:
            X509v3 Basic Constraints: critical
                CA:TRUE
            X509v3 Subject Key Identifier:
                ... шестнадцатеричные байты ...
            X509v3 Authority Key Identifier:
                ... шестнадцатеричные байты ...
    Signature Algorithm: ED448
    Signature Value:
        ... шестнадцатеричные байты ...
```

Обратите внимание, что поля Issuer и Subject различны. Это значит, что сертификат несамоподписанный.

5. Теперь выпустим листовый сертификат и подпишем его закрытым ключом сертификата промежуточного УЦ. Делается это так же, как при выпуске сертификата промежуточного УЦ.

```
$ openssl genpkey -algorithm ED448 -out leaf_keypair.pem
$ openssl req \
    -new \
    -subj "/CN=Leaf" \
    -addext "basicConstraints=critical,CA:FALSE" \
    -key leaf_keypair.pem \
    -out leaf_csr.pem
$ openssl x509 \
    -req \
    -in leaf_csr.pem \
    -copy_extensions copyall
    -CA intermediate_cert.pem \
    -CAkey intermediate_keypair.pem \
    -days 3650 \
    -out leaf_cert.pem
```

На этот раз мы задали `CA:FALSE`, а не `CA:TRUE`, потому что листовые сертификаты не должны использоваться для выпуска других сертификатов.

6. Посмотрим на сгенерированный листовый сертификат:

```
$ openssl x509 -in leaf_cert.pem -noout -text
Certificate:
    Data:
        Version: 1 (0x0)
        Serial Number:
            ... шестнадцатеричные байты ...
        Signature Algorithm: ED448
        Issuer: CN = Intermediate CA
        Validity
            Not Before: Mar 27 23:34:19 2022 GMT
```

```
            Not After : Mar 24 23:34:19 2032 GMT
        Subject: CN = Leaf
        Subject Public Key Info:
            Public Key Algorithm: ED448
                ED448 Public-Key:
                pub:
                    ... шестнадцатеричные байты ...
        X509v3 extensions:
            X509v3 Basic Constraints: critical
                CA:FALSE
            X509v3 Subject Key Identifier:
                ... шестнадцатеричные байты ...
            X509v3 Authority Key Identifier:
                ... шестнадцатеричные байты ...
    Signature Algorithm: ED448
    Signature Value:
        ... шестнадцатеричные байты ...
```

Выглядит хорошо – листовый сертификат выпущен сертификатом промежуточного УЦ.

Теперь проверим листовый сертификат с помощью двух других.

Проверка сертификата в командной строке

Проверить сертификат позволяет команда `openssl verify`, документированная на странице руководства `openssl-verify`.

Проверим только что сгенерированный листовый сертификат. Будем считать, что наш сертификат корневого УЦ надежный. Сертификат промежуточного УЦ будем считать ненадежным, но он позволит построить цепочку сертификатов.

Вот как проверяется листовый сертификат в командной строке:

```
$ openssl verify \
    -verbose \
    -show_chain \
    -trusted root_cert.pem \
    -untrusted intermediate_cert.pem \
    leaf_cert.pem
leaf_cert.pem: OK
Chain:
depth=0: CN = Leaf (untrusted)
depth=1: CN = Intermediate CA (untrusted)
depth=2: CN = Root CA
```

Обратите внимание на флаги `-trusted` и `-untrusted`. Флаг `-trusted` задает файл, содержащий один или несколько надежных сертификатов. Чтобы проверить сертификат, `openssl verify` должна построить цепочку от проверяемого сертификата до надежного. Флаг `-untrusted` задает файл, содержащий один или несколько ненадежных сертификатов. Ненадежные сертификаты

полезны в качестве промежуточных в цепочке. Оба флага можно использовать несколько раз для задания нескольких файлов.

Команда `openssl verify` сообщила `OK`, т. е. проверка сертификата завершилась успешно. Также напечатана цепочка сертификатов, построенная в ходе проверки.

Проверка сертификата из программы

OpenSSL 3.0 предоставляет только один API для проверки сертификата. Он включает функции, начинающиеся префиксом `X509_`.

Мы напишем небольшую программу, которая будет проверять сгенерированный ранее листовый сертификат точно так же, как это делает `openssl verify`.

Перечислим страницы руководства, относящиеся к используемому API:

```
$ man X509_STORE_new
$ man X509_STORE_load_file
$ man DEFINE_STACK_OF
$ man PEM_read_x509
$ man X509_STORE_CTX_new
$ man X509_verify_cert
$ man X509_STORE_CTX_get_error
$ man X509_free
```

Наша программа будет принимать три аргумента:

1) имя файла, содержащего надежные сертификаты;
2) имя файла, содержащего ненадежные сертификаты;
3) имя файла, содержащего проверяемый сертификат.

Составим общий план.

1. Загрузить надежные сертификаты.
2. Загрузить ненадежные сертификаты.
3. Загрузить подлежащий проверке сертификат.
4. Создать и инициализировать контекст проверки сертификата.
5. Проверить сертификат.

Реализация программы x509-verify

1. Сначала загрузим надежные сертификаты. Последовательность надежных сертификатов нужно передать функции проверки в виде контейнера `X509_STORE`. По счастью, OpenSSL предлагает удобную функцию, `X509_STORE_load_file()`, для загрузки сертификатов из PEM-файла в `X509_STORE`:

   ```
   const char* trusted_cert_fname = argv[1];
   X509_STORE* trusted_store = X509_STORE_new();
   X509_STORE_load_file(trusted_store, trusted_cert_fname);
   ```

2. Далее нужно загрузить ненадежные сертификаты. Их следует подать на вход функции проверки в виде объекта `STACK_OF(X509)`. В OpenSSL сертификаты X.509 представляются типом `X509`, а `STACK_OF` – макрос, который строит стек объектов. Этот макрос напоминает шаблонный тип `std::stack` в C++. Он принимает тип элементов стека в качестве параметра. Стеки используются во многих API и в самом коде OpenSSL. Объект `STACK_OF(X509)` часто служит для представления цепочки подписания сертификатов или просто списков сертификатов. Функции OpenSSL, работающие со стеками, например `sk_X509_new_null()` и `sk_X509_push()`, тоже являются макросами. Их имена начинаются префиксом `sk_` и параметризованы типом элементов стека. Например, в именах функций `sk_`, работающих с объектом `STACK_OF(X509)`, встречается подстрока `X509`. Иначе говоря, имена начинаются не просто с `sk_`, а с `sk_X509_`. Поскольку в данном случае подстрока `X509` – всего лишь параметр в имени функции, то вы не найдете страницы руководства с названием `sk_X509_push`. Зато найдете страницу руководства `sk_TYPE_push`, являющуюся псевдонимом страницы `DEFINE_STACK_OF`.
3. К сожалению, в OpenSSL нет удобной функции, которая загружала бы сертификаты в стек `STACK_OF(X509)` одной операцией. Поэтому загружать их придется по одному, пользуясь функцией `PEM_read_X509()`. Сначала определим длину файла, содержащего ненадежные сертификаты:

```
const char* untrusted_cert_fname = argv[2];
FILE* untrusted_cert_file = fopen(untrusted_cert_fname, “rb”);
fseek(untrusted_cert_file, 0, SEEK_END);
long untrusted_cert_file_len = ftell(untrusted_cert_file);
fseek(untrusted_cert_file, 0, SEEK_SET);
```

4. Затем загрузим сертификаты по одному в структуру `STACK_OF(X509)`:

```
STACK_OF(X509)* untrusted_stack = sk_X509_new_null();
while (ftell(untrusted_cert_file) < untrusted_cert_file_len){
    X509* untrusted_cert =
        PEM_read_X509(untrusted_cert_file, NULL, NULL, NULL);
    sk_X509_push(untrusted_stack, untrusted_cert);
}
```

   Мы не можем просто написать цикл с условием `while(!feof(untrusted_cert_file))`, потому что функция `feof()` обнаруживает конец файла лишь тогда, когда не было попытки читать за пределами файла. А если вызвать `PEM_read_X509()` в конце файла, до того как конец обнаружен, то `PEM_read_X509()` вернет ошибку. Поэтому мы проверяем в цикле положение указателя файла.
5. Загрузив надежные и ненадежные сертификаты, мы должны загрузить сертификат, подлежащий проверке. Он всего один, поэтому можно воспользоваться функцией `PEM_read_X509()`:

```
const char* target_cert_fname = argv[3];
FILE* target_cert_file = fopen(target_cert_fname, “rb”);
```

```
X509* target_cert =
PEM_read_X509(target_cert_file, NULL, NULL, NULL);
```

6. Все сертификаты загружены. Теперь можно создать и инициализировать контекст проверки сертификатов, объект типа `X509_STORE_CTX`:

```
X509_STORE_CTX* ctx = X509_STORE_CTX_new();
X509_STORE_CTX_init(
    ctx, trusted_store, target_cert, untrusted_stack);
```

7. И вот решающий момент – проверка сертификата с использованием контекста:

```
int err = X509_verify_cert(ctx);
```

   Функция возвращает 1, если сертификат успешно проверен. Любое другое значение означает ошибку.
8. В случае ошибки информацию о ней можно получить с помощью функций `X509_STORE_CTX_get_error()` и `X509_verify_cert_error_string()`:

```
if (err != 1) {
    int error_code = X509_STORE_CTX_get_error(ctx);
    const char* error_string =
        X509_verify_cert_error_string(error_code);
    fprintf(
        stderr,
        "X509 verification error: %s\n",
        error_string);
}
```

9. После завершения проверки объекты следует освободить:

```
X509_STORE_CTX_free(ctx);
X509_free(target_cert);
sk_X509_pop_free(untrusted_stack, X509_free);
X509_STORE_free(trusted_store);
```

Полный исходный код программы `x509-verify` имеется на GitHub по адресу https://github.com/PacktPublishing/Demystifying-Cryptography-with-OpenSSL-3/blob/main/Chapter08/x509-verify.c.

Выполнение программы x509-verify

Запустим нашу программу `x509-verify` и проверим листовый сертификат, воспользовавшись сертификатом корневого УЦ как надежным и сертификатом промежуточного УЦ как ненадежным:

```
$ ./x509-verify \
    root_cert.pem \
    intermediate_cert.pem \
    leaf_cert.pem
Verification succeeded
```

Как и следовало ожидать, проверка сертификата прошла успешно.

Теперь рассмотрим случай ошибки. На этот раз не станем предоставлять нашей программе сертификат промежуточного УЦ, поэтому она не сможет построить цепочку до надежного сертификата. Но программе все равно нужно передать три аргумента, поэтому повторим аргумент `leaf_cert.pem` дважды:

```
$ ./x509-verify \
    root_cert.pem \
    leaf_cert.pem \
    leaf_cert.pem
X509 verification error: unable to get local issuer certificate
Verification failed
```

Проверка не прошла – как и должно быть.

Как видим, программа `x509-verify` правильно обрабатывает оба случая: успех и ошибку.

Резюме

В этой главе мы узнали о сертификатах X.509 – зачем они нужны и какую информацию содержат. Мы также узнали о цепочках сертификатов и их роли в проверке сертификатов. Затем поговорили об УЦ, а также о разнице между сертификатами корневого и промежуточных УЦ. Мы рассказали о процедуре выпуска сертификатов X.509 и о нескольких типах сертификатов, в т. ч. с проверкой домена и о расширенном типе (EV). После этого перешли к расширениям X509v3. И закончили теоретическую часть описанием концепции PKI.

В практической части главы мы узнали о том, как генерируются самоподписанные и несамоподписанные сертификаты. Затем показали, как проверить сертификат в командной строке и из программы на C.

В следующей главе мы поговорим об установлении TLS-подключений и передаче по ним данных.

Часть IV

TLS-подключения и безопасная связь

В этой части мы узнаем о протоколе TLS, пришедшем на смену SSL: по какой причине он необходим и почему так широко распространен в современном интернете. Затем научимся создавать безопасные каналы связи с помощью протокола TLS, отправлять и принимать по ним данные и закрывать каналы. Мы также узнаем, почему для TLS-подключений так важны сертификаты X.509 и как они позволяют противостоять атакам с человеком посередине. Далее покажем, как вставить в TLS-подключение сертификат и как удаленная сторона может проверить его. Работа с TLS и сертификатами иллюстрируется примерами использования командной строки и кода на C.

Эта часть состоит из трех глав:

- главы 9 «Установление TLS-подключений и передача по ним данных»;
- главы 10 «Использование сертификатов X.509 в TLS»;
- главы 11 «Специализированные применения TLS».

Глава 9

Установление TLS-подключений и передача по ним данных

В этой главе мы узнаем о **безопасном протоколе транспортного уровня** (Transport Layer Security – **TLS**). Протокол TLS, пришедший на смену **протоколу безопасных советов** (Secure Sockets Layer – **SSL**), используется для организации безопасной связи по сети и служит основой для протоколов более высокого уровня, в т. ч. **безопасного протокола передачи гипертекста** (Hypertext Transfer Protocol Secure – **HTTPS**) и **безопасного простого протокола электронной почты** (Simple Mail Transfer Protocol Secure – **SMTPS**). Протокол TLS лучше всего виден в вебе, но используется и в других приложениях, например для передачи файлов, электронной почты, мгновенного обмена сообщениями, передачи голоса по IP, удаленного доступа, подключения к базам данных, передачи финансовых данных и во многих других приложениях, нуждающихся в зашифрованной связи.

Мы узнаем об основах протокола TLS и кратко познакомимся с его историей. В практической части главы мы научимся устанавливать TLS-подключения, передавать по ним данные и правильно их закрывать. Будут приведены примеры работы с TLS из командной строки и из программы на C.

В этой главе рассматриваются следующие темы:

- что такое протокол TLS;
- история протокола TLS;
- установление клиентского TLS-подключения в командной строке;
- подготовка сертификатов для подключения к TLS-серверу;
- прием запроса на подключение к TLS-серверу в командной строке;
- базовые объекты ввода-вывода в OpenSSL;
- установление клиентского TLS-подключения из программы;
- прием запроса на подключение к TLS-серверу из программы.

Технические требования

В этой главе встречаются команды, запускаемые из командной строки, и C-код, который можно откомпилировать и выполнить. Для запуска команд понадобится программа `openssl` и соответствующие динамические библиотеки OpenSSL. Для сборки кода на C будут нужны динамические или статические библиотеки OpenSSL, заголовки библиотек, компилятор C и компоновщик.

Мы реализуем демонстрационные программы, чтобы применить полученные знания на практике. Их полный исходный код находится по адресу https://github.com/PacktPublishing/Demystifying-Cryptography-with-OpenSSL-3/tree/main/Chapter09.

Что такое протокол TLS

TLS – протокол общего назначения, ставящий целью обеспечить безопасную связь. На момент написания книги последней версией была 1.3. TLS обеспечивает следующие аспекты безопасности:

- конфиденциальность передаваемых пользовательских данных путем **симметричного шифрования**. Вычислительно невозможно расшифровать и прочитать передаваемые данные. Здесь и далее под *пользовательскими данными* я понимаю данные, которыми обмениваются пользователи протокола TLS, отправитель и получатель, – в отличие от служебных данных, необходимых самому протоколу для функционирования;
- целостность и аутентичность передаваемых пользовательских данных, защита от изменения путем **шифрования с аутентификацией** или **кода аутентичности сообщения** (Message Authentication Code – **MAC**), или имитовставки. Вычислительно невозможно изменить передаваемые пользовательские данные так, чтобы принимающая сторона этого не заметила. Начиная с версии TLS 1.3 шифрование с аутентификацией обязательно;
- доказательство подлинности сторон и защита от **атак с человеком посередине** с помощью **сертификатов X.509**;
- **совершенная прямая секретность** (Perfect Forward Secrecy – **PFS**) посредством обмена ключами по **эфемерному протоколу Диффи–Хеллмана** (Ephemeral Diffie-Hellman – **DHE**) или **эфемерному протоколу Диффи–Хеллмана на эллиптических кривых** (Ephemeral Elliptic Curve Diffie-Hellman – **ECDHE**). Вычислительно невозможно расшифровать перехваченный и записанный TLS-сеанс, даже если закрытый ключ сертификата сервера окажется скомпрометирован в будущем. Начиная с версии TLS 1.3 PFS-обмен ключами обязателен.

Что такое PFS? Чтобы ответить на этот вопрос, мы должны разобраться в обмене ключами. Под **обменом ключами**, или **совместной выработкой**

ключа в TLS, понимается процедура согласования ключа симметричного шифрования, который будет использован для шифрования пользовательских данных в сеансе TLS. В самом старом методе **обмена ключами RSA** клиент генерирует симметричный ключ и шифрует его открытым ключом из сертификата сервера. В более современных методах используются варианты протокола обмена ключами **Диффи–Хеллмана** (**DH**), когда и у клиента, и у сервера имеется асимметричная пара ключей, а симметричный ключ формируется из собственного закрытого ключа и открытого ключа другой стороны. В ранних версиях TLS допускались методы обмена ключами, не удовлетворяющие требованию PFS, например RSA и статические варианты DH/ECDH. При использовании такого метода пара ключей из сертификата сервера используется как для аутентификации, так и для обмена ключами. В таком случае если противник записал TLS-сеанс, а впоследствии украл закрытый ключ из сертификата сервера, то он сможет восстановить симметричный сеансовый ключ, использованный в этом сеансе, и, следовательно, расшифровать пользовательские данные, отправленные в этом сеансе. При использовании PFS-метода обмена ключами, например DHE или ECDHE, клиент и сервер генерируют временные (эфемерные) пары ключей для обмена ключами. Эти эфемерные пары нужны только для обмена и сразу после него уничтожаются. В таком случае, даже если ключ из сертификата сервера будет украден, противник не сможет расшифровать TLS-сеанс, потому что не располагает эфемерным закрытым ключом из этого сеанса. Как видим, при применении PFS-метода для аутентификации используется пара ключей из сертификата, а для обмена ключами – эфемерная пара ключей. Мы можем заключить, что PFS в протоколе TLS это именно то свойство метода обмена ключами, которое не дает противнику расшифровать записанный TLS-сеанс, даже если ключ из сертификата сервера будет скомпрометирован.

Обычно TLS работает поверх какого-нибудь надежного транспортного протокола, например **протокола управления передачей** (Transmission Control Protocol – **TCP**). Чаще всего TLS используется совместно с протоколом прикладного уровня, например **Hypertext Transfer Protocol** (**HTTP**), работающим поверх TLS. Уровни протоколов для этого типичного случая представлены на рисунке ниже.

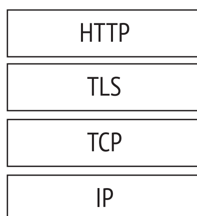


Рис. 9.1 ❖ Уровни протоколов

Принято считать, что прикладной протокол, работающий поверх TLS, является безопасным вариантом этого протокола. Например, протокол HTTPS – безопасный вариант HTTP. HTTPS – это HTTP поверх TLS, а SMTPS –

SMTP поверх TLS. Легко видеть, что названия безопасных вариантов прикладных протоколов заканчиваются буквой *S*. Исходный прикладной протокол и его TLS-вариант обычно прослушивают разные порты. Например, HTTP по умолчанию прослушивает порт `80`, а HTTPS – порт `443`. Но некоторые протоколы поддерживают оба варианта на одном порту. Так, SMTP поддерживает команду `STARTTLS`, которая *доводит* текущее TCP-подключение до уровня SMTPS. Этот метод явного создания TLS-сеанса называется **оппортунистическим TLS**.

Существует также модификация протокола TLS под названием **безопасный протокол транспортного уровня дейтаграмм** (Datagram Transport Layer Security – **DTLS**), предназначенный для работы поверх ненадежного транспортного протокола, например **протокола пользовательских дейтаграмм** (User Datagram Protocol – **UDP**). Отметим, что TLS поверх TCP распространен гораздо шире, чем DTLS поверх UDP. Однако популярные программы, использующие DTLS, все же имеются, например OpenVPN, которая может использовать как DTLS поверх UDP, так и обычный режим – TLS поверх TCP.

Как и TCP, TLS – клиент-серверный протокол. Чтобы начать передавать данные по TLS, клиент и сервер должны сначала установить TCP-подключение и выполнить TLS-квитирование.

TLS-квитирование

Клиент обычно начинает процедуру TLS-квитирования сразу после установления TCP-подключения, но в случае оппортунистического TLS клиент и сервер сначала обмениваются незашифрованными данными.

Как именно протекает TLS-квитирование, зависит от версии протокола TLS, но важные шаги перечислены ниже.

- Клиент отправляет список поддерживаемых им версий протокола TLS и комплектов шифров. В версии TLS 1.3 имя комплекта шифров – это составное имя вида `TLS_AES_256_GCM_SHA384` или `TLS_CHACHA20_POLY1305_SHA256`. Как видим, имя комплекта шифров определяет алгоритм шифрования, например `AES_256` или `CHACHA20`, режим работы блочного шифра, например `GCM`, или метод аутентификации шифрования, например `POLY1305`, и криптографическую функцию хеширования, например `SHA384`. В старых версиях TLS используется другая схема именования комплектов шифров с более длинными именами, например `TLS_DHE_RSA_WITH_AES_256_CBC_SHA256`. В этом случае комплект шифров включает также метод обмена ключами, например `DHE`, и алгоритм подписания сертификата, например `RSA`. Эта схема была отвергнута, потому что приводит к комбинаторному взрыву из-за слишком большого числа категорий алгоритмов и режимов: алгоритм обмена ключами, алгоритм подписания сертификата, симметричный шифр, длина симметричного ключа, режим работы блочного шифра и алгоритм вычисления хеша сообщения. В TLS 1.3 поддерживаются только режимы **шифрования с аутентификацией и присоединенными данными** (Authenticated

Encryption with Associated Data – **AEAD**). Старые версии TLS поддерживают режимы шифрования без аутентификации, и тогда для аутентификации зашифрованных данных применяется **имитовставка на основе функции хеширования** (**HMAC**).

- Сервер выбирает самую старшую версию протокола и предпочтительный комплект шифров из тех, что поддерживаются и клиентом, и сервером. Если не удается согласовать версию протокола или комплект шифров, то квитирование завершается безрезультатно.
- Сервер отправляет свой сертификат X.509, возможно, вместе с другими сертификатами, которые позволяют построить цепочку подписания. Клиент проверяет сертификат сервера, пользуясь имеющимся у него надежным сертификатом **удостоверяющего центра** (**УЦ**). Клиент также проверяет, что в поле Subject находится доменное имя сервера. Дополнительные сведения об использовании сертификатов X.509 см. в главе 10.
- Сервер подписывает небольшую порцию данных, полученную от клиента, своим закрытым ключом, который соответствует открытому ключу в сертификате сервера. Клиент проверяет подпись. Тем самым сервер доказывает, что действительно является владельцем сертификата. В прежних версиях TLS существовал альтернативный способ доказать владение сертификатом: клиент шифрует данные открытым ключом из сертификата сервера, а сервер расшифровывает их своим закрытым ключом. Но этот способ, включающий шифрование и расшифрование, работает только с RSA-ключами и сертификатами сервера.
- Факультативно клиент отправляет свой сертификат, и сервер проверяет его.
- Если клиент отправил свой сертификат, то он также доказывает владение им, подписывая или расшифровывая какие-то данные своим закрытым ключом.
- Клиент и сервер согласуют симметричный сеансовый ключ и **вектор инициализации** (**IV**), которые будут использоваться для шифрования пользовательских данных, передаваемых по протоколу TLS. Клиент и сервер применяют алгоритм обмена ключами и совместной выработки ключей, например *ECDHE*, чтобы согласовать **квитированный секрет**, который в прежних версиях TLS назывался **предварительным мастер-секретом**. Затем клиент и сервер используют **псевдослучайную функцию** (pseudorandom function – **PRF**), например **функцию формирования ключа на основе HMAC** (HMAC-based key derivation function – **HKDF**), для генерирования **мастер-секрета**, а впоследствии также сеансового ключа и IV. Вместо использования алгоритма выработки ключа клиент может запросить повторное использование сохраненного квитированного секрета из прежнего TLS-сеанса. Это экономит вычислительные ресурсы, потому что асимметричная криптография, применяемая в процессе совместной выработки, обходится довольно дорого.
- Клиент и сервер обмениваются данными о расширениях TCP, которые хотели бы использовать. Например, популярным расширением явля-

ется **индикация имени сервера** (Server Name Indication – **SNI**); она позволяет выбрать конкретный сайт на общем хостинге, где на одном и том же сервере и порту может обслуживаться много сайтов. Другое полезное расширение TCP – `pre_shared_key` – позволяет использовать предварительно выданные ключи и повторно применять квитированный секрет из прежнего сеанса, о чем упоминалось выше.

- Наконец, клиент и сервер аутентифицируют все ранее отправленные в ходе квитирования сообщения с помощью HMAC. Каждая сторона проверяет аутентификацию. Если проверка HMAC не проходит, считается, что квитирование завершилось безрезультатно, и подключение сбрасывается. Эта мера уменьшает риск манипулирования квитированием сторонним злоумышленником, например убеждающим стороны использовать менее стойкий шифр, когда подключение еще в должной мере не защищено.

Важность TLS-квитирования трудно переоценить, но это всего лишь часть TLS-подключения. Посмотрим, что происходит дальше.

Что происходит после TLS-квитирования?

После TLS-квитирования клиент и сервер могут посылать друг другу сообщения, инкапсулированные в *TLS-записи*. Передаваемые данные шифруются сеансовым ключом и аутентифицируются либо аутентификационным жетоном, либо HMAC, если используется старая версия протокола в режиме шифрования без аутентификации. Получатель проверяет аутентификацию каждой TLS-записи и прерывает подключение в случае обнаружения ошибки.

Когда одна сторона TLS-подключения обнаруживает ошибку, она должна уведомить другую сторону с помощью тревожного сообщения. Тревожные сообщения могут указывать на фатальную ошибку или содержать предупреждение. При получении фатальной ошибки получатель должен прекратить отправку и прием данных.

Одно из возможных тревожных сообщений называется `close_notify`. Оно служит для правильного размыкания TLS-подключения. С помощью такого предупреждения одна сторона уведомляет другую о том, что закрывает свою часть подключения и больше не будет передавать данные. В версии TLS 1.3 сторона, отправившая сообщение `close_notify`, может продолжить чтение данных из подключения. В прежних версиях сторона, получившая `close_notify`, была обязана отбросить все ожидающие операции записи и сама отправить `close_notify alert`, что привело бы к закрытию TLS-подключения. Спецификация протокола TLS обязывает клиента и сервера отправлять сообщения `close_notify`, когда подключение подошло к логическому завершению и больше не предполагается ни отправлять, ни получать данные. Это требование позволит свести к минимуму возможность атаки с досрочным прерыванием. Если обе стороны получили друг от друга сообщение `close_notify`, то могут быть уверены, что не человек посередине прервал подключе-

ние. Заметим, что закрытие TLS-подключения не означает автоматического закрытия базового TCP-подключения. TCP-подключение нужно закрывать отдельно после закрытия TLS-подключения – конечно, если не осталось еще не переданных данных.

Отметим использование различных криптографических средств, рассмотренных ранее, в построении протокола TLS. Симметричное шифрование нужно для шифрования пользовательских данных. Криптографические функции хеширования лежат в основе HMAC, цифровых подписей и сертификатов. Шифрование с аутентификацией и HMAC нужны для аутентификации пользовательских и служебных данных. Функции формирования ключа используются для генерирования секретов, сеансовых ключей и IV. Цифровые подписи применяются для подтверждения действительности и владения сертификатом X.509. Сертификаты X.509 используются для проверки личности и уменьшения риска атак с человеком посередине. Протокол TLS в том виде, в каком мы его сейчас знаем, – результат исследований и экспериментов многих людей на протяжении многих лет; он объединяет в себе многочисленные криптографические технологии, чтобы дать нам средства безопасного взаимодействия по сети.

В этом разделе мы несколько раз упоминали о различиях между TLS 1.3 и более старыми версиями протокола. В следующем разделе мы немного поговорим о его истории.

История протокола TLS

Протокол TLS пришел на смену SSL. Протокол SSL был первоначально разработан компанией Netscape Communications Corporation в 1990-х годах.

Версия SSL 1.0 не была обнародована и не использовалась ни в каких известных программных продуктах из-за дефектов, влияющих на безопасность протокола. Она применялась только внутри Netscape.

Версия SSL 2.0 была опубликована в 1995 году. Она поддерживала симметричные шифры DES, 3DES, RC2, RC4 и IDEA, имитовставку на основе MD5 (не HMAC), обмен ключами по протоколу RSA и сертификаты на основе RSA. Криптоаналитики очень скоро нашли в протоколе многочисленные дефекты. В SSL 2.0 была нестойкая MAC-аутентификация, незащищенные процедуры квитирования, и, как выяснилось, он был уязвим для атак с удлинением, досрочным прерыванием, понижением версии шифра и человеком посередине. SSL 2.0 не снискал большой популярности и вскоре уступил место версии SSL 3.0, появившейся в следующем году. Но хотя версия SSL 2.0 почти не использовалась, официально она была объявлена нерекомендуемой только в 2011 году.

Версия SSL 3.0, опубликованная в 1996 году, стала реакцией на дефекты, обнаруженные в SSL 2.0. Протокол был полностью перепроектирован. Все существующие версии протокола TLS возведены на фундаменте SSL 3.0 и имеют с ним много общего. В SSL 3.0 исправлены дефекты SSL 2.0 и по-

явилось много новых возможностей, в т. ч. поддержка обмена эфемерными и неэфемерными ключами по протоколу Диффи–Хеллмана, MAC на основе SHA-1, DSA-подписанные сертификаты, сжатие, повторное использование сеансов, повторное согласование параметров сеанса, аутентификация квитирования и согласование версии протокола. Поддержка слабых шифров, в т. ч. DES, RC2 и 40-битового RC4, была исключена. SSL 3.0 никогда официально не поддерживал шифр AES, хотя некоторые библиотеки, в т. ч. OpenSSL, поддерживали AES в качестве нестандартного расширения. SSL 3.0 был популярен долго, но со временем в нем были обнаружены уязвимости к некоторым атакам, в т. ч. **Browser Exploit Against SSL/TLS** (**BEAST**), **Compression Ratio Info-Leak Made Easy** (**CRIME**) и особенно **Padding Oracle On Downgraded Legacy Encryption** (**POODLE**). Стоит отметить, что библиотеки SSL смогли смягчить последствия этих изъянов в протоколе SSL путем расщепления записей и запрета сжатия. В SSL 3.0 также использовалась неудачная конструкция «MAC, затем шифрование». Еще одним недостатком SSL 3.0 было отсутствие поддержки расширений TLS. Из-за этого в браузерах возникала вполне реальная проблема – при согласовании параметров протокола SSL 3.0 невозможно было использовать важное расширение SNI для задания имени веб-сервера на общем хостинге. В 2015 году SSL 3.0 был наконец объявлен нерекомендуемым.

Протокол TLS 1.0 был опубликован в 1999 году как модернизация SSL 3.0. Спецификация TLS 1.0 разрабатывалась не Netscape, а **рабочей группой по TLS** (TLS Working Group – **TLS WG**), организованной **Инженерным советом интернета** (Internet Engineering Task Force – **IETF**). Протокол был переименован в TLS, чтобы избежать юридических сложностей с Netscape. В версии TLS 1.0 была введена поддержка расширений TLS и улучшены различные аспекты протокола, в частности PRF, MAC, сообщение *Finished* и тревожные сообщения. В отличие от SSL, в TLS 1.0 использовалась HMAC для аутентификации TLS-записей. Впоследствии в TLS 1.0 были включены дополнительные симметричные шифры – AES, Camellia, SEED и ГОСТ89, – а также поддержка эллиптической криптографии, в частности ECDSA-подписанных сертификатов и методов выработки ключа ECDH/ECDHE. Версия TLS 1.0 была уязвима для атак BEAST и CRIME точно так же, как SSL 3.0. Оригинальная атака POODLE для TLS 1.0 не работала, но позже обнаружилось, что на некоторых серверах с TLS 1.0 использовались библиотеки TLS с ошибкой, делавшей возможной модифицированную атаку POODLE. Протокол TLS 1.0 был объявлен нерекомендуемым в 2021 году.

Версия TLS 1.1 была опубликована в 2006 году как дальнейшее развитие TLS 1.0. В ней использовался явный зашифрованный IV вместо неявного IV в прежних версиях протокола, поэтому он был неуязвим для атаки BEAST. В TLS 1.1 также была включена обработка ошибок дополнения для предотвращения атак, связанных с CBC-дополнением, улучшено повторное использование сеансов и внедрены другие усовершенствования. Несмотря на все это, TLS 1.1 не поддерживал новых криптографических алгоритмов, таких как режимы шифрования AEAD и функции хеширования SHA-2. TLS 1.1 опирался на комплекты шифров на основе режима CBC, уязвимые для атаки дополнения с оракулом, а также на ныне нерекомендуемые хеш-функции MD5 и SHA-1

в качестве MAC и PRF. TLS 1.1 был официально объявлен нерекомендуемым вместе с TLS 1.0 в 2021 году.

TLS 1.2 был опубликован в 2008 году. Добавлены режимы шифрования AEAD, в частности GCM и CCM, и более стойкие хеш-функции, SHA-256 и SHA-384, в качестве MAC и PRF. Одновременно была исключена поддержка устаревших алгоритмов, таких как DES, IDEA, MD5 и SHA-1. Впоследствии в TLS 1.2 была добавлена поддержка алгоритмов ARIA, ChaCha20 и Poly1305. Режимы шифрования AEAD покончили с атаками дополнения с оракулом типа POODLE. Как обычно, в новой версии протокола было много мелких улучшений различных аспектов протокола. На момент написания этой книги TLS 1.2 еще не объявлен нерекомендуемым.

Версия TLS 1.3, опубликованная в 2018-м, стала крупной чисткой протокола TLS. Из-за непрестанного расширения функционала в прежних версиях TLS накопилось много унаследованного. В TLS 1.3 исключена поддержка многих небезопасных или устаревших алгоритмов и средств, в т. ч. комплектов шифров без режима AEAD, методов обмена ключами без PFS, таких как RSA и неэфемерные DH/ECDH, DSA-подписанных сертификатов, слабых и малоиспользуемых эллиптических кривых, согласования формата точки эллиптической кривой, специальных групп DHE, протокола ChangeCipherSpec, сжатия и повторного согласования параметров. TLS 1.3 поддерживает только симметричные AEAD-шифры AES-GCM, AES-CCM, ChaCha20-Poly1305, HMAC на основе SHA-256 и SHA-384, методы обмена ключами DHE/ECDHE. В качестве псевдослучайной функции используется HKDF. Поддерживаются сертификаты на основе RSA, ECDSA и PureEdDSA. Сертификаты на основе RSA и ECDSA должны быть подписаны с применением функций хеширования SHA-256, SHA-384 или SHA-512; однако ради обратной совместимости пока еще поддерживаются сертификаты, подписанные с применением хеша сообщения SHA-1. В результате исключения устаревшей функциональности TLS 1.3 стал неуязвим для атак CRIME и POODLE. Квитирование в TLS 1.3 быстрее, чем в предыдущих версиях, потому что требуется только один обмен данными вместо двух. TLS 1.3 также поддерживает кардинальное ускорение работы за счет режима квитирования **с нулевым оборотным временем** (zero round-trip time – **0-RTT**), который позволяет TLS-клиенту отправлять зашифрованные данные в первой же передаваемой порции, если можно повторно использовать предыдущий сеанс TLS. В TLS 1.3 имеется новый механизм повторного использования сеансов, который поддерживает возобновления предыдущего сеанса и предварительную выдачу ключей. Имеются и другие, не столь важные усовершенствования.

Какая версия TLS лучшая? Разумеется, последняя! Но, возможно, вам придется поддерживать прежние версии ради совместимости со сторонами, использующими старые коммуникационные программы. Так часто бывает, когда оборудование перестает получать обновления ПО, как, например, старые мобильные телефоны, планшеты, маршрутизаторы или устройства для **интернета вещей** (Internet-of-Things – **IoT**).

Теперь перейдем к практической части главы и научимся создавать TLS-подключения в командной строке и из программы.

Установление клиентского TLS-подключения в командной строке

Для установления клиентского TLS-подключения мы воспользуемся подкомандой `openssl s_client`, документированной на странице руководства:

```
$ man openssl-s_client
```

В интернете имеется HTTPS-сервер, предназначенный для использования в качестве примера, https://example.org/. Подключимся к нему по TLS и запросим домашнюю страницу:

```
$ openssl s_client -connect example.org:443
```

Программа `openssl` выводит много информации о том, как прошло TLS-квитирование, какие используются криптографические алгоритмы. Печатается даже сертификат сервера в кодировке `base64`.

Мы также можем запросить проверку сертификата сервера и его доменного имени, добавив флаг `-verify_return_error` и `-verify_hostname`:

```
$ openssl s_client \
    -connect example.org:443 \
    -verify_return_error \
    -verify_hostname example.org
```

Если вы хотите проверить сертификат сервера, то нужно поместить сертификаты надежных УЦ в каталог, где OpenSSL ожидает их найти. Например, в моей системе Ubuntu Linux системный OpenSSL ищет надежные сертификаты в каталоге `/etc/ssl/certs`. Если же вы установите свою версию OpenSSL, например в каталог `/opt/openssl-3.0.0`, то она будет искать сертификаты в каталоге `/opt/openssl-3.0.0/ssl/certs`. В таком случае сообщить OpenSSL о надежных сертификатах можно одним из следующих методов:

- скопировать все файлы, включая символические ссылки, из системного каталога `certs`, например `/etc/ssl/certs`, в каталог `/opt/openssl-3.0.0/ssl/certs`;
- сделать каталог `/opt/openssl-3.0.0/ssl/certs` символической ссылкой на `/etc/ssl/certs`;
- записать в переменную окружения `SSL_CERT_DIR` значение `/etc/ssl/certs`;
- добавить параметр `-CApath /etc/ssl/certs` в команду `openssl s_client`.

Если вы работаете в ОС, не имеющей системного хранилища сертификатов для OpenSSL, например Windows, то можете скачать подготовленный Mozilla пакет сертификатов надежных УЦ с сайта утилиты `curl`. На момент написания книги пакет был размещен по URL-адресу https://curl.se/ca/cacert.pem. Скачать его можно, например, с помощью самой утилиты `curl`:

```
$ curl --remote-name https://curl.se/ca/cacert.pem
```

Затем скачанный пакет сертификатов можно предъявить OpenSSL одним из следующих методов:

- записать в переменную окружения `SSL_CERT_FILE` абсолютный или относительный путь к скачанному файлу `cacert.pem`;
- добавить параметр `-CAfile path/to/cacerts.pem` в команду `openssl s_client`.

После успешного установления TLS-подключения мы можем ввести или скопировать любые данные для передачи по нему. Все данные, передаваемые серверу, будут печататься на терминале. Введите следующие строки, чтобы отправить HTTP-запрос на получение домашней страницы сервера:

```
GET / HTTP/1.1
Host: example.org
Connection: close
(пустая строка, просто нажмите Enter)
```

Мы получим HTTP-ответ, содержащий заголовки и HTML-код домашней страницы сервера. Сервер закроет подключение, потому что мы задали заголовок `Connection: close` в HTTP-запросе. Мы вернемся к командной строке. Если указанный заголовок был опущен, то сервер не закроет соединение. Чтобы корректно закрыть его, разомкнув TLS и TCP-подключения, нужно нажать клавишу Q, а за ней *Enter* или имитировать **конец файла** (**EOF**) на терминале, нажав *Ctrl+D* в Unix-системах или *Ctrl+Z+Enter* в Windows. Если вы хотите принудительно выйти из `openssl s_client` без корректного размыкания подключения, нажмите *Ctrl+C*.

Как видите, все просто. Теперь посмотрим, как установить TLS-подключение на стороне сервера. Для этого необходимы сертификат сервера и соответствующий ему закрытый ключ. Следовательно, сначала нужно сгенерировать два сертификата.

Подготовка сертификатов для подключения к TLS-серверу

Чтобы принять запрос на подключение к TLS-серверу, нам необходимо сгенерировать две пары ключей и два сертификата: сертификат сервера и самоподписанный сертификат УЦ, которым будет подписан сертификат сервера. У читателя может возникнуть вопрос, почему бы не сгенерировать просто самоподписанный сертификат сервера. Потому что использование самоподписанного сертификата сервера большинство TLS-клиентов и библиотек, в т. ч. OpenSSL, считают ошибкой. Тогда другой вопрос: почему бы не использовать повторно сертификаты из хранилища ОС для нашего TLS-сервера? Да потому, что у нас нет их закрытых ключей.

Как и в главе 8, мы будем использовать подкоманды `openssl req` и `openssl x509` для генерирования пар ключей и сертификатов. Но на этот раз мы еще и продемонстрируем комбинированные команды генерирования.

Сначала сгенерируем сертификат УЦ. Воспользуемся командой, которая объединяет генерирование пары ключей, **запроса на подписание сертификата** (**CSR**) и сертификата в одной операции. Это возможно только для генерирования самоподписанного сертификата.

```
$ openssl req \
    -newkey ED448 \
    -x509 \
    -subj "/CN=Root CA" \
    -addext "basicConstraints=critical,CA:TRUE" \
    -days 3650 \
    -noenc \
    -keyout ca_keypair.pem \
    -out ca_cert.pem
```

Теперь у нас есть пара ключей и сертификат УЦ, так что можно создать и подписать сертификат сервера. И снова мы воспользуемся комбинированной командой, но объединить все три операции не сможем. Можно объединить только две: генерирование пары ключей сервера и CSR:

```
$ openssl req \
    -newkey ED448 \
    -subj "/CN=localhost" \
    -addext "basicConstraints=critical,CA:FALSE" \
    -noenc \
    -keyout server_keypair.pem \
    -out server_csr.pem
```

Следующая команда создаст (или выпустит) сертификат сервера, подписанный сертификатом УЦ:

```
$ openssl x509 \
    -req \
    -in server_csr.pem \
    -copy_extensions copyall \
    -CA ca_cert.pem \
    -CAkey ca_keypair.pem \
    -days 3650 \
    -out server_cert.pem
```

Заметим, что мы указали имя сервера (`localhost`) в поле Subject сертификата. Это необходимо, чтобы проверка имени сервера прошла успешно, если TLS-клиент захочет ее выполнить.

Подготовив сертификаты, попробуем установить подключение к TLS-серверу.

Прием запроса на подключение к TLS-серверу в командной строке

1. Чтобы принять запрос на подключение к TLS-серверу, мы воспользуемся подкомандой `openssl s_server`, документированной на странице руководства:

   ```
   $ man openssl-s_server
   ```

2. В подкоманде `openssl s_server` укажем номер порта, сертификат сервера и соответствующую пару ключей. Вот как запускается TLS-сервер:

   ```
   $ openssl s_server \
       -port 4433 \
       -key server_keypair.pem \
       -cert server_cert.pem
   ```

3. Чтобы проверить, может ли наш сервер принимать подключения и передавать и принимать по ним данные, мы можем запустить TLS-клиента в другом окне терминала и подключиться к серверу:

   ```
   $ openssl s_client \
       -connect localhost:4433 \
       -verify_return_error \
       -verify_hostname localhost \
       -CAfile ca_cert.pem
   ```

   Заметим, что на этот раз мы задали флаг `-CAfile` в подкоманде `openssl s_client`, чтобы клиент мог найти сертификат надежного УЦ и проверить сертификат сервера.
4. Теперь TLS-клиент и TLS-сервер, работающие в разных окнах, соединены и могут отправлять друг другу данные. Наберите что-нибудь в одном окне, нажмите *Enter*, и вы увидите, что введенная строка появилась в другом окне. Для завершения подключения отправьте EOF или нажмите *Ctrl+C*.

Мы научились создавать клиентские и серверные TLS-подключения в командной строке и передавать по ним данные. Теперь посмотрим, как то же самое делается из программы.

Базовые объекты ввода-вывода в OpenSSL

Чтобы установить TLS-подключение и передавать по нему данные, мы будем использовать **базовые объекты ввода-вывода** (Basic Input/Output – **BIO**). BIO предоставляют одинаковый **интерфейс прикладного программирова-**

ния (**API**) для работы с разными типами каналов **ввода-вывода**: файлами, сокетами и TLS-потоками.

BIO делятся на два типа: **источники**, или **стоки**, и **фильтры**. BIO источника, или стока, представляет концевую точку ввода-вывода, например файл или сокет. BIO фильтра преобразует проходящие через него данные. Например, шифровальный BIO зашифровывает данные при записи и расшифровывает при чтении. BIO можно соединять и строить из них цепочки. Например, SSL BIO можно соединить с BIO сокета, организовав тем самым TLS-связь через сокет.

OpenSSL поддерживает следующие BIO источника или стока:

- BIO приема запроса на подключение (`BIO_s_accept`): функции приема запроса к сокету TCP/IP или Unix;
- спаренный BIO (`BIO_s_bio`): пара BIO такая, что данные, записываемые в одну половину пары, можно прочитать из другой половины;
- BIO подключения (`BIO_s_connect`): функции подключения к сокету TCP/IP или Unix;
- дескрипторный BIO (`BIO_s_fd`): ввод-вывод с помощью целочисленного **файлового дескриптора** (**fd**);
- файловый BIO (`BIO_s_file`): ввод-вывод с помощью указателей на `FILE` в языке C;
- BIO памяти (`BIO_s_mem`): ввод-вывод в буферы памяти и обратно;
- нулевой BIO (`BIO_s_null`): BIO, который отбрасывает все записанные в него данные и не содержит никаких данных для чтения. Работает по аналогии с устройством `/dev/null` в Unix и `NUL` в Windows;
- BIO сокета (`BIO_s_socket`): сокет TCP/IP или Unix, предназначенный для ввода-вывода.

OpenSSL также поддерживает следующие BIO фильтров:

- Base64 BIO (`BIO_f_base64`): кодирование и декодирование Base64;
- буферизующий BIO (`BIO_f_buffer`): буферизация для следующего BIO в цепочке;
- шифровальный BIO (`BIO_f_cipher`): симметричное шифрование и расшифрование;
- BIO хеша сообщений (`BIO_f_md`): вычисление хеш-значения данных, записанных в BIO или прочитанных из BIO;
- BIO нулевого фильтра (`BIO_f_null`): проходной BIO, который просто пропускает данные к следующему BIO. Не путать с нулевым BIO источника или стока;
- SSL BIO (`BIO_f_ssl`): функции протоколов SSL и TLS. Как правило, используются совместно с BIO сокета, но могут применяться и с BIO других типов. По историческим причинам, в OpenSSL все еще используется подстрока *SSL*, а не *TLS* в именах объектов и функций;
- BIO буферизации чтения (`BIO_f_readbuffer`): допускающий только чтение BIO, который использует собственный буфер для реализации поддержки функций `tell` и `seek` в потоковом BIO.

Для работы с BIO используется API, состоящий из функций с префиксом `BIO_`.

Для создания BIO служит функция BIO_new(), а также функции, специализированные для BIO конкретного типа, например BIO_new_socket() или BIO_new_accept(). Некоторые функции даже могут создавать короткие цепочки BIO. Например, BIO_new_buffer_ssl_connect() создает цепочку BIO, состоящую из буферизующего BIO, SSL BIO и BIO подключения.

Чтение из BIO и запись в BIO производятся такими функциями, как BIO_read(), BIO_write(), BIO_read_ex(), BIO_write_ex(), BIO_gets(), BIO_get_line() и BIO_puts().

Функция BIO_puts() удобна, когда требуется записать в BIO строку, завершающуюся нулем. Отметим, что, в отличие от функции puts() из стандартной библиотеки C, BIO_puts() не добавляет в конец символ новой строки, т. е. работает, как fputs().

BIO_gets() читает строку из BIO и, как fgets(), оставляет символ новой строки в прочитанных данных. BIO_gets() возвращает число прочитанных байтов, но с одним подвохом: если прочитанные данные содержат нулевой символ, то BIO_gets() вернет число байтов до первого нулевого символа, даже если реально было прочитано больше. Поэтому лучше использовать функцию BIO_get_line(), добавленную в OpenSSL 3.0, которая всегда возвращает истинное число прочитанных байтов вне зависимости от наличия нулевых символов.

Функции чтения и записи поддерживаются большинством BIO. Но некоторые BIO поддерживают также специфические функции. Например, SSL BIO включает функцию BIO_do_handshake(), которая выполняет TLS-квитирование.

Об ошибке в процессе работы BIO обычно сигнализирует как кодом возврата, так и добавлением ошибки в очередь ошибок OpenSSL. Поэтому после операций с BIO необходимо просматривать, обрабатывать и очищать очередь ошибок.

Когда объект BIO больше не нужен, его следует освободить функцией BIO_free(). Вызов BIO_free() может оказывать влияние на используемые структуры ввода-вывода, например могут закрываться файлы или сетевые подключения. Существует также функция BIO_free_all(), которая освобождает всю цепочку BIO.

Дополнительные сведения о BIO можно найти на соответствующих страницах руководства. Вот некоторые из них:

```
$ man bio
$ man BIO_new
$ man BIO_s_connect
$ man BIO_s_accept
$ man BIO_f_ssl
$ man BIO_read
$ man BIO_push
$ man BIO_ctrl
```

В следующем разделе мы испытаем BIO в деле. Мы будем использовать BIO для установления TLS-подключения, передачи и приема данных по нему и корректного размыкания.

Установление клиентского TLS-подключения из программы

Мы напишем небольшую программу `tls-client`, которая будет подключаться к HTTPS-серверу по протоколу TLS, отправлять HTTP-запрос и читать ответ от сервера.

Для этого воспользуемся BIO API и SSL API, предоставляемыми OpenSSL. BIO API позволит создать TLS-подключение, отправлять и принимать по нему данные и корректно разомкнуть его. SSL API поможет настроить проверку сертификата сервера, выяснять, активно ли еще соединение, и различать разные типы ошибок. Как уже было сказано, по историческим причинам в OpenSSL все еще используется подстрока *SSL* вместо *TLS* в именах объектов и функций, работающих с TLS.

Дополнительные сведения о SSL API можно найти на страницах руководства. Вот некоторые из них:

```
$ man ssl
$ man SSL_CTX_new
$ man SSL_CTX_free
$ man SSL_CTX_load_verify_locations
$ man SSL_CTX_use_certificate_chain_file
$ man SSL_CTX_set_verify
$ man SSL_new
$ man SSL_free
$ man SSL_set_tlsext_host_name
$ man SSL_set1_host
```

Наша программа будет принимать три аргумента:

1) имя сервера;
2) порт сервера;
3) необязательный аргумент: имя файла, содержащего один или более сертификатов надежных УЦ для проверки сертификата сервера.

Ниже приведен общий план реализации.

1. Создать контекст SSL, объект `SSL_CTX`.
2. Загрузить в контекст SSL сертификаты надежных УЦ.
3. Разрешить проверку сертификата сервера в контексте SSL.
4. Создать SSL BIO из контекста SSL.
5. Установить TLS-подключение к серверу.
6. Отправить серверу HTTP-запрос.
7. Прочитать ответ сервера.
8. Разомкнуть TLS-подключение.

Реализация программы tls-client

1. Сначала выделим память для буферов чтения и записи:

```
const size_t BUF_SIZE = 16 * 1024;
char* in_buf = malloc(BUF_SIZE);
char* out_buf = malloc(BUF_SIZE);
```

2. Затем создадим контекст SSL:

```
SSL_CTX* ctx = SSL_CTX_new(TLS_client_method());
```

   Контекст SSL – объект, в котором хранятся параметры и данные для создания TLS-сеанса. Примером данных может служить коллекция сертификатов надежных УЦ. Примером параметра может служить флаг проверки сертификата другой стороны. Оба примера будут использованы в коде. Многие параметры, задаваемые в контексте SSL, могут быть заданы и для отдельных TLS-подключений, представленных объектами типа `SSL`. Если один и тот же параметр задан в объекте `SSL_CTX` и в объекте `SSL`, то параметр в объекте `SSL` имеет преимущество.
   Обратите внимание, что мы передали `TLS_client_method()` в качестве аргумента функции `SSL_CTX_new()`. Так мы указываем, что TLS-подключения, создаваемые в этом контексте, будут клиентскими.
3. Следующий шаг – создание надежных сертификатов. Если при вызове программы был задан третий аргумент, то мы загрузим сертификаты из указанного файла. В противном случае сертификаты загружаются из мест, подразумеваемых по умолчанию, например из файла `/etc/ssl/certs` в Ubuntu Linux или из путей, заданных переменными окружения `SSL_CERT_DIR` и `SSL_CERT_FILE`:

```
const char* trusted_cert_fname = argv[3];
if (trusted_cert_fname)
    err = SSL_CTX_load_verify_locations(
        ctx, trusted_cert_fname, NULL);
else
    err = SSL_CTX_set_default_verify_paths(ctx);
if (err <= 0) {
    if (error_stream)
        fprintf(
            error_stream,
            "Could not load trusted certificates\n");
    goto failure;
}
```

4. Установим некоторые параметры в контексте SSL:

```
SSL_CTX_set_verify(ctx, SSL_VERIFY_PEER, NULL);
SSL_CTX_set_mode(ctx, SSL_MODE_AUTO_RETRY);
```

Задание флага SSL_VERIFY_PEER разрешает проверку сертификата другой стороны. В нашем случае другой стороной является TLS-сервер, так что его сертификат и проверяется. Если проверка не пройдет, то процедура TLS-квитирования завершится ошибкой и TLS-подключение будет разорвано.

Задание флага SSL_MODE_AUTO_RETRY упрощает программирование ввода-вывода по протоколу TLS. Бывает так, что мы хотим записать пользовательские данные в TLS-подключение, а протокол TLS в это время хочет прочитать какие-то служебные данные, чтобы продолжить, и тогда попытка записи заканчивается ошибкой SSL_ERROR_WANT_READ. В таком случае мы должны прочитать данные из подключения, а затем повторить попытку записи. И наоборот, случается, что мы хотим читать, а протокол намеревается записывать, тогда мы получаем ошибку SSL_ERROR_WANT_WRITE. Если флаг SSL_MODE_AUTO_RETRY поднят, то OpenSSL будет повторно выполнять чтение или запись самостоятельно, и нам обрабатывать ошибки SSL_ERROR_WANT_READ и SSL_ERROR_WANT_WRITE не придется.

5. Следующий шаг – создать SSL BIO, используя контекст SSL, и задать имя и порт удаленного TLS-сервера:

```
const char* hostname = argv[1];
const char* port = argv[2];
BIO* ssl_bio = BIO_new_ssl_connect(ctx);
BIO_set_conn_hostname(ssl_bio, hostname);
BIO_set_conn_port(ssl_bio, port);
```

SSL BIO содержит объект типа SSL, представляющий TLS-подключение.

6. Мы должны получить этот объект и установить в нем параметры:

```
SSL* ssl = NULL;
BIO_get_ssl(ssl_bio, &ssl);
SSL_set_tlsext_host_name(ssl, hostname);
SSL_set1_host(ssl, hostname);
```

Функция SSL_set_tlsext_host_name() задает имя сервера для расширения SNI, чтобы можно было подключиться к нужному серверу на общем хостинге. Функция SSL_set1_host() задает имя сервера для проверки сертификата. В процессе квитирования OpenSSL будет сравнивать указанное имя с именами в полях сертификата **Common Name** (**CN**) и **Subject Alternative Names** (**SAN**). Если совпадения не будет, то процедура квитирования закончится ошибкой и TLS-подключение будет разорвано. Если не вызывать функцию SSL_set1_host(), проверка имени сервера не производится – проверяется только цепочка сертификатов.

7. Следующий шаг – установить TLS-подключение:

```
err = BIO_do_connect(ssl_bio);
if (err <= 0) {
    if (error_stream)
        fprintf(
            error_stream,
            "Could not connect to server %s on port %s\n",
```

```
            hostname,
            port);
    goto failure;
}
```

Функция `BIO_do_connect()` пытается выполнить процедуру квитирования обоих протоколов: TCP и TLS. Для удобства SSL BIO позволяет опустить явный вызов `BIO_do_connect()`. Попытка квитирования TCP и TLS также предпринимается при первой операции чтения или записи с помощью SSL BIO. Однако обычно имеет смысл устанавливать TLS-подключение явно и обрабатывать возможные ошибки подключения после вызова.

8. Следующий шаг – отправить HTTP-запрос:

```
snprintf(
    out_buf,
    BUF_SIZE,
    "GET / HTTP/1.1\r\n"
    "Host: %s\r\n"
    "Connection: close\r\n"
    "User-Agent: Example TLS client\r\n"
    "\r\n",
    hostname);
int request_length = strlen(out_buf);
printf("*** Sending to the server:\n");
printf("%s", out_buf);
int nbytes_written =
    BIO_write(ssl_bio, out_buf, request_length);
if (nbytes_written != request_length) {
    if (error_stream)
        fprintf(
            error_stream,
            "Could not send all data to the server\n");
    goto failure;
}
printf("*** Sending to the server finished\n");
```

9. Далее читаем ответ сервера:

```
printf("*** Receiving from the server:\n");
while ((SSL_get_shutdown(ssl) & SSL_RECEIVED_SHUTDOWN)
        != SSL_RECEIVED_SHUTDOWN) {
    int nbytes_read = BIO_read(ssl_bio, in_buf, BUF_SIZE);
    if (nbytes_read <= 0) {
        int ssl_error = SSL_get_error(ssl, nbytes_read);
        if (ssl_error == SSL_ERROR_ZERO_RETURN)
            break;
        if (error_stream)
            fprintf(
                error_stream,
                "Error %i while reading data"
                " from the server\n",
```

```
                ssl_error);
            goto failure;
        }
        fwrite(in_buf, 1, nbytes_read, stdout);
    }
    printf("*** Receiving from the server finished\n");
```

10. Мы читаем данные от сервера, пока тот не закроет свою сторону TLS-подключения. После этого рекомендуется закрыть нашу сторону:

    ```
    BIO_ssl_shutdown(ssl_bio);
    ```

11. Закрыв соединение, мы должны освободить память, занятую объектами и буферами:

    ```
    if (ssl_bio)
        BIO_free_all(ssl_bio);
    if (ctx)
        SSL_CTX_free(ctx);
    free(out_buf);
    free(in_buf);
    ```

12. Наконец, нужно проверить и очистить очередь ошибок OpenSSL:

    ```
    if (ERR_peek_error()) {
        exit_code = 1
        if (error_stream) {
            fprintf(
                error_stream,
                "Errors from the OpenSSL error queue:\n");
            ERR_print_errors_fp(error_stream);
        }
        ERR_clear_error();
    }
    ```

Полный исходный код программы `tls-client` находится на GitHub по адресу https://github.com/PacktPublishing/Demystifying-Cryptography-with-OpenSSL-3/blob/main/Chapter09/tls-client.c.

Выполнение программы tls-client

Запустим нашу программу, указав сайт `example.org`:

```
$ ./tls-client example.org 443
*** Sending to the server:
GET / HTTP/1.1
Host: example.org
Connection: close
User-Agent: Example TLS client
*** Sending to the server finished
```

```
*** Receiving from the server:
HTTP/1.1 200 OK
...
*** Receiving from the server finished
TLS communication succeeded
```

Как видим, программа успешно подключилась к серверу, отправила и приняла данные и успешно закрыла подключение.

Проверим также, как обрабатываются ошибки. Для этого запустим `tls-client`, указав не относящийся к TLS порт на том же сервере:

```
$ ./tls-client example.org 80
Could not connect to server example.org on port 80
Errors from the OpenSSL error queue:
C0F17C57A97F0000:error:0A00010B:SSL routines:ssl3_get_record
:wrong version number
:ssl/record/ssl3_record.c:354:
C0F17C57A97F0000:error:0A000197:SSL routines:SSL_shutdown
:shutdown while in init
:ssl/ssl_lib.c:2242:
TLS communication failed
```

Как видим, программа `tls-client` обнаружила ошибки и сообщила о них. Мы можем заключить, что `tls-client` умеет взаимодействовать с TLS-сервером и обрабатывать ошибки.

В следующем разделе мы научимся устанавливать серверное TLS-подключение и отправлять по нему данные.

Прием запроса на подключение к TLS-серверу из программы

Мы напишем небольшую программу `tls-server`, которая будет принимать запросы на TLS-подключения, читать HTTP-запрос от подключившегося TLS-клиента и отправлять ему HTTP-ответ.

Наша программа будет принимать три аргумента:

1) порт сервера;
2) имя файла, содержащего пару ключей TLS-сервера;
3) имя файла, содержащего цепочку сертификатов TLS-сервера.

В нашем случае цепочка сертификатов будет содержать всего один сертификат – самого сервера. Но если бы были промежуточные сертификаты, то мы могли бы добавить их в файл после сертификата сервера, чтобы помочь TLS-клиенту выполнить проверку. Не имеет особого смысла включать в цепочку сертификат корневого УЦ, потому что TLS-клиент в любом случае должен иметь его среди надежных сертификатов, чтобы проверить сертификат сервера.

Общий план реализации состоит из следующих пунктов.

1. Создать контекст SSL, объект типа `SSL_CTX`.
2. Загрузить в контекст SSL пару ключей сервера.
3. Загрузить в контекст SSL цепочку сертификатов сервера.
4. Проверить, что загруженная пара ключей соответствует загруженному сертификату сервера.
5. Создать BIO приема запроса на подключение из контекста SSL и начать прием входящих TCP-подключений.
6. Когда TCP-подключение принято, BIO приема запроса на подключение создает новый BIO сокета, представляющий принятое подключение. Мы отсоединим BIO сокета от BIO приема запроса на подключение.
7. Создать новый SSL BIO и связать его с BIO сокета.
8. Обработать принятое подключение с помощью сформированной цепочки из двух BIO: SSL и сокета. План обработки подключения мы изложим чуть ниже.
9. Завершив обработку подключения, продолжить прослушивание порта и прием входящих запросов на подключение.

Ниже приведен план обработки принятых подключений.

1. Выполнить процедуру TLS-квитирования для принятого подключения.
2. Прочитать HTTP-запрос от TLS-клиента.
3. Отправить ответ сервера.
4. Разомкнуть TLS-подключение.

Реализация программы tls-server

1. Сначала создадим контекст `SSL`:

   ```
   SSL_CTX* ctx = SSL_CTX_new(TLS_server_method());
   ```

   Обратите внимание, что мы передали `TLS_server_method()` в качестве аргумента функции `SSL_CTX_new()`. Так мы указываем, что TLS-подключения, создаваемые в этом контексте, будут серверными.
2. Затем загрузим пару ключей сервера:

   ```
   const char* server_keypair_fname = argv[2];
   err = SSL_CTX_use_PrivateKey_file(
           ctx, server_keypair_fname, SSL_FILETYPE_PEM);
   if (err <= 0) {
       if (error_stream)
           fprintf(
               error_stream,
               “Could not load server keypair from file %s\n”,
               server_keypair_fname);
       goto failure;
   }
   ```

3. Далее загрузим цепочку сертификатов сервера:

```
const char* server_cert_chain_fname = argv[3];
err = SSL_CTX_use_certificate_chain_file(
        ctx, server_cert_chain_fname);
if (err <= 0) {
    if (error_stream)
        fprintf(
            error_stream,
            "Could not load server certificate chain"
            " from file %s\n",
            server_cert_chain_fname);
    goto failure;
}
```

4. Проверим, что загруженная пара ключей соответствует сертификату сервера, т. е. первому сертификату в загруженной цепочке:

```
err = SSL_CTX_check_private_key(ctx);
if (err <= 0) {
    if (error_stream)
        fprintf(
            error_stream,
            "Server keypair does not match"
            " server certificate\n");
    goto failure;
}
```

5. Установим тот же флаг `SSL_MODE_AUTO_RETRY`, что при разработке программы `tls-client` в предыдущем разделе:

```
SSL_CTX_set_mode(ctx, SSL_MODE_AUTO_RETRY);
```

   В режиме сервера мы не собираемся запрашивать и проверять сертификат другой стороны (клиента), поэтому флаг `SSL_VERIFY_PEER` устанавливать не нужно.
6. Создадим новый объект приема запроса на подключение и начнем прослушивать указанный порт:

```
const char* port = argv[1];
BIO* accept_bio = BIO_new_accept(port);
err = BIO_do_accept(accept_bio);
if (err <= 0) {
    if (error_stream)
        fprintf(
            error_stream,
            "Could not bind to port %s and start listening"
            " for incoming TCP connections\n",
            port);
    goto failure;
}
```

7. Организуем цикл, в котором будем прослушивать запросы на входящие TCP-подключения:

```
while (1) {
    printf("\n");
    printf("*** Listening on port %s\n", port);
    printf("\n");
```

8. Внутри цикла пытаемся принять входящее TCP-подключение:

```
err = BIO_do_accept(accept_bio);
if (err <= 0) {
    if (error_stream)
        fprintf(
            error_stream,
            "Error when trying to accept connection\n");
    if (ERR_peek_error()) {
        if (error_stream) {
            fprintf(
                error_stream,
                "Errors from the OpenSSL error queue:\n");
            ERR_print_errors_fp(error_stream);
        }
        ERR_clear_error();
    }
    continue;
}
```

Отметим, что мы вызываем `BIO_do_accept()` второй раз. Так работает BIO приема запроса на подключение. Первый вызов `BIO_do_accept()` настраивает прослушивающий сокет. Он эквивалентен системным вызовам `bind()` и `listen()`. Второй вызов `BIO_do_accept()` осуществляет фактический прием входящего TCP-подключения к прослушивающему сокету. Он эквивалентен системному вызову `accept()`.
После того как запрос на TCP-подключение успешно принят, BIO создает новый BIO сокета по аналогии с тем, как системный вызов `accept()` создает новый сокет подключения. BIO приема запроса на подключение также присоединяется к вновь созданному BIO сокета, создавая цепочку BIO прием–сокет. Новый BIO представляет принятое подключение, в отличие от прослушивающего сокета или BIO. Мы собираемся отсоединить BIO сокета, создать SSL BIO и вставить его в цепочку перед BIO сокета, а все последующее взаимодействие производить во вновь созданном SSL BIO. Проделаем эти шаги один за другим.

9. Отсоединить BIO сокета от BIO приема:

```
BIO* socket_bio = BIO_pop(accept_bio);
```

10. Создать новый SSL BIO:

```
BIO* ssl_bio = BIO_new_ssl(ctx, 0);
```

11. Вставить SSL BIO перед BIO сокета:

```
BIO_push(ssl_bio, socket_bio);
```

12. Все последующее взаимодействие производится в SSL BIO в отдельной функции `handle_accepted_connection()`:

```
handle_accepted_connection(ssl_bio, error_stream);
```

13. Напомним, что мы все еще находимся в цикле `while()` прослушивания. Пора его закрыть:

```
} // конец цикла while
```

Функция `handle_accepted_connection()` выполняет TLS-квитирование, получение и отправку данных по подключению, его размыкание, обработку ошибок и освобождение SSL BIO и BIO сокета. При желании эту функцию можно выполнять в отдельном потоке. Тогда несколько входящих подключений можно будет обрабатывать одновременно. BIO приема запроса на подключение уже отсоединен от BIO сокета, поэтому может принимать следующие подключения, не оказывая влияния на те, что обрабатываются в данный момент. Для простоты и переносимости мы в этом примере запустили только один поток.

Пора реализовать функцию `handle_accepted_connection()`.

1. Сначала выделим память для буфера чтения:

```
const size_t BUF_SIZE = 16 * 1024;
char* in_buf = malloc(BUF_SIZE);
```

2. Затем выполним TLS-квитирование:

```
err = BIO_do_handshake(ssl_bio);
if (err <= 0) {
    if (error_stream)
        fprintf(error_stream, “TLS handshaking error\n”);
    goto failure;
}
```

3. Далее читаем HTTP-запрос от клиента. Для простоты мы не будем тщательно разбирать запрос, а прекратим чтение, встретив первую пустую строку:

```
printf(“*** Receiving from the client:\n”);
while ((SSL_get_shutdown(ssl) & SSL_RECEIVED_SHUTDOWN)
       != SSL_RECEIVED_SHUTDOWN) {
    int nbytes_read =
        BIO_get_line(ssl_bio, in_buf, BUF_SIZE);
    if (nbytes_read <= 0) {
        int ssl_error = SSL_get_error(ssl, nbytes_read);
        if (ssl_error == SSL_ERROR_ZERO_RETURN)
                break;
```

```
    if (error_stream)
        fprintf(
            error_stream,
            "Error %i while reading data"
            " from the client\n",
            ssl_error);
        goto failure;
    }
    fwrite(in_buf, 1, nbytes_read, stdout);
    if (!strcmp(in_buf, "\r\n") || !strcmp(in_buf, "\n"))
            break;
}
printf("*** Receiving from the client finished\n");
```

4. Прочитав запрос клиента, отправим ответ сервера:

```
const char* response =
    "HTTP/1.0 200 OK\r\n"
    "Content-type: text/plain\r\n"
    "Connection: close\r\n"
    "Server: Example TLS server\r\n"
    "\r\n"
    "Hello from the TLS server!\n";
int response_length = strlen(response);
printf("*** Sending to the client:\n");
printf("%s", response);
int nbytes_written =
    BIO_write(ssl_bio, response, response_length);
if (nbytes_written != response_length) {
    if (error_stream)
        fprintf(
            error_stream,
            "Could not send all data to the client\n");
    goto failure;
}
printf("*** Sending to the client finished\n");
```

5. После того как ответ отправлен клиенту и больше отправлять нечего, рекомендуется разомкнуть нашу сторону TLS-подключения:

```
BIO_ssl_shutdown(ssl_bio);
```

6. По завершении взаимодействия мы должны освободить память, занятую объектами и буферами:

```
if (ssl_bio)
    BIO_free_all(ssl_bio);
free(in_buf);
```

7. Наконец, проверим и очистим очередь ошибок OpenSSL. Отметим, что очередь ошибок обрабатывается для каждого входящего подключения, а не для приложения в целом:

```
if (ERR_peek_error()) {
    exit_code = 1;
    if (error_stream) {
        fprintf(
            error_stream,
            "Errors from the OpenSSL error queue:\n");
        ERR_print_errors_fp(error_stream);
    }
    ERR_clear_error();
}
```

На этом функция `handle_accepted_connection()` завершается. Управление возвращается циклу `while()`, который принимает запросы на входящие подключения. Теперь можно принять и обработать следующее подключение.

Полный исходный код программы `tls-server` находится на GitHub по адресу https://github.com/PacktPublishing/Demystifying-Cryptography-with-OpenSSL-3/blob/main/Chapter09/tls-server.c.

Выполнение программы tls-server

1. Откройте окно терминала и запустите программу `tls-server`:

```
$ ./tls-server 4433 server_keypair.pem server_cert.pem
*** Listening on port 4433
```

   Теперь сервер работает и готов принимать запросы на подключение.

2. Откройте другое окно терминала и запустите ранее написанную программу `tls-client`, которая будет обращаться к `tls-server` как к серверу:

```
$ ./tls-client localhost 4433 ca_cert.pem
*** Sending to the server:
GET / HTTP/1.1
Host: localhost
Connection: close
User-Agent: Example TLS client
*** Sending to the server finished
*** Receiving from the server:
HTTP/1.0 200 OK
Content-type: text/plain
Connection: close
Server: Example TLS server
Hello from the TLS server!
*** Receiving from the server finished
TLS communication succeeded
```

   Легко видеть, что с точки зрения `tls-client` с TLS-подключением не возникло никаких проблем. Отметим, что на этот раз мы задали третий аргумент `tls-client` – имя файла, содержащего сертификат УЦ, чтобы `tls-client` могла проверить сертификат сервера.

3. Перейдите в первое окно и проверьте, что напечатала `tls-server`:

   ```
   *** Receiving from the client:
   GET / HTTP/1.1
   Host: localhost
   Connection: close
   User-Agent: Example TLS client
   *** Receiving from the client finished
   *** Sending to the client:
   HTTP/1.0 200 OK
   Content-type: text/plain
   Connection: close
   Server: Example TLS server
   Hello from the TLS server!
   *** Sending to the client finished
   *** Listening on port 4433
   ```

   Как видим, `tls-server` успешно обработала подключение от `tls-client` и продолжила прослушивать запросы на последующие подключения.
4. Попробуем еще одну вещь – запуск другого TLS-клиента, утилиты `curl`. Запустите во втором окне терминала следующую команду:

   ```
   $ curl https://localhost:4433 --cacert ca_cert.pem
   Hello from the TLS server!
   ```

   Как видим, `tls-server` совместима и с `curl`! Вывод `tls-server` в первом окне подтверждает, что она действительно обслужила `curl`. Взгляните на заголовок запроса `User-Agent`:

   ```
   *** Receiving from the client:
   GET / HTTP/1.1
   Host: localhost:4433
   User-Agent: curl/7.74.0
   Accept: */*
   *** Receiving from the client finished
   *** Sending to the client:
   HTTP/1.0 200 OK
   Content-type: text/plain
   Connection: close
   Server: Example TLS server
   Hello from the TLS server!
   *** Sending to the client finished
   *** Listening on port 4433
   ```

   Программа `tls-server` работает и принимает подключения, пока ее не снимут. Это можно сделать, например, нажав *Ctrl+C* в том окне терминала, где она запущена.

Мы закончили практическую часть. Мы научились создавать TLS-подключения и использовать объекты BIO и написали две программы, в которых эти знания используются.

Резюме

В этой главе мы узнали о важности протокола TLS, зачем он нужен и где используется. Мы также узнали, как протокол работает на верхнем уровне, что включено в состав процедуры квитирования и что происходит после квитирования. Теоретическую часть мы закончили кратким знакомством с историей протокола TLS, поговорили о его развитии и о том, чем отличаются старые и новые версии SSL и TLS.

В практической части главы мы научились создавать клиентские и серверные TLS-подключения, в командной строке и из программы на C. Мы также узнали о базовых объектах ввода-вывода в OpenSSL.

В следующей главе речь пойдет о более продвинутом использовании сертификатов X.509 в TLS, в частности о проверке сертификатов, управляемой пользователем, и о клиентских сертификатах в TLS.

Глава 10

Использование сертификатов X.509 в TLS

В главе 8 мы узнали о **сертификатах X.509**, а в главе 9 – о **безопасном протоколе транспортного уровня** (**TLS**) и о том, почему в нем важны сертификаты. Мы также рассмотрели несколько очевидных, но очень популярных применений сертификатов в TLS, в частности проверку сертификата сервера, выполняемую OpenSSL по умолчанию, проверку имени сервера и использование сертификата при приеме запроса на подключение к TLS-серверу.

В этой главе мы продолжим разговор об использовании сертификатов X.509 в TLS и рассмотрим более сложные сценарии. Рассматриваются следующие темы:

- специальная проверка сертификата другой стороны в программах на C;
- использование списков отозванных сертификатов в программах на C;
- протокол онлайновой проверки состояния сертификата;
- использование клиентских сертификатов в TLS.

Технические требования

В этой главе встречаются команды, запускаемые из командной строки, и C-код, который можно откомпилировать и выполнить. Для запуска команд понадобятся программа `openssl` и соответствующие динамические библиотеки OpenSSL. Для сборки кода на C будут нужны динамические или статические библиотеки OpenSSL, заголовки библиотек, компилятор C и компоновщик.

Мы реализуем демонстрационные программы, чтобы применить полученные знания на практике. Их полный исходный код находится по адресу https://github.com/PacktPublishing/Demystifying-Cryptography-with-OpenSSL-3/tree/main/Chapter10.

Специальная проверка сертификата другой стороны в программах на C

Любое TLS-подключение устанавливается между двумя сторонами: клиентом и сервером. Каждая сторона может запрашивать и проверять сертификат другой стороны. На практике сертификат сервера почти всегда проверяется в процессе TLS-квитирования. До версии TLS 1.3 протокол TLS поддерживал анонимные шифры, позволявшие серверу работать без сертификата. Но использовались они редко и по умолчанию были запрещены. Таким образом, на практике при работе по протоколу TLS сертификат всегда был обязателен. А вот сертификаты TLS-клиентов, напротив, использовались редко. Однако проверка сертификата клиента в приложении, работающем с OpenSSL, очень похожа на проверку сертификата сервера. Поэтому имеет смысл говорить о проверке сертификата другой стороны и не ограничиваться проверкой сертификата сервера. В большинстве примеров кода мы будем проверять сертификат сервера. Дополнительные сведения о проверке клиентских сертификатов будут приведены ниже в разделе «Запрос и проверка сертификата TLS-клиента на стороне сервера из программы».

Проверить сертификат другой стороны протокола TLS в OpenSSL можно несколькими способами:

- воспользоваться встроенным в OpenSSL кодом проверки сертификата, вызвав функцию `SSL_CTX_set_verify(ctx, SSL_VERIFY_PEER, NULL)` или `SSL_set_verify(ctx, SSL_VERIFY_PEER, NULL)`, как мы делали в программе `tls-client` в главе 9. Встроенную процедуру можно настроить, включив некоторые дополнительные проверки, например проверку имени сервера с помощью функции `SSL_set1_host()` или проверку назначения сертификата с помощью функции `SSL_CTX_set_purpose()` либо `SSL_set_purpose()`;
- сочетать встроенную в OpenSSL проверку с собственной функцией обратного вызова, для чего следует вызвать функцию `SSL_CTX_set_verify(ctx, SSL_VERIFY_PEER, verify_callback)` или `SSL_set_verify(ctx, SSL_VERIFY_PEER, verify_callback)`. Отметим, что последний аргумент функции – не `NULL`, а указатель на функцию обратного вызова, осуществляющую проверку;
- воспользоваться функцией «большой» проверки сертификата, замещающей встроенную в OpenSSL проверку, для чего вызвать функцию `SSL_CTX_set_cert_verify_callback(ctx, cert_verify_callback, arg)`;
- получить и проверить сертификат другой стороны сразу после завершения квитирования. Это позволяют сделать функции `SSL_get_peer_certificate()`, `SSL_get_peer_cert_chain()` и `SSL_get0_verified_chain()`. Можно форсировать завершение квитирования, вызвав функцию `SSL_CTX_set_verify()` или `SSL_set_verify()` с флагом `SSL_VERIFY_NONE` или всегда возвращая код успеха из функций обратного вызова.

В этом разделе мы напишем небольшую программу, демонстрирующую использование обратного вызова «малой» проверки, который можно задать с помощью одной из функций `SSL_CTX_set_verify()` или `SSL_set_verify()`.

Ниже перечислены страницы руководства для функций, которые мы собираемся использовать:

```
$ man SSL_CTX_set_verify
$ man SSL_get_app_data
$ man X509_STORE_CTX_get_ex_data
$ man X509_STORE_CTX_get_error
$ man X509_STORE_CTX_get_current_cert
$ man X509_get_subject_name
$ man X509_NAME_print_ex
$ man BIO_s_mem
```

На странице руководства по `SSL_CTX_set_verify()` написано, что функция обратного вызова для проверки должна иметь такую сигнатуру:

```
int SSL_verify_cb(
    int preverify_ok, X509_STORE_CTX* x509_store_ctx);
```

Эта функция принимает следующие параметры:

- `preverify_ok`: индикатор ошибки;
- `x509_store_ctx`: контекст проверки сертификата X.509.

Функция обратного вызова будет вызываться по меньшей мере один раз для каждого сертификата в цепочке сертификатов противоположной стороны. Первый раз она вызывается для последнего сертификата в цепочке, т. е. для сертификата корневого УЦ, если цепочку сертификатов можно довести до конца. Затем цепочка перебирается от конца к началу, и функция обратного вызова вызывается для всех промежуточных сертификатов и, наконец, для сертификата другой стороны.

В библиотеке OpenSSL есть понятие **глубины сертификата**. Это число, показывающее, как далеко данный сертификат отстоит от сертификата другой стороны в цепочке. Глубина самого сертификата другой стороны всегда равна 0. Глубина его издателя равна 1, глубина издателя издателя – 2 и т. д. Есть также понятие **глубины проверки**; это глубина (или, если хотите, длина) всей цепочки сертификатов. Максимальную глубину проверки можно ограничить с помощью любой из функций `SSL_CTX_set_verify_depth()` или `SSL_set_verify_depth()`.

При вызове функции обратного вызова для проверки параметр `x509_store_ctx` будет содержать информацию о проверяемом в данный момент сертификате из цепочки. Получить сам текущий сертификат позволяет функция `X509_STORE_CTX_get_current_cert()`, а его глубину – функция `X509_STORE_CTX_get_error_depth()`.

Параметр `preverify_ok` функции обратного вызова, который может принимать значение 0 или 1, говорит, обнаружила ли OpenSSL какую-нибудь проблему в текущем сертификате. Если `preverify_ok` равен `1`, то никаких проблем не было, и OpenSSL перейдет к следующему сертификату. Если же `preverify_ok` равен `0`, то OpenSSL обнаружила какую-то ошибку. В таком случае ее код можно получить от функции `X509_STORE_CTX_get_error()`, а строковое представление – от функции `X509_verify_cert_error_string()`. Кроме того, OpenSSL добавит ошибку в очередь ошибок. Как уже отмечалось, важно обрабатывать и очищать очередь ошибок – не забывайте об этом.

Отметим, что OpenSSL может несколько раз вызывать функцию обратного вызова с параметром `preverify_ok=0` для одного сертификата, если в этом сертификате несколько ошибок. Также полезно знать, что если функция обратного вызова вызвана с параметром `preverify_ok=1`, то `X509_STORE_CTX_get_error()` необязательно вернет значение `X509_V_OK`, означающее успех. Если функция обратного вызова раньше вызывалась с `preverify_ok=0`, то `X509_STORE_CTX_get_error()` вернет последнюю известную ошибку. И только если при проверке текущего и предыдущих сертификатов ошибок не было, эта функция вернет `X509_V_OK`.

Функция обратного вызова должна вернуть либо 1 (успех), либо 0 (неудача). Если она вернула `0`, то проверка завершается немедленно, другой стороне посылается тревожное сообщение TLS, содержащее ошибку, и процедура квитирования завершается безуспешно. Если она вернула 1, то проверка продолжается. Если все обратные вызовы возвращают 1, то считается, что проверка прошла успешно.

Если в качестве функции обратного вызова с помощью `SSL_CTX_set_verify()` или `SSL_set_verify()` задан `NULL`, то используется функция проверки по умолчанию, которая всегда возвращает `preverify_ok`.

Отметим, что функция обратного вызова для проверки не получает аргумент типа `void*`, указывающий на пользовательские данные, в отличие от многих других обратных вызовов. Таким образом, если мы хотим избежать использования глобальных переменных – а это рекомендуемая практика, – то использовать при проверке данные из других частей приложения не так-то просто. Но возможно. По счастью, мы можем получить пользовательские данные через параметр `x509_store_ctx`, указывающий на контекст проверки. Из `x509_store_ctx` можно получить указатель типа `SSL*`, а затем взять пользовательские данные из объекта `SSL`. В примере кода ниже показано, как это делается.

Теперь напишем программу `verify-callback`, демонстрирующую использование функции обратного вызова для проверки. Эта программа подключается к TLS-серверу, инициализирует TLS-квитирование, печатает информацию о проверке цепочки сертификатов другой стороны с помощью функции обратного вызова и отключается. В основу будет положена программа `tls-client` из главы 9.

Программа `verify-callback` будет принимать в командной строке те же аргументы, что `tls-client`:

1) имя сервера;
2) порт сервера;
3) факультативное имя файла, содержащего один или несколько сертификатов надежных УЦ для проверки сертификата сервера.

Регистрация функции обратного вызова для проверки

В нашей программе `verify-callback` будет использоваться функция обратного вызова со следующей сигнатурой:

```
int verify_callback(
    int preverify_ok, X509_STORE_CTX* x509_store_ctx);
```

В программе tls-client была такая строка кода, которая разрешала процесс проверки сертификата другой стороны:

```
SSL_CTX_set_verify(ctx, SSL_VERIFY_PEER, NULL);
```

Мы заменим ее строкой, устанавливающей функцию обратного вызова в объекте SSL_CTX:

```
SSL_CTX_set_verify(ctx, SSL_VERIFY_PEER, verify_callback);
```

Мы также подготовим пользовательские данные для обратного вызова проверки. Но эти данные будут находиться не в объекте SSL_CTX, а в объекте SSL. Так будет проще получить пользовательские данные в коде функции verify_callback. Мы можем использовать в качестве пользовательских данных любой указатель. В этом примере такими данными будет указатель FILE* error_stream, т. е. описатель файла для печати ошибок и диагностической информации. В следующей строке показано, как задаются пользовательские данные:

```
SSL_set_app_data(ssl, error_stream);
```

SSL_set_app_data() – это макрос для вызова функции SSL_set_ex_data(), являющейся членом семейства функций *_set_ex_data(). С помощью таких функций можно установить один или несколько указателей на **дополнительные данные** в объектах типа SSL, BIO, X509 и др. Подробнее о них можно прочитать на страницах руководства SSL_set_app_data и CRYPTO_set_ex_data. Лично мне API *_ex_data кажется неудобным и запутанным, поэтому я рекомендую пользоваться им как можно реже. Если вам нужно задать несколько элементов пользовательских данных в объекте SSL, создайте структуру, поместите в нее данные и установите указатель на структуру в объекте SSL, воспользовавшись функцией SSL_set_app_data(), как мы поступили в этом примере.

Нас интересует проверка сертификата, поэтому мы не станем обмениваться с сервером запросами и ответами. Вместо этого мы закроем подключение сразу после успешного или неудачного квитирования. Поэтому код отправки HTTP-запросов и получения ответов на них, унаследованный от программы tls-client, будет удален.

Реализация функции обратного вызова для проверки

1. Наша функция обратного вызова будет просто печатать диагностическую информацию о параметрах вызова и текущем сертификате. Эта информация будет выводиться в поток FILE*, заданный в качестве пользовательских данных в объекте SSL.

Начнем реализацию со строки сигнатуры:

```
int verify_callback(
    int preverify_ok, X509_STORE_CTX* x509_store_ctx) {
```

2. Первым делом получим пользовательские данные:

```
int ssl_ex_data_idx = SSL_get_ex_data_X509_STORE_CTX_idx();
SSL* ssl = X509_STORE_CTX_get_ex_data(
    x509_store_ctx, ssl_ex_data_idx);
FILE* error_stream = SSL_get_app_data(ssl);
```

Для получения пользовательских данных (в данном случае – потока `FILE*`) нам нужен объект `SSL`. Получить его можно как дополнительные данные с определенным индексом из параметра `x509_store_ctx parameter`. Сначала нужно получить этот индекс, потом указатель `SSL*` и уже потом указатель на пользовательские данные `FILE* error_stream`.

3. Затем получим глубину текущего сертификата и текущую ошибку, если таковая имеется:

```
int depth = X509_STORE_CTX_get_error_depth(x509_store_ctx);
int error_code = preverify_ok ?
    X509_V_OK :
    X509_STORE_CTX_get_error(x509_store_ctx);
const char* error_string =
    X509_verify_cert_error_string(error_code);
```

4. Далее получаем текущий сертификат и его поле Subject:

```
X509* current_cert =
    X509_STORE_CTX_get_current_cert(x509_store_ctx);
X509_NAME* current_cert_subject =
    X509_get_subject_name(current_cert);
```

Здесь Subject – объект типа `X509_NAME`.

5. Теперь получим текстовое представление объекта `X509_NAME`, напечатав его в объект BIO памяти:

```
BIO* mem_bio = BIO_new(BIO_s_mem());
X509_NAME_print_ex(
    mem_bio,
    current_cert_subject,
    0,
    XN_FLAG_ONELINE & ~ASN1_STRFLGS_ESC_MSB);
```

6. Далее получим указатель на текст, помещенный в BIO памяти, и его длину:

```
char* bio_data = NULL;
long bio_data_len = BIO_get_mem_data(mem_bio, &bio_data);
```

Отметим, что текст не завершается нулем и может содержать нулевые символы, потому-то знать его длину так важно.

7. Теперь у нас достаточно данных для печати диагностического сообщения, выведем его в `error_stream`:

```
fprintf(
    error_stream,
    "verify_callback() called with depth=%i, "
    "preverify_ok=%i, error_code=%i, error_string=%s\n",
    depth,
    preverify_ok,
    error_code,
    error_string);
fprintf(error_stream, "Certificate Subject: ");
fwrite(bio_data, 1, bio_data_len, error_stream);
fprintf(error_stream, "\n");
```

   Можно было бы напечатать объект `X509_NAME`, содержащий субъект сертификата, прямо в `error_stream`, воспользовавшись функцией `X509_NAME_print_ex_fp()`. Я этого не сделал, потому что хотел продемонстрировать использование BIO памяти и получить текстовое представление объекта `X509_NAME`. В настоящих программах часто бывает необходимо помещать текст в память и как-то обрабатывать его, а не просто выводить в поток `FILE*`.
8. Наша функция `verify_callback` почти готова. Осталось освободить выделенную память:

```
BIO_free(mem_bio);
X509_free(current_cert);
```

   Отметим, что мы не освободили объект `X509_NAME`, содержащий субъект сертификата, поскольку этим объектом владеет объект `X509`, содержащий сам сертификат, поэтому `X509_NAME` будет освобожден при освобождении `X509`. В общем случае трудно предсказать, какие объекты OpenSSL вы должны освобождать, а какие – нет. Обычно эта информация имеется в документации по OpenSSL, но так бывает не всегда. Если в документации ответа нет, то приходится искать его другими способами, например в исходном коде OpenSSL, путем экспериментов или спросив у местного эксперта.
9. Освободив память, мы должны вернуть код завершения. Проще всего вернуть `preverify_ok`:

```
return preverify_ok;
}
```

   Теперь вы знаете, как написать и использовать функцию обратного вызова для проверки сертификата.

Полный исходный код программы `verify-callback` находится на GitHub по адресу: https://github.com/PacktPublishing/Demystifying-Cryptography-with-OpenSSL-3/blob/main/Chapter10/verify-callback.c.

Выполнение программы

Запустим программу для сервера www.example.org, с которым уже работали раньше. Но прежде убедитесь, что OpenSSL может найти на вашем компьютере сертификаты надежных УЦ. О том, как сообщить OpenSSL об их местонахождении, см. в главе 9.

Запускаем программу:

```
$ ./verify-callback www.example.org 443
verify_callback() called with depth=2, preverify_ok=1, error_code=0,
error_string=ok
Certificate Subject: C = US, O = DigiCert Inc, OU = www.digicert.com,
CN = DigiCert Global Root CA
verify_callback() called with depth=1, preverify_ok=1,
error_code=0, error_string=ok
Certificate Subject: C = US, O = DigiCert Inc, CN = DigiCert
TLS RSA SHA256 2020 CA1
verify_callback() called with depth=0, preverify_ok=1, error_code=0,
error_string=ok
Certificate Subject: C = US, ST = California, L = Los Angeles,
O = Internet Corporation for Assigned Names and Numbers, CN = www.example.org
TLS communication succeeded
```

Как видим, OpenSSL построила цепочку проверки сертификата другой стороны и проверила ее, начав с сертификата корневого УЦ и закончив сертификатом сервера. Программа verify-callback напечатала то, что ожидается в случае успеха. А что будет в случае неудачи?

Чтобы смоделировать этот сценарий, мы запустим программу verify-callback для сервера в специальном домене badssl.com. Серверы в этом домене сконфигурированы для тестирования различных ошибок на этапе TLS-квитирования. Перечень серверов с указанием причин ошибок имеется на сайте https://badssl.com/. В примере ниже мы воспользуемся сервером incomplete-chain.badssl.com. Ожидается, что OpenSSL не сможет построить цепочку проверки сертификата другой стороны, потому что не сумеет найти сертификат издателя. Проверим, так ли это на самом деле:

```
$ ./verify-callback incomplete-chain.badssl.com 443
verify_callback() called with depth=0, preverify_ok=0, error_code=20,
error_string=unable to get local issuer certificate
Certificate Subject: C = US, ST = California, L = Walnut Creek,
O = Lucas Garron Torres, CN = *.badssl.com
Could not connect to server incomplete-chain.badssl.com on port 443
Errors from the OpenSSL error queue:
C0C158FD5D7F0000:error:0A000086:SSL routines:tls_post_process_
server_certificate:certificate verify failed:ssl/statem/statem_clnt.c:1882:
C0C158FD5D7F0000:error:0A000197:SSL routines:SSL_shutdown:
shutdown while in init:ssl/ssl_lib.c:2242:
TLS communication failed
```

Ожидания оправдались. Наша функция обратного вызова получила следующие данные:

```
depth=0, preverify_ok=0, error_code=20,
error_string=unable to get local issuer certificate
```

Налицо проблема (`preverify_ok=0`) с сертификатом сервера (`depth=0`). И состоит она в том, что OpenSSL не смогла найти сертификат издателя (`error_code=20`, `error_string=unable to get local issuer certificate`).

Отсюда можно сделать вывод, что программа `verify-callback` работает как в нормальной ситуации, так и при наличии ошибок и может диагностировать проблемы TLS-квитирования с помощью обратного вызова для проверки сертификата.

Ошибки в цепочке сертификатов – не единственная причина возможных отказов TLS-квитирования. Иногда оказывается скомпрометирован закрытый ключ сертификата, из-за чего сертификат отзывают. Состояние отзыва сертификата можно проверить, воспользовавшись **списками отозванных сертификатов** (**CRL**) или **протоколом онлайновой проверки состояния сертификата** (Online Certificate Status Protocol – **OCSP**). В следующем разделе мы поговорим о CRL.

Использование списков отозванных сертификатов в программах на C

CRL – это структура данных, в которой перечислены отозванные сертификаты. Сертификат может быть отозван по разным причинам, в т. ч. компрометация или утрата закрытого ключа, ошибка в сертификате, прекращение деятельности владельца сертификата, замещение одного сертификата другим и т. д.

Списки CRL можно скачать с веб-серверов **удостоверяющих центров** (**УЦ**). Эти списки часто представлены в формате **особых правил кодирования** (Distinguished Encoding Rules – **DER**). Например, CRL для сертификата сайта `www.example.org` можно скачать по адресу http://crl3.digicert.com/DigiCertTLSRSASHA2562020CA1-4.crl.

Скачанный файл CRL можно просмотреть командой `openssl crl`:

```
$ openssl crl \
    -in DigiCertTLSRSASHA2562020CA1-4.crl \
    -inform DER \
    -noout \
    -text \
    | less
```

Будет напечатано:

```
Certificate Revocation List (CRL):
      Version 2 (0x1)
```

```
      Signature Algorithm: sha256WithRSAEncryption
      Issuer: C = US, O = DigiCert Inc,
              CN = DigiCert TLS RSA SHA256 2020 CA1
      Last Update: May 22 07:00:10 2022 GMT
      Next Update: May 29 07:00:10 2022 GMT
      CRL extensions:
          X509v3 Authority Key Identifier:
              ... шестнадцатеричные байты ...
        X509v3 CRL Number:
238
        X509v3 Issuing Distribution Point: critical
              Full Name:
                  URI:http://crl3.digicert.com/ DigiCertTLSRSASHA2562020CA1-4.crl
Revoked Certificates:
    Serial Number: 048A6B01889FA7BEF34AC92DAAC36079
        Revocation Date: Jun 22 00:31:11 2021 GMT
        CRL entry extensions:
            X509v3 CRL Reason Code:
                Key Compromise
    Serial Number: 098AB8A98137F3432A18DE8C1B7F6D2F
        Revocation Date: Jul 3 05:58:20 2021 GMT
        CRL entry extensions:
            X509v3 CRL Reason Code:
                Key Compromise
    ... другие сертификаты ...
Signature Algorithm: sha256WithRSAEncryption
    ... шестнадцатеричные байты ...
```

Как видим, CRL содержит следующую информацию:

- версия формата CRL;
- алгоритм, использованный при подписании CRL;
- издатель CRL в формате **различимого имени** (**DN**);
- временная метка последнего обновления;
- временная метка следующего обновления;
- факультативные расширения CRL;
- список отозванных сертификатов;
- подпись.

CRL должен быть подписан закрытым ключом издателя. Издателем CRL является то же лицо, которое выпустило сертификат. Только издатель сертификата имеет право отозвать сертификат, поместив его в CRL.

Отозванные сертификаты идентифицируются серийными номерами. В записи об отозванном сертификате может быть указана причина отзыва, но это необязательно.

Для программной сверки сертификата с CRL есть два подхода:

- CRL можно поместить в объект `X509_STORE` с помощью функции `X509_STORE_add_crl()`. Это хороший способ, если уже имеется CRL для проверяемого сертификата. Полезно кешировать списки CRL, потому что они могут быть очень велики – десятки мегабайт для крупных УЦ;
- конкретный сертификат можно поискать в CRL с помощью функции обратного вызова, которая задается обращением к `X509_STORE_set_look-`

up_crls(). Этот способ рекомендуется, если заранее неизвестно, какой сертификат проверяется – типичная ситуация для сертификатов серверов в TLS-подключениях. В следующем примере кода мы будем пользоваться этим методом.

Теперь напишем программу crl-check, демонстрирующую использование функции обратного вызова для поиска в CRL. Эта функция должна иметь следующую сигнатуру:

```
STACK_OF(X509_CRL)* X509_STORE_CTX_lookup_crls_fn(
    const X509_STORE_CTX* x509_store_ctx,
    const X509_NAME* x509_name);
```

Функция принимает такие параметры:

- x509_store_ctx: контекст проверки сертификата X.509;
- x509_name: субъект сертификата, состояние отзыва которого должно быть сверено с CRL.

Функция возвращает стек объектов X509_CRL, представляющих списки CRL. В OpenSSL стеки часто используются как списки. Мы увидим, что функция обратного вызова при необходимости может вернуть несколько CRL.

Мы положим в основу crl-check программу verify-callback, описанную выше, и дополним ее код.

Программа crl-check будет принимать те же три аргумента в командной строке, что и verify-callback:

1) имя сервера;
2) порт сервера;
3) факультативное имя файла, содержащего один или несколько сертификатов надежных УЦ для проверки сертификата сервера.

Мы будем использовать кое-какие функции OpenSSL, с которыми раньше не встречались. Ниже перечислены соответствующие им страницы руководства:

```
$ man SSL_CTX_get_cert_store
$ man X509_STORE_set_lookup_crls
$ man X509_STORE_set_flags
$ man X509_NAME_print_ex_fp
$ man X509_get_ext_d2i
$ man ASN1_STRING_get0_data
$ man OSSL_HTTP_get
$ man d2i_X509_CRL_bio
```

Приступим к реализации программы crl-check. Для начала зарегистрируем функцию обратного вызова для поиска в CRL.

Регистрация функции обратного вызова для поиска в CRL

1. В коде нашей программы crl-check будет использоваться функция со следующей сигнатурой:

```
STACK_OF(X509_CRL)* lookup_crls(
    const X509_STORE_CTX* x509_store_ctx,
    const X509_NAME* x509_name);
```

Эту функцию OpenSSL будет вызывать для поиска в CRL.

2. Чтобы гарантировать, что наша функция будет вызвана, мы должны установить ее в объекте X509_STORE с помощью функции X509_STORE_set_lookup_crls() и разрешить сверку с CRL, подняв флаг X509_V_FLAG_CRL_CHECK в том же объекте. Объект X509_STORE исполняет роль хранилища сертификатов для объекта SSL_CTX. В нем можно хранить надежные сертификаты, ненадежные сертификаты, списки CRL, параметры проверки и дополнительные зависящие от приложения данные. Мы можем получить указатель на объект X509_STORE от объекта SSL_CTX.
   Чтобы установить функцию обратного вызова и разрешить проверку CRL, необходимо добавить следующий код в нашу функцию run_tls_client():

```
X509_STORE* x509_store = SSL_CTX_get_cert_store(ctx);
X509_STORE_set_lookup_crls(x509_store, lookup_crls);
X509_STORE_set_flags(x509_store, X509_V_FLAG_CRL_CHECK);
```

Следующий шаг – реализовать функцию обратного вызова для поиска в CRL.

Реализация функции обратного вызова для поиска в CRL

1. Как видно из сигнатуры, функция обратного вызова принимает объект X509_NAME, содержащий субъект сертификата, во втором параметре. Если бы у нас был какой-нибудь кеш CRL, позволяющий получить нужный CRL, зная субъект сертификата, то этот параметр нам бы помог. Но в данном примере такого кеша нет. Поэтому мы получим текущий сертификат из объекта X509_STORE_CTX, который был передан функции обратного вызова в первом параметре, затем получим точки распространения CRL из сертификата и скачаем нужный CRL с одной из этих точек распространения.
   Прежде всего получим поток ошибок так же, как делали это в функции verify_callback():

```
int ssl_ex_data_idx = SSL_get_ex_data_X509_STORE_CTX_idx();
SSL* ssl = X509_STORE_CTX_get_ex_data(
    x509_store_ctx, ssl_ex_data_idx);
FILE* error_stream = SSL_get_app_data(ssl);
```

2. Теперь получим текущий сертификат, для которого нужно проверить состояние отзыва:

```
X509* current_cert =
    X509_STORE_CTX_get_current_cert(x509_store_ctx);
```

3. Далее получим и напечатаем полезную информацию о сертификате:

   ```
   int depth = X509_STORE_CTX_get_error_depth(x509_store_ctx);
   X509_NAME* current_cert_subject =
       X509_get_subject_name(current_cert);
   fprintf(
       error_stream,
       "lookup_crls() called with depth=%i\n",
       depth);
   fprintf(error_stream, "Looking up CRL for certificate: ");
   X509_NAME_print_ex_fp(
       error_stream,
       current_cert_subject,
       0,
       XN_FLAG_ONELINE & ~ASN1_STRFLGS_ESC_MSB);
   fprintf(error_stream, "\n");
   ```

   Заметим, что на этот раз мы воспользовались функцией `X509_NAME_print_ex_fp()`, а не стали печатать субъект сертификата в BIO, как в функции `verify_callback()`.
4. Следующий шаг – получить точки распространения CRL из сертификата. Не в каждом сертификате эти точки определены. Но к счастью, в сертификате сервера `www.example.org`, который мы хотим проверить, они есть, и с них можно скачать CRL. Вот как получить точки распространения CRL:

   ```
   CRL_DIST_POINTS* crl_dist_points =
       (CRL_DIST_POINTS*) X509_get_ext_d2i(
           current_cert,
           NID_crl_distribution_points,
           NULL,
           NULL);
   ```

   `CRL_DIST_POINTS` – это псевдоним типа (typedef) `STACK_OF(DIST_POINT)`, где `DIST_POINT` – структура, описывающая точку распространения CRL.
5. Далее необходимо обойти точки распространения, попытаться скачать CRL с каждой из них и вернуть скачанный CRL:

   ```
   int crl_dist_point_count =
       sk_DIST_POINT_num(crl_dist_points);
   for (int i = 0; i < crl_dist_point_count; i++) {
       DIST_POINT* dist_point =
           sk_DIST_POINT_value(crl_dist_points, i);
       X509_CRL* crl =
           download_crl_from_dist_point(
               dist_point, error_stream);
       if (!crl)
           continue;
       STACK_OF(X509_CRL)* crls = sk_X509_CRL_new_null();
       sk_X509_CRL_push(crls, crl);
       return crls;
   ```

```
    }
    return NULL;
```

Функция lookup_crls() написана. Как видим, она возвращает NULL, если скачать CRL невозможно.

Вы, вероятно, заметили, что для скачивания CRL мы вызывали функцию download_crl_from_dist_point(). Ее тоже нужно реализовать.

Реализация функции скачивания CRL с точки распространения

1. Функция download_crl_from_dist_point() имеет следующую сигнатуру:

```
X509_CRL* download_crl_from_dist_point(
    const DIST_POINT* dist_point, FILE* error_stream);
```

2. Структура, описывающая точку распространения, может содержать несколько строк, называемых общими именами. Получим их:

```
const DIST_POINT_NAME* dist_point_name =
    dist_point->distpoint;
if (!dist_point_name || dist_point_name->type != 0)
    return NULL;
const GENERAL_NAMES* general_names =
    dist_point_name->name.fullname;
if (!general_names)
    return NULL;
```

3. Затем обойдем в цикле общие имена:

```
int general_name_count = sk_GENERAL_NAME_num(general_names);
for (int i = 0; i < general_name_count; i++) {
    const GENERAL_NAME* general_name =
        sk_GENERAL_NAME_value(general_names, i);
    ...
}
```

4. Ниже показано тело цикла. Сначала мы ищем среди общих имен URL-адреса:

```
int general_name_type = 0;
const ASN1_STRING* general_name_asn1_string =
    (const ASN1_STRING*) GENERAL_NAME_get0_value(
        general_name, &general_name_type);
if (general_name_type != GEN_URI)
    continue;
```

5. Если мы нашли общее имя, содержащее URL, получим его текстовое представление:

```
const char* url =
    (const char*) ASN1_STRING_get0_data(
        general_name_asn1_string);
```

6. В этом примере поддерживаются только URL типа HTTP, поэтому все остальные мы пропускаем. Это нормально, потому что точками распространения CRL обычно являются URL-адреса типа HTTP:

```
const char* http_url_prefix = "http://";
size_t http_url_prefix_len = strlen(http_url_prefix);
if (strncmp(url, http_url_prefix, http_url_prefix_len))
    continue;
```

7. Сейчас мы нашли URL-адрес CRL-списка. Попробуем скачать его:

```
fprintf(error_stream, "Found CRL URL: %s\n", url);
X509_CRL* crl = download_crl_from_http_url(url);
```

8. Если попытка скачивания не увенчалась успехом, мы можем попробовать другое общее имя:

```
if (!crl) {
    fprintf(
        error_stream,
        "Failed to download CRL from %s\n", url);
    continue;
}
```

9. Если мы успешно скачали CRL, то вернем его:

```
fprintf(error_stream, "Downloaded CRL from %s\n", url);
return crl;
```

   На этом тело цикла завершается.

Если мы вышли из цикла нормально, значит, не смогли скачать CRL с текущей точки распространения CRL. В таком случае мы вернем NULL:

```
return NULL;
```

Если функция download_crl_from_dist_point() вернула NULL, то вызов функции lookup_crls() вернет следующую точку распространения.

Функция download_crl_from_dist_point() вызывает download_crl_from_http_url(), чтобы скачать CRL с найденного URL-адреса. Рассмотрим ее код.

Реализация функции скачивания CRL с URL-адреса типа HTTP

Функция download_crl_from_http_url() совсем короткая:

```
X509_CRL* download_crl_from_http_url(const char* url) {
    BIO* bio = OSSL_HTTP_get(
```

```
        url,
        NULL /* proxy */,
        NULL /* no_proxy */,
        NULL /* wbio */,
        NULL /* rbio */,
        NULL /* bio_update_fn */,
        NULL /* arg */,
        65536 /* buf_size */,
        NULL /* headers */,
        NULL /* expected_content_type */,
        1 /* expect_asn1 */,
        50 * 1024 * 1024 /* max resp len */,
        60 /* timeout */);
    X509_CRL* crl = d2i_X509_CRL_bio(bio, NULL);
    BIO_free(bio);
    return crl;
}
```

В функции `download_crl_from_http_url()` мы использовали две функции, не встречавшиеся ранее: `OSSL_HTTP_get()` и `d2i_X509_CRL_bio()`. `OSSL_HTTP_get()` – одна из функций HTTP-клиента OpenSSL; это новая функциональность, добавленная в OpenSSL 3.0. Функция `d2i_X509_CRL_bio()` читает CRL в коде DER из BIO и преобразует его в объект `X509_CRL`. В OpenSSL много подобных функций преобразования, например `i2d_X509()`, `d2i_RSAPublicKey_fp()` и др. Имена таких функций состоят из следующих компонентов:

- `d2i` или `i2d` обозначает направление преобразования. `i` означает «internal» – внутренняя C-структура в памяти, а `d` – «DER». Следовательно, `d2i` означает «DER во внутреннее представление», а `i2d` – «внутреннее представление в DER»;
- тип объекта, например `X509_CRL`;
- факультативный суффикс, обозначающий канал ввода или вывода для чтения или записи объекта в коде DER. Суффикс `bio` означает объект BIO, а суффикс `fp` – указатель на файл. Если суффикса нет, то объект в коде DER читается или записывается в память.

Как уже отмечалось, не во всех сертификатах определены точки распространения CRL. Но даже если они определены, скачивание CRL может завершиться неудачно. И что тогда будет? Если флаг `X509_V_FLAG_CRL_CHECK` поднят и OpenSSL не смогла найти CRL, то проверка сертификата на отзыв завершается с ошибкой `X509_V_ERR_UNABLE_TO_GET_CRL`. Зачастую такой исход нежелателен, потому что очень малая доля выпущенных сертификатов отзывается. Если проверка на отзыв не прошла, то с гораздо большей вероятностью сертификат не был отозван. Как избежать ошибки при проверке в случае, когда OpenSSL не смогла получить CRL? Нет никакого флага, который инструктировал бы OpenSSL игнорировать ошибку `X509_V_ERR_UNABLE_TO_GET_CRL`, если флаг `X509_V_FLAG_CRL_CHECK` установлен. Однако все ошибки при проверке обрабатываются функцией обратного вызова, поэтому мы можем попросить OpenSSL игнорировать такую ошибку, вернув 1 из обратного вызова. Для этого нужно только добавить в функцию `verify_callback()` следующие строчки:

```
if (error_code == X509_V_ERR_UNABLE_TO_GET_CRL)
    return 1;
```

И это последние строчки, которые нужно добавить в программу `crl-check`.

Полный исходный код программы `crl-check` находится в GitHub по адресу https://github.com/PacktPublishing/Demystifying-Cryptography-with-OpenSSL-3/blob/main/Chapter10/crl-check.c.

Выполнение программы

Запустим программу, указав тот же сервер `www.example.org`, что и в предыдущем примере.

```
$ ./crl-check example.org 443
* lookup_crls() called with depth=0
  Looking up CRL for certificate: C = US, ST = California, L = Los Angeles,
O = Internet Corporation for Assigned Names and Numbers, CN = www.example.org
  Found CRL URL: http://crl3.digicert.com/ DigiCertTLSRSASHA2562020CA1-4.crl
  Downloaded CRL from http://crl3.digicert.com/ DigiCertTLSRSASHA2562020CA1-4.crl
* verify_callback() called with depth=2, preverify_ok=1,
error_code=0, error_string=ok
  Certificate Subject: C = US, O = DigiCert Inc, OU = www. digicert.com,
CN = DigiCert Global Root CA
* verify_callback() called with depth=1, preverify_ok=1,
error_code=0, error_string=ok
  Certificate Subject: C = US, O = DigiCert Inc, CN = DigiCert
TLS RSA SHA256 2020 CA1
* verify_callback() called with depth=0, preverify_ok=1,
error_code=0, error_string=ok
  Certificate Subject: C = US, ST = California, L = Los Angeles,
O = Internet Corporation for Assigned Names and Numbers, CN = www.example.org
TLS communication succeeded
```

Как видим, наша программа `crl-check` успешно нашла и скачала CRL, так что OpenSSL смогла проверить сертификат на предмет отзыва, сверив его с этим CRL.

Проверим также противоположный случай – когда сертификат отозван. Для этого нужно обратиться к серверу `revoked.badssl.com`:

```
$ ./crl-check revoked.badssl.com 443
* lookup_crls() called with depth=0
  Looking up CRL for certificate: CN = revoked.badssl.com
  Found CRL URL: http://crl3.digicert.com/RapidSSLTLSDVRSAMixedSHA2562020CA-1.crl
  Downloaded CRL from http://crl3.digicert.com/
RapidSSLTLSDVRSAMixedSHA2562020CA-1.crl
* verify_callback() called with depth=0, preverify_ok=0,
error_code=23, error_string=certificate revoked
  Certificate Subject: CN = revoked.badssl.com
Could not connect to server revoked.badssl.com on port 443
Errors from the OpenSSL error queue:
```

```
C081A6A4617F0000:error:0A000086:SSL routines:tls_post_process_
server_certificate:certificate verify failed:ssl/statem/statem_clnt.c:1882:
C081A6A4617F0000:error:0A000197:SSL routines:SSL_shutdown:
shutdown while in init:ssl/ssl_lib.c:2242:
TLS communication failed
```

Как видим, проверка сертификата оказалась безуспешной, потому что сертификат был отозван. Об этом можно судить по тому, что `error_code=23`, а сообщение об ошибке `error_string=certificate revoked`.

Проведенные тесты подтверждают, что программа `crl-check` работает, как задумано, и способна обнаруживать отозванные сертификаты.

Итак, мы научились проверять сертификаты на предмет отзыва с помощью списков CRL. Но это не единственный метод. Существует также способ проверить, отозван ли сертификат, с помощью протокола OCSP. Его мы и рассмотрим в следующем разделе.

Протокол онлайновой проверки состояния сертификата

В этом разделе мы сначала расскажем, что такое протокол онлайновой проверки состояния сертификата, OCSP, и как он работает. А затем научимся его использовать в командной строке и из программы на C.

Что такое протокол онлайновой проверки состояния сертификата

OCSP – более современный метод проверки сертификата на предмет отзыва, позволяющий обойтись гораздо меньшим сетевым трафиком, чем в случае CRL. При использовании OCSP не нужно скачивать большие CRL-файлы. Нужно лишь отправить запрос OCSP-серверу, который называется также **OCSP-ответчиком**, о состоянии интересующего сертификата. OCSP-серверы, как и списки CRL, создаются и обслуживаются издателями сертификатов.

В запросе OCSP-ответчику, представленном в формате ASN.1, клиент указывает список сертификатов, которые нужно проверить на предмет отзыва. OCSP-сервер посылает ответ, тоже в формате ASN.1, содержащий состояния запрошенных сертификатов, срок действия ответа и еще кое-какую информацию. Ответ подписан закрытым ключом издателя сертификатов.

Поскольку OCSP-ответы подписаны, OCSP-серверы обычно работают по незашифрованному протоколу HTTP. Выбор HTTP вместо HTTPS позволяет также избежать циклических OCSP-запросов при проверке сертификата HTTPS-сервера, обслуживающего OCSP.

Иногда OCSP критикуют за недостаточную конфиденциальность. Если TLS-клиент, например браузер, проверяет сертификат каждого TLS-сервера

по OCSP, то OCSP-ответчики знают, на какие сайты заходил браузер. Каждый OCSP-ответчик знает только о сайте, на котором используются сертификаты, выпущенные его владельцем. Однако если кто-нибудь прослушивает интернет-подключение клиента, то сможет прочитать все OCSP-запросы и ответы, отправленные по незашифрованному протоколу HTTP. С другой стороны, прослушивающий и так уже знает IP-адреса серверов, к которым обращался пользователь. Следовательно, он знает о таких популярных сайтах, как Google или Facebook, располагая их IP-адресами. Дополнительно злоумышленник получает информацию о небольших сайтах на общих хостингах, для которых один и тот же IP-адрес может использоваться несколькими сайтами, а также на некоторых блогинговых платформах, где у каждого блога имеется свой домен третьего уровня.

Все проблемы конфиденциальности, серьезные и не очень, могут быть решены с помощью **вшивания OCSP**. Это означает, что OCSP-ответ о сертификате сервера посылается TLS-клиенту TLS-сервером во время процедуры TLS-квитирования, так что TLS-клиенту вообще не нужно контактировать с OCSP-ответчиком. Таким образом, OCSP-ответ вшивается в TLS-квитирование. Вшивание OCSP реализовано с помощью TLS-расширения **Certificate Status Request** (запрос о состоянии сертификата). TLS-клиент должен запросить состояние сертификата во время квитирования, если хочет получить его от TLS-сервера. С другой стороны, сервер должен периодически опрашивать OCSP-ответчик на предмет состояния своего сертификата, чтобы всегда иметь наготове актуальную информацию для передачи клиентам.

OpenSSL поддерживает опрос OCSP-сервера в командной строке и из программы. Сначала научимся проверять сертификаты на отзыв в командной строке.

Использование OCSP в командной строке

Для запросов к OCSP-серверу в командной строке предназначена подкоманда `openssl ocsp`.

1. Прежде всего нам понадобятся сертификаты сервера и его издателя. Их можно получить с помощью команды `openssl s_client`:

   ```
   $ echo | openssl s_client \
       -connect www.example.org:443 \
       -showcerts
   ```

   Обратите внимание на флаг `-showcerts`. Он инструктирует `openssl` выводить на терминал сертификаты, отправленные сервером. Команда порождает длинную распечатку, содержащую сертификаты:

   ```
   Certificate chain
    0 s:C = US, ST = California, L = Los Angeles, O =
   Internet\C2\A0Corporation\C2\A0for\C2\A0Assigned\C2\ A0Names\C2\A0and\C2\A0Numbers,
   CN = www.example.org
      i:C = US, O = DigiCert Inc, CN = DigiCert TLS RSA SHA256 2020 CA1
   ```

```
-----BEGIN CERTIFICATE-----
... символы в коде base64 ...
-----END CERTIFICATE-----
 1 s:C = US, O = DigiCert Inc, CN = DigiCert TLS RSA SHA256 2020 CA1
   i:C = US, O = DigiCert Inc, OU = www.digicert.com,
CN = DigiCert Global Root CA
-----BEGIN CERTIFICATE-----
... символы в коде base64 ...
-----END CERTIFICATE-----
```

2. Теперь мы можем скопировать сертификаты в формате PEM с терминала в файлы. Сохраним первый напечатанный сертификат, для которого `CN = www.example.org`, в файл `www.example.org.cert.pem`, а второй сертификат, с `CN = DigiCert TLS RSA SHA256 2020 CA1`, в файл `DigiCert_Intermediate_CA1.pem`.
3. Сертификаты у нас есть, но как найти URL-адрес OCSP-ответчика? К счастью, сертификаты часто содержат сведения о его URL-адресе в расширении X509v3 **Authority Information Access** (Доступ к информации об УЦ). Сертификат сайта `www.example.org` также содержит это расширение и URL-адрес OCSP-ответчика. Мы можем извлечь этот адрес, задав флаг `-ocsp_uri` в подкоманде `openssl x509`:

```
$ openssl x509 \
    -in www.example.org.cert.pem \
    -noout \
    -ocsp_uri
http://ocsp.digicert.com
```

4. Затем можно проверить сертификат на предмет отзыва по протоколу OCSP:

```
$ openssl ocsp \
    -issuer DigiCert_Intermediate_CA1.pem \
    -cert www.example.org.cert.pem \
    -url http://ocsp.digicert.com
WARNING: no nonce in response
Response verify OK
www.example.org.cert.pem: good
        This Update: Jun 12 05:09:01 2022 GMT
        Next Update: Jun 19 04:24:01 2022 GMT
```

   Как видим, сертификат по-прежнему хороший, т. е. не отозван. Сообщение `Response verify OK` означает, что OCSP-ответ был успешно проверен: он подписан ожидаемым издателем и его срок действия не истек.
5. Добавив в командную строку флаги `-text` и `-resp_text`, мы сможем увидеть текстовые представления OCSP-запроса и ответа. Выполните следующую команду:

```
$ openssl ocsp \
    -issuer DigiCert_Intermediate_CA1.pem \
    -cert www.example.org.cert.pem \
    -url http://ocsp.digicert.com \
```

```
    -text \
    -resp_text
```

Она печатает показанный ниже результат. Первая его часть – OCSP-запрос:

```
OCSP Request Data:
    Version: 1 (0x0)
    Requestor List:
        Certificate ID:
            Hash Algorithm: sha1
            Issuer Name Hash:
                E4E395A229D3D4C1C31FF0980C0B4EC0098AABD8
            Issuer Key Hash:
                B76BA2EAA8AA848C79EAB4DA0F98B2C59576B9F4
            Serial Number:
                0FAA63109307BC3D414892640CCD4D9A
    Request Extensions:
        OCSP Nonce:
            0410C5D14D224CCCEA68EEEC1478533D0DF8
```

Вторая часть – OCSP-ответ и состояние проверки ответа и сертификата:

```
OCSP Response Data:
    OCSP Response Status: successful (0x0)
    Response Type: Basic OCSP Response
    Version: 1 (0x0)
    Responder Id: B76BA2EAA8AA848C79EAB4DA0F98B2C59576B9F4
    Produced At: Jun 12 05:25:33 2022 GMT
    Responses:
    Certificate ID:
        Hash Algorithm: sha1
        Issuer Name Hash:
            E4E395A229D3D4C1C31FF0980C0B4EC0098AABD8
        Issuer Key Hash:
            B76BA2EAA8AA848C79EAB4DA0F98B2C59576B9F4
        Serial Number:
            0FAA63109307BC3D414892640CCD4D9A
    Cert Status: good
    This Update: Jun 12 05:09:01 2022 GMT
    Next Update: Jun 19 04:24:01 2022 GMT
    Signature Algorithm: sha256WithRSAEncryption
        ... шестнадцатеричные байты ...
WARNING: no nonce in response
Response verify OK
www.example.org.cert.pem: good
        This Update: Jun 12 05:09:01 2022 GMT
        Next Update: Jun 19 04:24:01 2022 GMT
```

Как видим, OCSP-запросы и ответы довольны простые. Оба содержат составной идентификатор сертификата и необязательное одноразовое число. Ответ содержит важные дополнительные данные: состояние сертификата, срок действия и подпись.

Проверив состояние отзыва действительного сертификата по OCSP, посмотрим, как обстоит дело с отозванным.

1. Мы уже знаем, что пример отозванного сертификата можно найти на сервере `revoked.badssl.com`. Как и в предыдущем примере, подключимся к серверу и получим цепочку сертификатов сервера без сертификата корневого УЦ:

   ```
   $ echo | openssl s_client \
       -connect revoked.badssl.com:443 \
       -showcerts
   ```

   Сохраним напечатанный на терминале сертификат сервера в файле `revoked.badssl.com.cert.pem`, а сертификат его издателя в файле `RapidSSL_Intermediate_CA1.pem`.
2. Получим URL-адрес OCSP-ответчика:

   ```
   $ openssl x509 \
       -in revoked.badssl.com.cert.pem \
       -noout \
       -ocsp_uri
   http://ocsp.digicert.com
   ```

3. Наконец, проверим состояние отзыва сертификата сервера:

   ```
   $ openssl ocsp \
       -issuer RapidSSL_Intermediate_CA1.pem \
       -cert revoked.badssl.com.cert.pem \
       -url http://ocsp.digicert.com
   WARNING: no nonce in response
   Response verify OK
   revoked.badssl.com.cert.pem: revoked
           This Update: Jun 12 01:09:02 2022 GMT
           Next Update: Jun 19 00:24:02 2022 GMT
           Revocation Time: Oct 27 21:38:48 2021 GMT
   ```

Как видим, теперь сообщается, что сертификат отозван. Полученные результаты подтверждают, что подкоманда `openssl ocsp` умеет получать состояние отзыва как действительного, так и отозванного сертификата.

Итак, мы научились использовать протокол OCSP в командной строке. Теперь посмотрим, как это делается из программы.

Использование OCSP в программах на C

Проверить состояние отзыва сертификата можно и из программы следующими способами:

- отправить явный запрос OCSP-ответчику. Этот метод зависит от возможности получить URL-адрес OCSP-ответчика, например с помощью расширения X509v3 **Authority Information Access** в сертификате сервера;

- использовать вшивание OCSP и соответствующую функцию обратного вызова. Этот метод зависит от поддержки TLS-сервером вшивания OCSP. В программе ниже мы воспользуемся именно этим методом.

Напишем небольшую программу `ocsp-check`, демонстрирующую использование функции обратного вызова для вшивания OCSP. Эта функция должна иметь такую сигнатуру:

```
int ocsp_callback(SSL* ssl, void* arg);
```

Она принимает следующие параметры:

- `ssl`: объект TLS-подключения;
- `arg`: указатель на пользовательские данные.

Функция обратного вызова должна возвращать одно из следующих значений:

- положительное число, если получен хороший вшитый OCSP-ответ;
- `0`, если получен плохой вшитый OCSP-ответ;
- отрицательное число при возникновении ошибки.

Чтобы можно было использовать функцию обратного вызова для вшитого OCSP, необходимо активировать расширение TLS Certificate Status Request (запрос состояния сертификата), вызвав функцию `SSL_CTX_set_tlsext_status_type()`. Функция обратного вызова прописывается в объекте `SSL_CTX` с помощью функции `SSL_CTX_set_tlsext_status_cb()`. Передаваемый ей указатель на пользовательские данные можно установить с помощью функции `SSL_CTX_set_tlsext_status_arg()`.

В основу программы `ocsp-check` мы положим описанную выше программу `verify-callback` и дополним ее код.

Программа `ocsp-check` будет принимать те же три аргумента в командной строке, что и `verify-callback`:

1) имя сервера;
2) порт сервера;
3) факультативное имя файла, содержащего один или несколько сертификатов надежных УЦ для проверки сертификата сервера.

Мы собираемся воспользоваться кое-какими функциями OpenSSL, с которыми раньше не встречались. Ниже перечислены соответствующие им страницы руководства:

```
$ man SSL_CTX_set_tlsext_status_type
$ man OCSP_response_status
$ man SSL_get0_verified_chain
$ man SSL_get_SSL_CTX
$ man SSL_CTX_get_cert_store
$ man OCSP_basic_verify
$ man OCSP_cert_to_id
```

Начнем реализацию `ocsp-check` с регистрации обратного вызова.

Регистрация функции обратного вызова для OCSP

1. Сначала объявим функцию, которая будет вызываться для обработки вшивания OCSP:

```
int ocsp_callback(SSL* ssl, void* arg);
```

2. В функции `run_tls_client()` мы должны активировать TLS-расширение Certificate Status Request и установить обратный вызов для обработки вшивания OCSP:

```
SSL_CTX_set_tlsext_status_type(ctx, TLSEXT_STATUSTYPE_ocsp);
SSL_CTX_set_tlsext_status_cb(ctx, ocsp_callback);
```

   Здесь мы также могли бы задать указатель на пользовательские данные с помощью функции `SSL_CTX_set_tlsext_status_arg()`, но в этом примере он не нужен. Мы и так уже задали пользовательские данные с помощью функции `SSL_set_app_data()`.
   Это все изменения, которые нужно внести для активации обработки вшивания OCSP. Теперь реализуем саму функцию обратного вызова.

Реализация функции обратного вызова для OCSP

1. Прежде всего объявим код возврата по умолчанию:

```
int ocsp_callback(SSL* ssl, void* arg) {
    int exit_code = 1;
```

   В этом примере я хочу оказать наибольшее доверие сертификату сервера. Это означает, что сертификат будет считаться отозванным, только если это можно доказать ответом от OCSP-сервера. Если с ответом что-то не так – например, его не удалось разобрать или он просрочен, – то ответ игнорируется и сертификат сервера считается действительным. Такая тактика оправдана, т. к. вероятность какой-то ошибки при получении TLS-сервером ответа от OCSP-ответчика гораздо выше, чем вероятность отзыва сертификата.
2. Первый параметр, переданный функции обратного вызова для обработки вшивания OCSP, – указатель на объект `SSL`, в котором хранится информация о текущем TLS-подключении. Это означает, что нам не нужно получать объект `SSL` таким же хитрым образом, как в функции `verify_callback()`. Объект `SSL` уже доступен, и мы легко можем получить от него пользовательские данные:

```
FILE* error_stream = SSL_get_app_data(ssl);
```

3. Следующий шаг очень важен – это получение OCSP-ответа:

```
const unsigned char* resp = NULL;
long resp_len = SSL_get_tlsext_status_ocsp_resp(ssl, &resp);
```

```
if (resp_len <= 0 || !resp) {
    if (error_stream)
        fprintf(
            error_stream,
            "* ocsp_callback() called "
            "without OCSP response\n");
    goto cleanup;
}
```

Здесь мы получаем OCSP-ответ в формате DER.

4. Далее нужно декодировать ответ и представить его в виде объекта `OCSP_RESPONSE`:

```
OCSP_RESPONSE* ocsp_response =
    d2i_OCSP_RESPONSE(NULL, &resp, resp_len);
if (!ocsp_response) {
    if (error_stream)
        fprintf(
            error_stream,
            "* ocsp_callback() could not decode "
            "OCSP response\n");
    goto cleanup;
}
```

5. Следующий шаг – печать декодированного OCSP-ответа. В версии OpenSSL 3.0 нет функции, которая печатала бы OCSP-ответ в поток `FILE`. Есть только функция `OCSP_RESPONSE_print()`, которая печатает ответ в объект BIO. Поэтому придется использовать файловый BIO:

```
if (error_stream) {
    BIO* bio = BIO_new_fp(error_stream, BIO_NOCLOSE);
    assert(bio);
    fprintf(
        error_stream,
        "* ocsp_callback() called "
        "with the following OCSP response:\n");
    fprintf(error_stream, " -----\n ");
    OCSP_RESPONSE_print(bio, ocsp_response, 0);
    fprintf(error_stream, " -----\n");
    BIO_free(bio);
}
```

Отметим, что при создании BIO мы указали флаг `BIO_NOCLOSE`. Это очень важно. Благодаря этому флагу следующий вызов `BIO_free()` не будет пытаться закрыть поток `FILE` при освобождении BIO.

6. После этого нужно проверить состояние OCSP-ответа в целом:

```
int res = OCSP_response_status(ocsp_response);
if (res != OCSP_RESPONSE_STATUS_SUCCESSFUL) {
    if (error_stream)
        fprintf(
```

```
        error_stream,
        “OCSP response status is not successful\n”);
    goto cleanup;
}
```

Состояние OCSP-ответа говорит нам, смог ли OCSP-ответчик успешно обработать запрос и сформировать ответ.

7. Далее проверяем подпись OCSP-ответа:

```
OCSP_BASICRESP* ocsp_basicresp =
    OCSP_response_get1_basic(ocsp_response);
STACK_OF(X509)* verified_chain =
    SSL_get0_verified_chain(ssl);
SSL_CTX* ctx = SSL_get_SSL_CTX(ssl);
X509_STORE* x509_store = SSL_CTX_get_cert_store(ctx);
res = OCSP_basic_verify(
    ocsp_basicresp, verified_chain, x509_store, 0);
if (res != 1) {
    if (error_stream)
        fprintf(
            error_stream,
            “OCSP response verification failed\n”);
    goto cleanup;
}
```

Обратите внимание на вызов функции `SSL_get0_verified_chain()`. Ее следует вызывать только после построения цепочки проверки сертификата сервера, в противном случае она может вернуть частичную или неправильную цепочку. По счастью, функция обратного вызова для обработки вшивания OCSP вызывается после проверки сертификата сервера, поэтому обращение к `SSL_get0_verified_chain()` безопасно.

8. Далее следует найти состояние сертификата TLS-сервера в OCSP-ответе. Этот ответ может содержать сведения об отзыве нескольких сертификатов, поэтому нужно поискать тот, который нас интересует:

```
X509* server_cert = sk_X509_value(verified_chain, 0);
X509* issuer_cert = sk_X509_value(verified_chain, 1);
OCSP_CERTID* server_cert_id =
    OCSP_cert_to_id(NULL, server_cert, issuer_cert);
ASN1_GENERALIZEDTIME* revocation_time = NULL;
ASN1_GENERALIZEDTIME* this_update_time = NULL;
ASN1_GENERALIZEDTIME* next_update_time = NULL;
int revocation_status = V_OCSP_CERTSTATUS_UNKNOWN;
int revocation_reason = OCSP_REVOKED_STATUS_NOSTATUS;
res = OCSP_resp_find_status(
    ocsp_basicresp,
    server_cert_id,
    &revocation_status,
    &revocation_reason,
    &revocation_time,
    &this_update_time,
```

```
        &next_update_time);
if (res != 1) {
    if (error_stream)
        fprintf(
            error_stream,
            "Server certificate status is not found "
            " in the OCSP response\n");
    goto cleanup;
}
```

Обращение к `OCSP_resp_find_status()` дало нам информацию о состоянии отзыва сертификата в момент отправки OCSP-ответа и о том, когда следует запрашивать новый OCSP-ответ для того же сертификата.

9. Проверим, действителен ли еще ответ:

```
res = OCSP_check_validity(
    this_update_time, next_update_time, 300, -1);
if (res != 1) {
    if (error_stream)
        fprintf(
            error_stream,
            "OCSP response is outdated\n");
    goto cleanup;
}
```

10. Итак, OCSP-ответ проверен, теперь проверим состояние отзыва сертификата и изменим переменную `exit_code`, если сертификат был отозван:

```
switch (revocation_status) {
    case V_OCSP_CERTSTATUS_REVOKED:
        if (error_stream)
            fprintf(
                error_stream,
                "Server certificate is revoked\n");
        exit_code = 0;
        break;
    case V_OCSP_CERTSTATUS_GOOD:
        if (error_stream)
            fprintf(
                error_stream,
                "Server certificate is not revoked\n");
        break;
    default:
        if (error_stream)
            fprintf(
                error_stream,
                "Server certificate revocation status "
                "is unknown\n");
}
```

11. И в конце функции обратного вызова освободим уже ненужные объекты и вернем код:

```
cleanup:
    if (server_cert_id)
        OCSP_CERTID_free(server_cert_id);
    if (ocsp_basicresp)
        OCSP_BASICRESP_free(ocsp_basicresp);
    if (ocsp_response)
        OCSP_RESPONSE_free(ocsp_response);
    return exit_code;
}
```

Полный исходный код программы ocsp-check находится на GitHub по адресу https://github.com/PacktPublishing/Demystifying-Cryptography-with-OpenSSL-3/blob/main/Chapter10/ocsp-check.c.

Выполнение программы

Запустим программу ocsp-check следующим образом:

```
$ ./ocsp-check www.example.org 443
* verify_callback() called with depth=2, preverify_ok=1,
error_code=0, error_string=ok
  Certificate Subject: C = US, O = DigiCert Inc, OU = www. digicert.com,
CN = DigiCert Global Root CA
* verify_callback() called with depth=1, preverify_ok=1,
error_code=0, error_string=ok
  Certificate Subject: C = US, O = DigiCert Inc, CN = DigiCert
TLS RSA SHA256 2020 CA1
* verify_callback() called with depth=0, preverify_ok=1,
error_code=0, error_string=ok
  Certificate Subject: C = US, ST = California, L = Los Angeles,
O = Internet Corporation for Assigned Names and Numbers, CN = www.example.org
* ocsp_callback() called with the following OCSP response:
  -----
  OCSP Response Data:
      ...
      Cert Status: good
      ...
  -----
  Server certificate is not revoked
TLS communication succeeded
```

Как видим, наша программа ocsp-check успешно проверяет состояние отзыва сертификата по протоколу OCSP.

Выше мы научились всесторонне проверять сертификат TLS-сервера с помощью специальной функции обратного вызова, списков CRL и по протоколу OCSP. В следующем разделе мы поговорим о еще одном важном аспекте использования сертификатов в TLS: клиентских сертификатах.

Использование клиентских сертификатов в TLS

В этом разделе мы научимся использовать клиентские сертификаты в TLS: генерировать их, упаковывать в контейнеры в соответствии со стандартом криптографии с открытым ключом **PKCS #12**, запрашивать у TLS-серверов и наделять ими TLS-клиентов.

Генерирование клиентских сертификатов TLS

Сертификат для TLS-подключения может предоставлять не только сервер, но и клиент. Однако по протоколу TLS клиент не отправляет сертификат по умолчанию, даже если владеет им. Клиент отправляет свой сертификат, только если TLS-сервер запрашивает его.

Еще одна особенность клиентских сертификатов в TLS – то, что они часто хранятся вместе с закрытым ключом сертификата (точнее, парой ключей) в формате PKCS #12. Это формат файла для хранения нескольких криптографических объектов, например сертификатов X.509 и их пар ключей. Хранимые объекты могут быть зашифрованы симметричным шифром и аутентифицированы **имитовставкой на основе функции хеширования** (**HMAC**). Типичный файл в формате #12, содержащий клиентский сертификат в TLS, защищен паролем и содержит сам сертификат, его пару ключей и сертификаты УЦ, образующие цепочку проверки клиентского сертификата. И хотя обычно файл содержит не только клиентский сертификат, называют его файлом клиентского сертификата.

Расширениями имен файлов в формате PKCS #12 являются `.p12` и `.pfx`. Если вы хотите использовать клиентский сертификат в популярном браузере, например Mozilla Firefox или Google Chrome, то должны «подсунуть» его браузеру в виде контейнерного файла PKCS #12, предпочтительно с расширением `.p12` или `.pfx`. В примере ниже мы прочтем клиентский сертификат из файла PKCS #12.

Сначала посмотрим, как сгенерировать клиентский сертификат и упаковать его в контейнер PKCS #12.

Клиентский сертификат генерируется почти так же, как серверный в главе 9. Мы также повторно воспользуемся сертификатом корневого УЦ и его парой ключей:

```
$ openssl req \
    -newkey ED448 \
    -subj "/CN=Client certificate" \
    -addext "basicConstraints=critical,CA:FALSE" \
    -noenc \
    -keyout client_keypair.pem \
    -out client_csr.pem
$ openssl x509 \
```

```
    -req \
    -in client_csr.pem \
    -copy_extensions copyall \
    -CA ca_cert.pem \
    -CAkey ca_keypair.pem \
    -days 3650 \
    -out client_cert.pem
```

После выполнения этих команд у нас будет клиентский сертификат и его пара ключей, сохраненные в PEM-файлах. В следующем разделе мы научимся упаковывать их в контейнер PKCS #12.

Упаковка клиентских сертификатов в контейнер PKCS #12

Для упаковки сгенерированного клиентского сертификата, его пары ключей и сертификата УЦ в контейнер PKCS #12 можно воспользоваться подкомандой `openssl pkcs12`, документированной на странице руководства `openssl-pkcs12`:

```
$ man openssl-pkcs12
```

Вот как пара ключей и сертификаты упаковываются в контейнер PKCS #12:

```
$ openssl pkcs12 \
    -export \
    -inkey client_keypair.pem \
    -in client_cert.pem \
    -certfile ca_cert.pem \
    -passout 'pass:SuperPa$$w0rd' \
    -out client_cert.p12
```

Отметим, что мы задали пароль для контейнера PKCS #12, `SuperPa$$w0rd`. Не очень безопасно задавать пароль в командной строке, потому что команда может быть сохранена в файле истории, а также видна в списке процессов. В этом примере мы поступили так только ради простоты. Программа `openssl` поддерживает и более безопасные способы задания пароля, например чтение из файла или из `stdin`. Подробнее о задании пароля можно прочитать на следующей странице руководства:

```
$ man openssl-passphrase-options
```

Итак, контейнер PKCS #12 успешно создан. Посмотрим, что внутри:

```
$ openssl pkcs12 \
    -in client_cert.p12 \
    -passin 'pass:SuperPa$$w0rd' \
    -noenc \
    -info
MAC: sha256, Iteration 2048
```

```
MAC length: 32, salt length: 8
PKCS7 Encrypted data: PBES2, PBKDF2, AES-256-CBC, Iteration 2048,
PRF hmacWithSHA256
Certificate bag
Bag Attributes
    localKeyID: 63 8B 00 96 4A 4D B7 E9 AF BD C2 09 A5 4A B8 3D A2 FB 40 85
subject=CN = Client certificate
issuer=CN = Root CA
-----BEGIN CERTIFICATE-----
... данные в коде base64 ...
-----END CERTIFICATE-----
Certificate bag
Bag Attributes: <No Attributes>
subject=CN = Root CA
issuer=CN = Root CA
-----BEGIN CERTIFICATE-----
... данные в коде base64 ...
-----END CERTIFICATE-----
PKCS7 Data
Shrouded Keybag: PBES2, PBKDF2, AES-256-CBC, Iteration 2048, PRF hmacWithSHA256
Bag Attributes
    localKeyID: 63 8B 00 96 4A 4D B7 E9 AF BD C2 09 A5 4A B8 3D A2 FB 40 85
Key Attributes: <No Attributes>
-----BEGIN PRIVATE KEY-----
... данные в коде base64 ...
-----END PRIVATE KEY-----
```

Как видим, созданный контейнер PKCS #12 содержит клиентский сертификат, сертификат УЦ и пару ключей клиентского сертификата. Содержимое контейнера зашифровано шифром AES-256-CBC и аутентифицировано имитовставкой HMAC-SHA256.

Подготовив клиентский сертификат, напишем простенький TLS-сервер, который будет запрашивать и проверять его.

Программный запрос и проверка клиентского сертификата TLS на стороне сервера

В этом разделе мы напишем простой TLS-сервер, который запрашивает и проверяет клиентский сертификат. В основу будет положена программа `tls-server` из главы 9, код которой мы доработаем. Новую программу назовем `tls-server2`.

Программа `tls-server2` будет принимать следующие параметры в командной строке:

1) порт сервера;
2) имя файла, содержащего пару ключей TLS-сервера;
3) имя файла, содержащего цепочку сертификатов TLS-сервера;
4) имя файла, содержащего сертификат надежного УЦ, который может проверить клиентский сертификат.

Для проверки клиентского сертификата необходимо загрузить соответствующий сертификат надежного УЦ и включить проверку сертификата другой стороны.

Проверка клиентского сертификата TLS

1. Прежде всего нужно загрузить сертификат надежного УЦ. Для этого мы получим четвертый аргумент программы и передадим его функции `run_tls_server()`. Добавим следующую строку в функцию `main()`:

   ```
   const char* trusted_cert_fname = argv[4];
   ```

2. Функции `run_tls_server()` необходимо также передать параметр `trusted_cert_fname` и добавить в нее следующие строки после кода, который загружает и проверяет закрытый ключ:

   ```
   err = SSL_CTX_load_verify_locations(
       ctx, trusted_cert_fname, NULL);
   if (err <= 0) {
       if (error_stream)
           fprintf(
               error_stream,
               "Could not load trusted certificates\n");
       goto failure;
   }
   ```

3. Еще нужно изменить следующий код, чтобы запросить клиентский сертификат и разрешить его проверку:

   ```
   SSL_CTX_set_verify(ctx, SSL_VERIFY_PEER, NULL);
   ```

   Флаг `SSL_VERIFY_PEER` инструктирует OpenSSL запросить клиентский сертификат. Отметим, что по умолчанию отсутствие сертификата у клиента не считается ошибкой на стороне сервера. Это поведение можно изменить, подняв флаг `SSL_VERIFY_FAIL_IF_NO_PEER_CERT` вместе с флагом `SSL_VERIFY_PEER`:

   ```
   SSL_CTX_set_verify(
       ctx,
       SSL_VERIFY_PEER | SSL_VERIFY_FAIL_IF_NO_PEER_CERT,
       NULL);
   ```

   Если подняты оба флага `SSL_VERIFY_PEER` и `SSL_VERIFY_FAIL_IF_NO_PEER_CERT` и клиент не предъявил сертификат, то TLS-квитирование завершается безуспешно.

Этого кода достаточно, чтобы запросить клиентский сертификат и проверить его. Однако было бы неплохо распечатать субъект этого сертификата и удостовериться, что TLS-клиент отправил свой сертификат, а TLS-сервер проверил его. Для этого напишем функцию `construct_response()`, которая строит ответ сервера, содержащий информацию о клиентском сертификате. В программе `tls-server2` мы отправим именно его, а не статический ответ, как в оригинальной программе `tls-server`.

Реализация функции построения ответа

Функция будет иметь следующую сигнатуру:

```
BIO* construct_response(SSL* ssl);
```

Параметр `ssl` необходим для получения клиентского сертификата. Функция вернет BIO памяти, содержащий ответ сервера.

Реализуем функцию `construct_response()`.

1. Для начала создадим BIO памяти и поместим в него заголовки ответа сервера:

```
BIO* mem_bio = BIO_new(BIO_s_mem());
const char* response_headers =
    "HTTP/1.0 200 OK\r\n"
    "Content-type: text/plain\r\n"
    "Connection: close\r\n"
    "Server: Example TLS server\r\n"
    "\r\n";
BIO_puts(mem_bio, response_headers);
```

2. Далее получим клиентский сертификат:

```
X509* peer_cert = SSL_get_peer_certificate(ssl);
```

В этом примере флаг `SSL_VERIFY_FAIL_IF_NO_PEER_CERT` не используется, поэтому TLS-клиент может и не предъявлять сертификат. В таком случае указатель `peer_cert pointer` будет равен `NULL`. Об этом мы позаботимся позже.

3. Следующий шаг – поместить в BIO памяти информацию о клиентском сертификате или его отсутствии:

```
if (peer_cert) {
    X509_NAME* peer_cert_subject =
        X509_get_subject_name(peer_cert);
    BIO_puts(
        mem_bio,
        "The TLS client certificate subject:\n");
    X509_NAME_print_ex(
        mem_bio,
        peer_cert_subject,
        0,
        XN_FLAG_ONELINE & ~ASN1_STRFLGS_ESC_MSB);
    BIO_puts(mem_bio, "\n");
    X509_free(peer_cert);
} else {
    BIO_puts(
        mem_bio,
        "The TLS client has not provided a certificate\n");
}
```

4. В завершение функции `construct_response()` вернем BIO памяти:

```
return mem_bio;
```

   Отлично – мы написали функцию `construct_response()`! Будем вызывать ее из `handle_accepted_connection()` вместо кода, который строил статический ответ и вычислял его длину.

5. Следующий код в оригинальной программе `tls-server`:

```
const char* response =
    "HTTP/1.0 200 OK\r\n"
    "Content-type: text/plain\r\n"
    "Connection: close\r\n"
    "Server: Example TLS server\r\n"
    "\r\n"
    "Hello from the TLS server!\n";
int response_length = strlen(response);
```

   необходимо заменить таким:

```
BIO* mem_bio = construct_response(ssl);
char* response = NULL;
long response_length = BIO_get_mem_data(mem_bio, &response);
```

6. Не забудем освободить память, выделенную в новом коде! Для этого добавим следующий код в конец функции `handle_accepted_connection()`:

```
if (mem_bio)
    BIO_free(mem_bio);
```

Это все изменения, которые пришлось внести в код TLS-сервера.

Полный код программы `tls-server2` находится на GitHub по адресу https://github.com/PacktPublishing/Demystifying-Cryptography-with-OpenSSL-3/blob/main/Chapter10/tls-server2.c.

Выполнение программы

1. Мы еще не написали программу TLS-клиента, который предъявляет сертификат. Однако для тестирования можно воспользоваться подкомандой `openssl s_client` или утилитой `curl`. Нам понадобятся два окна терминала – одно для TLS-сервера, другое для TLS-клиента. Запустим программу `tls-server2` в первом окне терминала:

```
$ ./tls-server2 \
    4433 \
    server_keypair.pem \
    server_cert.pem \
    ca_cert.pem
*** Listening on port 4433
```

2. Запустим во втором окне терминала программу `openssl` в качестве TLS-клиента, указав клиентский сертификат:

```
$ openssl s_client \
    -connect localhost:4433 \
    -CAfile ca_cert.pem \
    -key client_keypair.pem \
    -cert client_cert.pem
```

3. После запуска `openssl` подключится к `tls-server2` и выведет кучу информации. Нажмите *Enter*. Будет выведен следующий ответ сервера:

```
HTTP/1.0 200 OK
Content-type: text/plain
Connection: close
Server: Example TLS server
The TLS client certificate subject:
CN = Client certificate
```

4. Заглянем в первое окно терминала, в котором работает `tls-server2`. Мы увидим такой вывод:

```
*** Receiving from the client:
*** Receiving from the client finished
*** Sending to the client:
HTTP/1.0 200 OK
Content-type: text/plain
Connection: close
Server: Example TLS server
The TLS client certificate subject:
CN = Client certificate
*** Sending to the client finished
```

   Как видим, `tls-server2` запросила, получила и успешно проверила клиентский сертификат, а также напечатала его поле Subject.
5. Такой же тест можно провести с помощью утилиты `curl`. Запустим ее во втором окне терминала:

```
$ curl \
    https://localhost:4433 \
    --cacert ca_cert.pem \
    --cert 'client_cert.p12:SuperPa$$w0rd' \
    --cert-type P12
The TLS client certificate subject:
CN = Client certificate
```

   Заметим, что клиентский сертификат был предъявлен `curl` в формате PKCS #12. Как видим, наша программа `tls-server2` может напечатать полезную информацию о клиентском сертификате, если TLS-клиент его предъявил. Но что будет, если клиент не предъявил сертификат?
6. Проверим это, воспользовавшись программой `openssl`. Запустим `openssl s_client` без сертификата:

```
$ openssl s_client \
-connect localhost:4433 \
-CAfile ca_cert.pem
```

7. После подключения и нажатия *Enter* мы получим следующий результат:

```
HTTP/1.0 200 OK
Content-type: text/plain
Connection: close
Server: Example TLS server
The TLS client has not provided a certificate
```

8. Аналогичный результат получается при использовании `curl`:

```
$ curl https://localhost:4433 --cacert ca_cert.pem
The TLS client has not provided a certificate
```

Итак, наша программа `tls-server2` работает, как мы и хотели. Она может запросить, проверить и сообщить о сертификате TLS-клиента, а также корректно обработать его отсутствие.

Программа `tls-server2` будет работать и принимать запросы на подключения, пока ее не снимут. Чтобы снять ее, нажмите *Ctrl+C* в том окне терминала, где она запущена.

В следующем разделе мы научимся загружать сертификат на стороне клиента и использовать его на этапе TLS-подключения.

Программное установление TLS-подключения с клиентским сертификатом

В этом разделе мы напишем небольшую программу TLS-клиента, в которой будет использоваться клиентский сертификат. Мы положим в основу программу `tls-client` из главы и разовьем ее. Новую программу назовем `tls-client2`.

Программа `tls-client2` будет принимать следующие параметры в командной строке:

1) имя сервера;
2) порт сервера;
3) имя файла, содержащего сертификат надежного УЦ для проверки сертификата сервера;
4) имя файла в формате PKCS #12, содержащего сертификат клиента и его пару ключей;
5) пароль для файла в формате PKCS #12 и сертификата клиента.

По сравнению с оригинальной программой `tls-client`, программа `tls-client2` принимает два дополнительных параметра. Теперь все пять параметров обязательны.

Мы собираемся загрузить клиентский сертификат из контейнерного файла в формате PKCS #12. Можно также загрузить клиентский сертификат и его

пару ключей из PEM-файлов, воспользовавшись функциями SSL_CTX_use_certificate_chain_file() и SSL_CTX_use_PrivateKey_file(), как в программах tls-server и tls-server2. Однако я хочу продемонстрировать загрузку из файла PKCS #12, потому что именно так часто распространяются клиентские сертификаты.

Мы будем пользоваться функциями OpenSSL, которые раньше не встречались. Ниже перечислены соответствующие страницы руководства:

```
$ man BIO_new_file
$ man d2i_PKCS12_bio
$ man PKCS12_parse
$ man SSL_CTX_use_cert_and_key
$ man PKCS12_free
```

Изменение кода, унаследованного от программы tls-client

1. Поскольку появилось два дополнительных параметра в командной строке, запомним их в переменных в функции main():

   ```
   const char* client_cert_fname = argv[4];
   const char* client_cert_password = argv[5];
   ```

2. Также нужно передать два новых параметра функции run_tls_client(), которая теперь будет выглядеть следующим образом:

   ```
   int run_tls_client(
       const char* hostname,
       const char* port,
       const char* trusted_cert_fname,
       const char* client_cert_fname,
       const char* client_cert_password,
       FILE* error_stream);
   ```

   Вызов run_tls_client() в main() выглядит так:

   ```
   run_tls_client(
       hostname,
       port,
       trusted_cert_fname,
       client_cert_fname,
       client_cert_password,
       stderr);
   ```

3. Внутри функции run_tls_client() после загрузки надежных сертификатов мы должны добавить обращение к функции load_client_certificate(), которая загрузит клиентский сертификат в объект SSL_CTX:

   ```
   int load_exit_code = load_client_certificate(
       client_cert_fname,
       client_cert_password,
       ctx,
       error_stream);
   ```

```
    if (load_exit_code != 0)
        goto failure;
```

Теперь нужно реализовать функцию `load_client_certificate`.

Загрузка сертификата TLS-клиента

Функция `load_client_certificate` будет иметь следующую сигнатуру:

```
int load_client_certificate(
    const char* client_cert_fname,
    const char* client_cert_password,
    SSL_CTX* ctx,
    FILE* error_stream);
```

Она возвращает 0 в случае успеха и ненулевое значение в случае ошибки.

1. Сначала определим код возврата по умолчанию, который нужно будет изменить в случае ошибки:

   ```
   int exit_code = 0;
   ```

2. Затем нужно создать файловый BIO и открыть файл с клиентским сертификатом для чтения:

   ```
   BIO* client_cert_bio = BIO_new_file(client_cert_fname, “rb”);
   if (!client_cert_bio) {
       if (error_stream)
           fprintf(
               error_stream,
               “Could not open client certificate file %s\n”,
               client_cert_fname);
       goto failure;
   }
   ```

3. Следующий шаг – загрузить файл с клиентским сертификатом в объект `PKCS12`:

   ```
   PKCS12* pkcs12 = d2i_PKCS12_bio(client_cert_bio, NULL);
   if (!pkcs12) {
       if (error_stream)
           fprintf(
               error_stream,
               “Could not load client certificate “
               “from file %s\n”,
               client_cert_fname);
       goto failure;
   }
   ```

   Отметим, что мы могли бы открыть файл функций `fopen()` и загрузить сертификат функцией `d2i_PKCS12_fp()`. Но я хотел продемонстрировать использование BIO еще одного типа – файлового.

4. Функция `d2i_PKCS12_bio()` загрузила содержимое контейнера PKCS #12, но не расшифровала и не разобрала его. Следующий шаг – проверить пароль и убедиться, что мы можем расшифровать контейнер PKCS #12. Этот шаг необязателен, но может быть полезен для отладки:

```
int res = PKCS12_verify_mac(
    pkcs12,
    client_cert_password,
    strlen(client_cert_password));
if (res != 1) {
    if (error_stream)
        fprintf(
            error_stream,
            "Invalid password was provided "
            "for client certificate file %s\n",
            client_cert_fname);
    goto failure;
}
```

5. Проверив пароль контейнера, мы должны попробовать расшифровать его, а затем разобрать содержимое и получить клиентский сертификат, его пару ключей и сертификаты УЦ для цепочки проверки:

```
res = PKCS12_parse(
    pkcs12,
    client_cert_password,
    &pkey,
    &cert,
    &cert_chain);
if (res != 1) {
    if (error_stream)
        fprintf(
            error_stream,
            "Could not decode client certificate "
            "loaded from file %s\n",
            client_cert_fname);
    goto failure;
}
```

6. Если контейнер PKCS #12 разобран успешно, то мы можем прописать клиентский сертификат, его пару ключей и сертификаты УЦ в объекте `SSL_CTX`:

```
res = SSL_CTX_use_cert_and_key(
    ctx,
    cert,
    pkey,
    cert_chain,
    1);
if (res != 1) {
    if (error_stream)
        fprintf(
```

```
            error_stream,
            "Could not use client certificate "
            "loaded from file %s\n",
            client_cert_fname);
    goto failure;
}
```

Отметим, что функция `SSL_CTX_use_cert_and_key()` увеличивает счетчик ссылок на объекты `cert`, `pkey` и `cert_chain`. Поэтому мы должны освободить их в конце функции. Поскольку объект `SSL_CTX` еще хранит ссылки на них, память, выделенная для этих объектов, при этом не освобождается. Это произойдет, когда будет освобожден сам объект `SSL_CTX`.

7. Если говорить о потенциальной отладке, то было бы неплохо проверить, что загруженная пара ключей клиентского сертификата соответствует самому клиентскому сертификату:

```
res = SSL_CTX_check_private_key(ctx);
if (res != 1) {
    if (error_stream)
        fprintf(
            error_stream,
            "Client keypair does not match "
            "client certificate\n");
    goto failure;
}
```

В этот момент мы успешно загрузили клиентский сертификат, его пару ключей и сертификат УЦ, а также прописали все это в объекте `SSL_CTX`.

8. Теперь закончим функцию `load_client_certificate()`, освободив использованные объекты и вернув код завершения:

```
    goto cleanup;
failure:
    exit_code = 1;
cleanup:
    if (cert_chain)
        sk_X509_pop_free(cert_chain, X509_free);
    if (cert)
        X509_free(cert);
    if (pkey)
        EVP_PKEY_free(pkey);
    if (pkcs12)
        PKCS12_free(pkcs12);
    if (client_cert_bio)
        BIO_free(client_cert_bio);
    return exit_code;
}
```

Отметим, что обращение к `BIO_free(client_cert_bio)` закрывает файл, прежде чем освободить память, выделенную объекту файлового BIO.

На этом мы закончили реализацию функции `load_client_certificate()` и всей программы `tls-client2`. Ее полный исходный код находится на GitHub по адресу https://github.com/PacktPublishing/Demystifying-Cryptography-with-OpenSSL-3/blob/main/Chapter10/tls-client2.c.

Выполнение программы

1. Как и в предыдущем примере, необходимо два окна терминала. В первом окне мы запустим программу `tls-server2`:

```
$ ./tls-server2 \
    4433 \
    server_keypair.pem \
    server_cert.pem \
    ca_cert.pem
*** Listening on port 4433
```

2. Во втором окне запустим программу `tls-client2`:

```
$ ./run-tls-client2.sh
*** Sending to the server:
GET / HTTP/1.1
Host: localhost
Connection: close
User-Agent: Example TLS client
*** Sending to the server finished
*** Receiving from the server:
HTTP/1.0 200 OK
Content-type: text/plain
Connection: close2
Server: Example TLS server
The TLS client certificate subject:
CN = Client certificate
*** Receiving from the server finished
TLS communication succeeded
```

   Как видим, наша программа `tls-client2` смогла загрузить клиентский сертификат и использовать его при установлении TLS-подключения. Программа `tls-server2` сообщила, что получила клиентский сертификат на стороне сервера. Отсюда можно сделать вывод, что программа `tls-client2` работает, как задумано.

Резюме

Эта глава отличалась от предыдущих. Ранее мы рассматривали одну крупную тему в каждой главе. Здесь же мы обсудили много мелких тем и написали несколько демонстрационных программ.

Мы узнали о том, как проверять сертификат другой стороны в процедуре TLS-квитирования. Мы также узнали о методах проверки с использованием CRL и OCSP. Затем обратились к сертификатам TLS-клиентов и контейнерам в формате PKCS #12. После этого мы научились запрашивать и проверять клиентский сертификат на стороне сервера, а также загружать и использовать сертификат на стороне клиента. И завершили главу, соединив между собой наши программы TLS-сервера и TLS-клиента, поддерживающие клиентские сертификаты.

Полученные в этой главе знания помогут вам при реализации продвинутой обработки сертификатов сервера и клиента в программах, работающих с TLS.

В следующей главе мы поговорим о более специальных применениях TLS.

Глава 11

Специализированные применения TLS

В главе 9 мы узнали о **безопасном протоколе транспортного уровня** (**TLS**) и о том, как устанавливать TLS-подключения. В этой главе мы узнаем о более продвинутом применении TLS и специальных сценариях.

Будут рассмотрены следующие темы:

- что такое закрепление сертификатов в TLS;
- использование закрепления сертификатов;
- что такое блокирующие и неблокирующие сокеты;
- использование неблокирующих сокетов в TLS;
- что такое TLS на нестандартных сокетах;
- использование TLS на нестандартных сокетах.

Технические требования

В этой главе встречаются команды, запускаемые из командной строки, и C-код, который можно откомпилировать и выполнить. Для запуска команд понадобятся программа `openssl` и соответствующие динамические библиотеки OpenSSL. Для сборки кода на C будут нужны динамические или статические библиотеки OpenSSL, заголовки библиотек, компилятор C и компоновщик.

Мы реализуем демонстрационные программы, чтобы применить полученные знания на практике. Их полный исходный код находится по адресу https://github.com/PacktPublishing/Demystifying-Cryptography-with-OpenSSL-3/tree/main/Chapter11.

Что такое закрепление сертификатов в TLS

Иногда проверка сертификатов производится не с помощью традиционных правил **инфраструктуры открытых ключей** (**PKI**) с применением хранилища сертификатов, надежных сертификатов и цепочек проверки сертификатов. Один из нестандартных методов проверки – закрепление сертификатов (certificate pinning). TLS-клиент *закрепляет* конкретный сертификат за сервером, т. е. ожидает, что этот сервер владеет в точности этим сертификатом. У этой идеи есть несколько вариаций, например закрепление нескольких возможных сертификатов, закрепление открытого ключа сертификата, а не самого сертификата, или закрепление сертификата конкретного издателя. Закрепление сертификата можно использовать как вместе со стандартной проверкой сертификата, так и вместо нее.

Закрепление сертификата – не очень популярный метод проверки сертификатов HTTPS-серверов в публичном интернете. Однако, хотя вы вряд ли задумывались об этом, закрепление сертификата – основной метод проверки открытого ключа в протоколе SSH. Открытый ключ SSH-сервера закрепляется на клиентах в файле `known_hosts`, заранее или при первом подключении. Открытые ключи пользователей SSH закрепляются на сервере в файле `authorized_keys`.

Что касается протокола TLS, закрепление сертификатов используется в некоторых приложениях, взаимодействующих через интернет только с одним сервером. Особенно оно популярно в приложениях для мобильного банкинга.

Почему выбирают закрепление сертификатов? Чтобы ответить на этот вопрос, рассмотрим текущую ситуацию, возникающую в модели PKI, основанной на цепочках сертификатов, надежных сертификатах и хранилищах сертификатов и используемой в большинстве приложений с поддержкой TLS.

В модели PKI по умолчанию, используемой в TLS, сертификат любого сервера разрешено подписывать сертификатом любого **удостоверяющего центра** (**УЦ**) из числа находящихся в хранилище доверенных сертификатов. Типичное хранилище сертификатов операционной системы или браузера содержит более 100 сертификатов надежных УЦ. Следовательно, мы предполагаем, что ни один из них не выпускает подложных сертификатов, которые можно использовать в **атаке с человеком посередине** (**MITM**). Но иногда подложные сертификаты все-таки появляются, потому что УЦ был взломан; такое случалось с УЦ DigiNotar, GlobalSign и Comodo. Бывает также, что УЦ выпускают промежуточные сертификаты для устройств, которые используются правоохранительными органами и могут служить посредниками для MITM-атак. Такие устройства помещаются на стороне интернет-провайдера подозреваемого преступника. Все TLS-подключения подозреваемого проходят через такое устройство, которое организует MITM-атаку против подозреваемого, выпуская в режиме реального времени сертификаты для любого сервера, обращения к которому проходят через устройство. Такой вид MITM-атаки позволяет перехватывать и расшифровывать TLS-подключения подозреваемого.

Существуют также коммерческие MITM-устройства. Они устанавливаются системными администраторами организаций для проверки TLS-трафика на присутствие вирусов и троянов. В таком случае сертификат MITM-устройства выпущен не хорошо известным УЦ, а УЦ самой организации. И организация устанавливает свой УЦ в качестве надежного в хранилища сертификатов на машинах всех работников.

Еще один способ атаковать модель PKI по умолчанию – установить подложный сертификат корневого УЦ в пользовательское хранилище сертификатов. Это можно сделать, взломав компьютер пользователя или воспользовавшись методами социальной инженерии. Следует ли считать три описанных сценария рисками? Каждый решает сам. Но многие банки приняли решение использовать закрепление сертификатов TLS в своих приложениях, чтобы уменьшить риск MITM-атак.

Помимо повышенной безопасности, закрепление сертификатов может принести и некоторую экономическую выгоду. Вместо того чтобы покупать сертификат сервера у коммерческого УЦ, вы можете сгенерировать его самостоятельно. Можно даже использовать самоподписанный сертификат, чтобы сократить затраты на обслуживание. При проверке закрепленного сертификата сервера на стороне TLS-клиента можно опустить многие обычно выполняемые проверки. Если сертификат закреплен, значит, он предварительно проверен и заведомо хороший.

Но у закрепления сертификатов есть и недостатки. Чаще всего, чтобы закрепить сертификаты, клиентскому приложению передаются сами сертификаты или их хеш-значения. Если вы доверяете только одному закрепленному сертификату, переданному приложению, то обновить его можно только путем обновления приложения. Кроме того, в таких случаях необходимо синхронизировать обновление сертификата на сервере с обновлением приложения. Это не очень практично, но у проблемы есть несколько решений. Одно из них – закрепить в приложении несколько допустимых сертификатов сервера, например текущий и следующий планируемый. Если срок действия сертификата – год или более, то имеет смысл попросить пользователей обновлять приложение часто, чтобы обновился также сертификат. Другое решение – реализовать собственный механизм автоматического обновления в самом приложении, чтобы оно забирало следующий сертификат, подключившись к серверу под защитой текущего. Еще одно решение – закрепить сертификат издателя (УЦ) вместо или вместе с сертификатом сервера. Это решение менее безопасно, чем закрепление сертификата сервера, но все же безопаснее полного отсутствия закрепления. Но тогда мы имеем ту же проблему обновления закрепленного сертификата издателя. Нужно обновлять его либо путем обновления всего приложения, либо путем реализации собственного механизма обновления внутри приложения.

Еще один недостаток закрепления сертификатов в TLS в качестве замены PKI по умолчанию – необходимость самостоятельно реализовывать отзыв. Но эта задача вполне посильная. Стоит также помнить, что в процессе MITM-атаки против модели PKI по умолчанию противник может без труда блокировать подключения TLS-клиента к серверам, распространяющим **список отозванных сертификатов** (**CRL**), и серверам **протокола онлайновой про-**

верки состояния сертификата (**OCSP**), а также опустить **вшивание OCSP** на стороне, обращенной к клиенту. Поэтому аргумент, связанный с отзывом, не слишком убедителен.

Итак, закрепление сертификата, используемое в TLS вместо модели PKI по умолчанию, имеет следующие плюсы и минусы.

Плюсы:

- повышенная безопасность – уменьшается риск MITM-атак;
- можно использовать сгенерированный внутри организации сертификат вместо покупки у коммерческого УЦ;
- более удобное обслуживание, потому что нет необходимости создавать **запрос на подписание сертификата** (**CSR**) и общаться с УЦ. Можно даже использовать самоподписанный сертификат.

Минусы:

- увеличение затрат на обслуживание, потому что теперь нужно самостоятельно организовывать обновление закрепленных сертификатов;
- увеличение затрат на обслуживание, потому что теперь нужно самостоятельно организовывать отзыв сертификатов.

В следующем разделе мы научимся реализовывать простое закрепление сертификатов на стороне клиента из программы.

Использование закрепления сертификатов

Чтобы научиться закреплять сертификат в коде на C, мы напишем небольшую программу `tls-cert-pinning`. Мы реализуем простой вариант закрепления: закрепим сертификат всего одного сервера и будем использовать закрепление вместо модели PKI по умолчанию, а не в дополнение к ней.

Мы установим «большой» обратный вызов проверки сертификата с помощью функции `SSL_CTX_set_cert_verify_callback()`, а не «малый» – с помощью функции `SSL_CTX_set_verify()`, – просто чтобы узнать еще об одном типе обратного вызова.

Функция `SSL_CTX_set_cert_verify_callback()` устанавливает функцию обратного вызова, которая должна выполнять всю процедуру проверки сертификата. Подразумеваемая по умолчанию реализация этой функции OpenSSL строит цепочку проверки сертификата, проверяет сигнатуры и срок действия и, среди прочего, вызывает функцию обратного вызова, установленную функцией `SSL_CTX_set_verify()`, если таковая имеется. Важно понимать, что установка обратного вызова с помощью функции `SSL_CTX_set_cert_verify_callback()` замещает весь процесс проверки по умолчанию. Поэтому использовать эту возможность нужно очень осторожно. Неправильная реализация «большого» обратного вызова может повлечь за собой кошмарные последствия. Если не уверены в своих силах, лучше установите «малый» обратный вызов с помощью функции `SSL_CTX_set_verify()`, рассмотренной в главе 10.

Мы считаем, что закрепляемый сертификат уже проверен. Поэтому нам даже не нужно строить цепочку проверки сертификата сервера; наша функ-

ция обратного вызова будет просто проверять, что сертификат сервера совпадает с закрепленным.

Для функции SSL_CTX_set_cert_verify_callback(), конечно же, есть страница руководства:

```
$ man SSL_CTX_set_cert_verify_callback
```

Наша программа tls-cert-pinning будет основана на программе tls-client из главы 9, в исходный код которой будет добавлена функциональность закрепления сертификата.

Программа tls-cert-pinning принимает следующие аргументы в командной строке:

1) имя сервера;
2) порт сервера;
3) закрепленный сертификат сервера.

В отличие от tls-client, в программе tls-cert-pinning третий аргумент обязателен.

Изменение функции run_tls_client()

В этом разделе мы добавим объявление функции обратного вызова для проверки сертификата и внесем изменения в функцию run_tls_client.

1. Сначала объявим функцию обратного вызова для проверки сертификата:

   ```
   int cert_verify_callback(
       X509_STORE_CTX* x509_store_ctx, void* arg);
   ```

   Как видим, она принимает два аргумента:

   - x509_store_ctx – контекст проверки сертификата X.509;
   - arg – указатель типа void* на произвольные пользовательские данные.

   Контекст проверки сертификата создается OpenSSL. Указатель на пользовательские данные устанавливается функцией SSL_CTX_set_cert_verify_callback() тогда же, когда и обратный вызов для проверки сертификата. Функция обратного вызова должна вернуть либо 1 – проверка прошла успешно, либо 0 – произошла ошибка.
2. Следующий шаг – загрузить закрепленный сертификат из файла, указанного в третьем аргументе командной строки. Для этого заменим уже имеющийся в программе tls-client код:

   ```
   const char* trusted_cert_fname = argv[3];
   if (trusted_cert_fname)
       err = SSL_CTX_load_verify_locations(
           ctx, trusted_cert_fname, NULL);
   else
       err = SSL_CTX_set_default_verify_paths(ctx);
   ```

```
if (err <= 0) {
    if (error_stream)
        fprintf(
            error_stream,
            "Could not load trusted certificates\n");
    goto failure;
}
```

следующим:

```
const char* pinned_server_cert_fname = argv[3];
FILE* pinned_server_cert_file = NULL;
X509* pinned_server_cert = NULL;
pinned_server_cert_file = fopen(
    pinned_server_cert_fname, "rb");
if (pinned_server_cert_file)
    pinned_server_cert = PEM_read_X509(
        pinned_server_cert_file, NULL, NULL, NULL);
if (!pinned_server_cert) {
    if (error_stream)
        fprintf(
            error_stream,
            "Could not load pinned server certificate\n");
    goto failure;
}
```

3. Следующий шаг – установить обратный вызов для проверки сертификата:

```
SSL_CTX_set_cert_verify_callback(
    ctx, cert_verify_callback, pinned_server_cert);
```

   Отметим, что `pinned_server_cert` передается функции обратного вызова в виде пользовательских данных.
4. В функции обратного вызова нам необходим еще один указатель, `error_stream`. Можно было бы объединить `pinned_server_cert` и `error_stream` в C-структуре, создать объект такой структуры и сделать указатель на него пользовательскими данными для функции `SSL_CTX_set_cert_verify_callback()`. Но указатель `error_stream` можно передать обратному вызову проще – как было сделано в главе 10 с помощью функции `SSL_set_app_data()`:

```
SSL_set_app_data(ssl, error_stream);
```

5. Еще необходимо освободить некоторые из вновь созданных объектов в конце функции `run_tls_client()`:

```
if (pinned_server_cert)
    X509_free(pinned_server_cert);
if (pinned_server_cert_file)
    fclose(pinned_server_cert_file);
```

Это все, что нужно изменить в функции `run_tls_client()` нашей программы.

Реализация функции cert_verify_callback()

1. Начнем реализацию функции `cert_verify_callback()` с получения указателей `error_stream` и `pinned_server_cert`, которые были переданы функции обратного вызова из `run_tls_client()`:

```
int ssl_ex_data_idx = SSL_get_ex_data_X509_STORE_CTX_idx();
SSL* ssl = X509_STORE_CTX_get_ex_data(
    x509_store_ctx, ssl_ex_data_idx);
FILE* error_stream = SSL_get_app_data(ssl);
X509* pinned_server_cert = arg;
```

2. Следующий шаг – получить фактический сертификат сервера:

```
X509* actual_server_cert =
    X509_STORE_CTX_get0_cert(x509_store_ctx);
```

Заметим, что функция `X509_STORE_CTX_get_current_cert()` теперь не дала бы нам сертификат сервера. В момент вызова функции «большого» обратного вызова в структуре `x509_store_ctx` инициализировано очень мало полей. И текущего сертификата среди них нет. Одна из задач «большого» обратного вызова – установить текущий сертификат, его глубину и состояние проверки.

3. Получив сертификат сервера, напечатаем диагностическую информацию:

```
if (error_stream) {
    X509_NAME* pinned_cert_subject =
        X509_get_subject_name(pinned_server_cert);
    X509_NAME* actual_cert_subject =
        X509_get_subject_name(actual_server_cert);
    fprintf(
        error_stream,
        "cert_verify_callback() called with the following "
        "pinned certificate:\n");
    X509_NAME_print_ex_fp(
        error_stream,
        pinned_cert_subject,
        2,
        XN_FLAG_ONELINE & ~ASN1_STRFLGS_ESC_MSB);
    fprintf(error_stream, "\n");
    fprintf(
        error_stream,
        "The server presented the following certificate:\n");
    X509_NAME_print_ex_fp(
        error_stream,
        actual_cert_subject,
        2,
        XN_FLAG_ONELINE & ~ASN1_STRFLGS_ESC_MSB);
        fprintf(error_stream, "\n");
}
```

4. Следующий шаг – сравнить закрепленный и фактический сертификаты:

```
int cmp = X509_cmp(pinned_server_cert, actual_server_cert);
```

5. Прежде чем проверять результат сравнения и возвращать управление, пропишем текущий сертификат и его глубину в контексте проверки:

```
X509_STORE_CTX_set_current_cert(
    x509_store_ctx, actual_server_cert);
X509_STORE_CTX_set_depth(x509_store_ctx, 0);
```

6. Наконец, проверим результат сравнения и либо вернем код успеха, либо установим код ошибки проверки и вернем код неудачи:

```
if (cmp == 0) {
    if (error_stream)
        fprintf(
            error_stream,
            "The certificates match. "
            "Proceeding with the TLS connection.\n");
    X509_STORE_CTX_set_error(x509_store_ctx, X509_V_OK);
    return 1;
} else {
    if (error_stream)
        fprintf(
            error_stream,
            "The certificates do not match. "
            "Aborting the TLS connection.\n");
    X509_STORE_CTX_set_error(
        x509_store_ctx, X509_V_ERR_APPLICATION_VERIFICATION);
    return 0;
}
```

На этом заканчивается реализация функции `cert_verify_callback()` и программы в целом.

Полный исходный код программы `tls-cert-pinning` находится на GitHub по адресу https://github.com/PacktPublishing/Demystifying-Cryptography-with-OpenSSL-3/blob/main/Chapter11/tls-cert-pinning.c.

Выполнение программы tls-cert-pinning

Для запуска `tls-cert-pinning` нам понадобится программа `tls-server` из главы 9, а также файл с сертификатом сервера, который нужно будет закрепить.

1. Откройте два окна терминала.
2. В первом окне запустите программу `tls-server`:

```
$ ./tls-server 4433 server_keypair.pem server_cert.pem
*** Listening on port 4433
```

3. Во втором окне запустите программу `tls-cert-pinning`:

```
$ ./tls-cert-pinning localhost 4433 server_cert.pem
* cert_verify_callback() called with the following pinned
certificate:
  CN = localhost
  The server presented the following certificate:
  CN = localhost
  The certificates match. Proceeding with the TLS connection.
*** Sending to the server:
GET / HTTP/1.1
Host: localhost
Connection: close
User-Agent: Example TLS client
*** Sending to the server finished
*** Receiving from the server:
HTTP/1.0 200 OK
Content-type: text/plain
Connection: close
Server: Example TLS server
Hello from the TLS server!
*** Receiving from the server finished
TLS communication succeeded
```

   Как видим, сертификат сервера совпадает с закрепленным и TLS-подключение завершилось успешно.

4. Проверим также, что происходит, когда сертификаты не совпадают:

```
$ ./tls-cert-pinning www.example.org 443 server_cert.pem
* cert_verify_callback() called with the following pinned certificate:
  CN = localhost
  The server presented the following certificate:
  C = US, ST = California, L = Los Angeles, O = Internet Corporation for Assigned
Names and Numbers, CN = www. example.org
  The certificates do not match. Aborting the TLS connection.
Could not connect to server www.example.org on port 443
Errors from the OpenSSL error queue:
4017BA4F1B7F0000:error:0A000086
:SSL routines:tls_post_process_server_certificate
:certificate verify failed
:../ssl/statem/statem_clnt.c:1883:
4017BA4F1B7F0000:error:0A000197
:SSL routines:SSL_shutdown
:shutdown while In init
:../ssl/ssl_lib.c:2244:
TLS communication failed
```

   Как видим, на этот раз сертификат сервера не совпал с закрепленным сертификатом, поэтому TLS-подключение было прервано. Мы можем сделать вывод, что наша простая реализация закрепляемого сертификата работает правильно.

Закрепление сертификатов было одним из продвинутых сценариев использования TLS. В следующих двух разделах мы узнаем еще об одном: использовании TLS-подключений на неблокирующих сокетах. Неблокирующие сокеты позволяют сделать приложение более отзывчивым или обрабатывать несколько подключений одновременно.

Что такое блокирующие и неблокирующие сокеты

Сетевые подключения можно создавать на **блокирующих** и **неблокирующих сокетах**. Режим по умолчанию зависит от ОС, но в большинстве ОС подразумевается блокирующий режим.

В блокирующем режиме, когда программа запрашивает **операцию ввода-вывода** через сокет, эта операция должна быть хотя бы частично выполнена до того, как управление вернется к программе (или должна быть возвращена ошибка). Как может случиться, что операция выполнена частично? Например, если программа пытается читать 100 байт из блокирующего сокета, то функция чтения (например, `recv()`) вернет управление, только если из сокета удалось прочитать хотя бы один байт (или возникла ошибка). Если из сети не поступает никаких данных, то выполнение текущего потока программы блокируется, т. е. поток будет ждать поступления данных. Иногда поток может ждать неопределенно долго. При попытке отправить данные текущий поток может быть блокирован, если буфер записи операционной системы заполнен. В таком случае функция отправки (например, `send()`) будет ждать, пока ОС отправит данные из буфера в сеть, освободит место и скопирует в буфер данные, которые программа хочет отправить. Функция `send()` вернет управление, только когда данные полностью или частично (зависит от ОС) скопированы в буфер (или в случае ошибки).

Неблокирующие сокеты, как явствует из названия, не блокируют выполнение программы. Если операцию ввода-вывода невозможно выполнить, то соответствующая функция, например `send()` или `recv()`, вернет управление немедленно, сообщив о том, что могла бы произойти блокировка. Предполагается, что программа повторит попытку через некоторое время, а пока может заняться в том же потоке чем-то другим, например обработать какие-то данные, отправить или принять данные по другому сетевому подключению, обновить индикаторы хода выполнения или отреагировать на события в **пользовательском интерфейсе** (**UI**) программы. Программа может также проверять, готов ли сокет к отправке или получению данных, либо ждать готовности сокета. В большинстве ОС имеются функции, которые ждут готовности к вводу-выводу на нескольких сокетах одновременно в течение заданного времени. Примерами таких функций являются `select()` и `poll()`. OpenSSL предоставляет собственные функции для той же цели, а именно `BIO_wait()` и `BIO_socket_wait()`, но они ограничены только одним **базовым объектом ввода-вывода** (**BIO**). Однако если требуется одновременное ожи-

дание на нескольких BIO, то можно получить от этих BIO дескрипторы сокетов с помощью функции `BIO_get_fd()` и передать набор таких дескрипторов функции `select()`, `poll()`, `epoll_wait()`, `kevent()`, `WSAEventSelect()`, `WSAPoll()` – в зависимости от ОС.

Пользуясь неблокирующими сокетами, программа может обеспечить отзывчивость UI даже при медленных сетевых подключениях, не выделяя отдельные потоки для сетевого взаимодействия. Еще одно популярное применение неблокирующих сокетов – серверные программы. Сервер может обслуживать несколько подключений в одном потоке, так что медленные подключения не будут приводить к блокированию. При такой стратегии потребляется меньше системных ресурсов, чем при обслуживании каждого подключения в отдельном потоке или процессе.

Во всех предыдущих примерах использовались блокирующие сокеты. В следующем разделе мы научимся использовать TLS на неблокирующем сокете из программы.

Использование неблокирующих сокетов в TLS

Чтобы научиться использовать TLS на неблокирующем сокете, мы напишем небольшую программу `tls-client-non-blocking`.

Мы собираемся использовать функции OpenSSL, которые прежде не встречались. Вот их страницы руководства:

```
$ man BIO_set_nbio
$ man BIO_should_retry
$ man BIO_wait
```

В основу `tls-client-non-blocking` мы положим программу `tls-client` из главы 9 и изменим в ней блокирующие сокеты на неблокирующие. Точнее, мы будем использовать неблокирующий BIO вместо блокирующего. Внести изменения придется только в функцию `run_tls_client()`.

Изменение функции run_tls_client()

1. Прежде всего переключим BIO в неблокирующий режим. Для этого нужно добавить следующие строки:

```
BIO_set_nbio(ssl_bio, 1);
if (err <= 0) {
    if (error_stream)
        fprintf(
            error_stream,
            "Could not enable non-blocking mode\n");
    goto failure;
}
```

Важно переключить BIO в неблокирующий режим до попытки подключения, иначе переключения не произойдет и BIO останется в блокирующем режиме. В нашем случае это означает, что функцию `BIO_set_nbio()` нужно вызывать раньше `BIO_do_connect()`.

2. Следующий шаг – определить переменные тайм-аута, которые впоследствии будут использоваться при обращениях к `BIO_wait()`:

```
const time_t TIMEOUT_SECONDS = 10;
const unsigned int NAP_MILLISECONDS = 100;
time_t deadline = time(NULL) + TIMEOUT_SECONDS;
```

В нашем примере для простоты используется 10-секундный тайм-аут для всего TLS-подключения. Это нормально, потому что обычно программа работает меньше 1 секунды.

3. Следующий шаг – сделать код установления подключения неблокирующим. Первоначально этот код в программе `tls-client` выглядел так:

```
err = BIO_do_connect(ssl_bio);
if (err <= 0) {
    if (error_stream)
        fprintf(
            error_stream,
            "Could not connect to server %s on port %s\n",
            hostname,
            port);
    goto failure;
}
```

Мы добавим в него цикл повтора:

```
err = BIO_do_connect(ssl_bio);
while (err <= 0 && BIO_should_retry(ssl_bio)) {
    int wait_err = BIO_wait(
        ssl_bio,
        deadline,
        NAP_MILLISECONDS);
    if (wait_err != 1)
        break;
    err = BIO_do_connect(ssl_bio);
}
if (err <= 0) {
    if (error_stream)
        fprintf(
            error_stream,
            "Could not connect to server %s on port %s\n",
            hostname,
            port);
    goto failure;
}
```

В цикле повтора мы вызываем функцию `BIO_should_retry()`, которая проверяет, была ли последняя ошибка ввода-вывода вызвана потен-

циальным блокированием и должны ли мы повторить попытку подключения. Если да, то мы ждем проявления какой-то активности в BIO с помощью функции `BIO_wait()` и делаем повторную попытку.

Как видим, функции `BIO_wait()` передаются аргументы `deadline` и `NAP_MILLISECONDS`. Аргумент `deadline` – самый поздний момент, до которого функция `BIO_wait()` будет ждать. Аргумент `NAP_MILLISECONDS` используется, только если OpenSSL была откомпилирована без поддержки сокетов. Такое бывает необходимо, если ОС не поддерживает стандартных сокетов (разработанных в Калифорнийском университете в Беркли для BSD Unix), а пользуется собственными нестандартными сокетами. Такого рода ОС обычно встречаются во встраиваемых устройствах. Если OpenSSL откомпилирована без поддержки сокетов, то `BIO_wait()` не сможет проверить наличие сетевой активности в сокете и вместо этого будет спать `NAP_MILLISECONDS`, а затем вернет управление. Если текущая ОС поддерживает стандартные сокеты (наиболее распространенный случай), то аргумент `NAP_MILLISECONDS` игнорируется, и `BIO_wait()` возвращает управление, когда либо сокет доступен для повтора операции ввода-вывода, либо истек срок ожидания, либо произошла ошибка.

`BIO_wait()` – удобная функция, потому что принимает BIO (а не сокет) в качестве параметра, имеет простую сигнатуру и скрывает от разработчика сложность низкоуровневых функций ОС. Однако важно понимать, что `BIO_wait()` обладает куда меньшей гибкостью, чем низкоуровневые функции ОС, например `select()` и `poll()`:

- как уже отмечалось, `BIO_wait()` может ждать только одного BIO, тогда как низкоуровневые функции могут ждать много сокетов одновременно;
- срок действия, передаваемый `BIO_wait()`, измеряется в секундах, это очень много. Низкоуровневые функции ОС принимают тайм-ауты, заданные с точностью по меньшей мере до миллисекунд;
- `BIO_wait()` может ждать только разрешения предыдущей ошибки ввода-вывода в BIO. Низкоуровневые функции ОС могут ждать наступления различных событий на любом переданном сокете. Однако `BIO_wait()` автоматически решает, события какого типа ожидать, и в этом ее преимущество.

Если вам нужно больше гибкости, чем может предложить `BIO_wait()`, можете получить от BIO внутренний файловый дескриптор с помощью функции `BIO_get_fd()` и применить к нему низкоуровневые функции ОС.

Когда программа обменивается данными по протоколу TLS, может случиться, что протокол хочет читать из сети в тот момент, когда ваша программа хочет записывать в сеть, и наоборот. Например, любая сторона TLS-подключения в любой момент может захотеть обновить сеансовые ключи. В нашей программе `tls-client-non-blocking` такие ситуации обрабатываются автоматически с помощью функций `BIO_should_retry()` и `BIO_wait()`. Но разработчик другой программы может использовать сокеты ОС напрямую и ждать их с помощью низкоуровневых функций ОС, например `select()` или `poll()`, а не `BIO_wait()`. В таком случае разработчик должен знать, чего ждать – чтения

или записи. Это позволяют определить такие функции, как BIO_should_read(), BIO_should_write(), BIO_should_io_special() и BIO_retry_type(). Можно также воспользоваться функцией SSL_get_error() и анализировать возвращенный ей код ошибки: SSL_ERROR_WANT_READ или SSL_ERROR_WANT_WRITE. Предположим, что наша программа хочет читать, используя функцию BIO_read(), но протокол TLS хочет записывать, и мы определили это, получив true от функции BIO_should_write(). Что делать? Нужно ли записать что-то в TLS-подключение функцией BIO_write()? Но что писать, если писать нечего, – вызвать BIO_write() с нулевым числом байтов? Нет. Не надо ничего писать. А надо ждать у сокета ОС, пока не *станет возможно писать* в этот сокет. После этого мы должны повторить операцию *чтения*, которую и собирались выполнить изначально, например BIO_read(). Может показаться, что это противоречит интуиции, но да, именно так – следует повторить чтение, хотя протокол TLS жаждет записи. Библиотека OpenSSL позаботится об операциях чтения и записи, необходимых для удовлетворения TLS. OpenSSL прочитает и запишет необходимые служебные данные, а затем прочитает прикладные данные и вернет их нам.

А теперь продолжим изменять функцию run_tls_client().

4. Следующий шаг – изменить код отправки данных с учетом неблокирующего режима. Оригинальный код в программе tls-client выглядел так:

```
int nbytes_written = BIO_write(
    ssl_bio, out_buf, request_length);
if (nbytes_written != request_length) {
    if (error_stream)
        fprintf(
            error_stream,
            "Could not send all data to the server\n");
    goto failure;
}
```

Отметим, что показанный выше код немного упрощен, даже в блокирующем режиме, чтобы сделать его понятнее. Предполагается, что BIO_write() в случае успеха возвращает управление только после отправки всего запроса. В большинстве случаев так оно и есть, потому что запросы невелики и сетевые буферы ОС не заполнены ранее отправленными данными. Но было бы надежнее организовать цикл и проверять в нем, отправлены ли данные целиком; если нет, то нужно продолжить отправку с того места, где она была прервана. Мы также будем использовать функции BIO_should_retry() и BIO_wait(), чтобы определить, готов ли BIO к записи. Вот как должен выглядеть код отправки в неблокирующем режиме:

```
int nbytes_written_total = 0;
while (nbytes_written_total < request_length) {
    int nbytes_written = BIO_write(
        ssl_bio,
        out_buf + nbytes_written_total,
```

```
            request_length - nbytes_written_total);
    if (nbytes_written > 0) {
        nbytes_written_total += nbytes_written;
        continue;
    }
    if (BIO_should_retry(ssl_bio)) {
        BIO_wait(
            ssl_bio,
            deadline,
            NAP_MILLISECONDS);
        continue;
    }
    if (error_stream)
        fprintf(
            error_stream,
            "Could not send all data to the server\n");
    goto failure;
}
```

5. Следующий шаг – модифицировать код получения данных, приспособив его к неблокирующему режиму. Вот как код выглядит сейчас:

```
while ((SSL_get_shutdown(ssl) & SSL_RECEIVED_SHUTDOWN)
        != SSL_RECEIVED_SHUTDOWN) {
    int nbytes_read = BIO_read(ssl_bio, in_buf, BUF_SIZE);
    if (nbytes_read <= 0) {
        int ssl_error = SSL_get_error(ssl, nbytes_read);
        if (ssl_error == SSL_ERROR_ZERO_RETURN)
            break;
        if (error_stream)
            fprintf(
                error_stream,
              "Error %i while reading data "
                "from the server\n",
                ssl_error);
        goto failure;
    }
    fwrite(in_buf, 1, nbytes_read, stdout);
}
```

Как видим, в оригинальном коде уже есть цикл получения. Мы добавим код повтора для неблокирующего режима, в результате чего код примет вид:

```
while ((SSL_get_shutdown(ssl) & SSL_RECEIVED_SHUTDOWN)
          != SSL_RECEIVED_SHUTDOWN) {
    int nbytes_read = BIO_read(ssl_bio, in_buf, BUF_SIZE);
    if (nbytes_read > 0) {
        fwrite(in_buf, 1, nbytes_read, stdout);
        continue;
    }
    if (BIO_should_retry(ssl_bio)) {
```

```
            err = BIO_wait(
                ssl_bio,
                deadline,
                NAP_MILLISECONDS);
            continue;
        }
        int ssl_error = SSL_get_error(ssl, nbytes_read);
        if (ssl_error == SSL_ERROR_ZERO_RETURN)
            break;
        if (error_stream)
            fprintf(
                error_stream,
                "Error %i while reading data "
                "from the server\n",
                ssl_error);
        goto failure;
    }
```

Это все изменения, которые нужно внести для использования TLS на неблокирующем сокете.

Полный исходный код программы `tls-client-non-blocking` находится на GitHub по адресу https://github.com/PacktPublishing/Demystifying-Cryptography-with-OpenSSL-3/blob/main/Chapter11/tls-client-non-blocking.c.

Выполнение программы tls-client-non-blocking

Для запуска программы нам, как и в предыдущем примере, понадобится программа `tls-server` из главы 9. Также будет нужен файл с сертификатом УЦ `ca_cert.pem`, чтобы программа `tls-client-non-blocking` могла проверить сертификат сервера.

1. Откройте два окна терминала.
2. В первом окне запустите программу `tls-server`:

```
$ ./tls-server 4433 server_keypair.pem server_cert.pem
*** Listening on port 4433
```

3. Во втором окне запустите программу `tls-client-non-blocking`:

```
$ ./tls-client-non-blocking localhost 4433 ca_cert.pem
*** Sending to the server:
GET / HTTP/1.1
Host: localhost
Connection: close
User-Agent: Example TLS client
*** Sending to the server finished
*** Receiving from the server:
HTTP/1.0 200 OK
Content-type: text/plain
Connection: close
Server: Example TLS server
```

```
Hello from the TLS server!
*** Receiving from the server finished
TLS communication succeeded
```

Как видим, взаимодействие по протоколу TLS с сервером `localhost` через неблокирующий сокет завершилось успешно, мы смогли отправить данные серверу и получить от него данные. Но будет ли программа работать через интернет?

4. Чтобы выяснить это, попробуем обратиться к серверу `www.example.org`:

```
$ ./tls-client-non-blocking www.example.org 443
*** Sending to the server:
GET / HTTP/1.1
Host: localhost
Connection: close
User-Agent: Example TLS client
*** Sending to the server finished
*** Receiving from the server:
... много текста ...
*** Receiving from the server finished
TLS communication succeeded
```

Как видим, обмен данными с `www.example.org` тоже прошел без сучка без задоринки. Таким образом, в неблокирующем режиме наша программа работает, как и ожидалось.

Использование TLS на неблокирующих сокетах – весьма важный сценарий. В следующих двух разделах мы поговорим еще об одном сценарии продвинутого использования TLS: на нестандартных сокетах с помощью BIO памяти. Это может понадобиться, если целевая ОС не поддерживает стандартные сокеты.

Что такое TLS на нестандартных сокетах

Стандартные сетевые сокеты поддерживаются большинством ОС. Но встречаются такие, особенно встраиваемые, которые поддерживают только свои собственные нестандартные сокеты или обработчики подключений. Как использовать OpenSSL в таких ОС? Это возможно с помощью объектов BIO памяти. OpenSSL может устанавливать TLS-подключения целиком в памяти.

На рис. 11.1 показан поток данных, входящий и исходящий из нестандартного сокета, при использовании BIO памяти.

Таким образом, когда программа хочет получить открытый текст из TLS-подключения, происходит следующее.

1. Программа сохраняет шифртекст из нестандартного сокета во входном буфере шифртекста.
2. Программа записывает шифртекст в читающий BIO памяти.
3. OpenSSL расшифровывает шифртекст из читающего BIO памяти и помещает результирующий открытый текст во входной буфер открытого текста.

4. Программа получает данные, принятые по TLS-подключению, из входного буфера открытого текста.

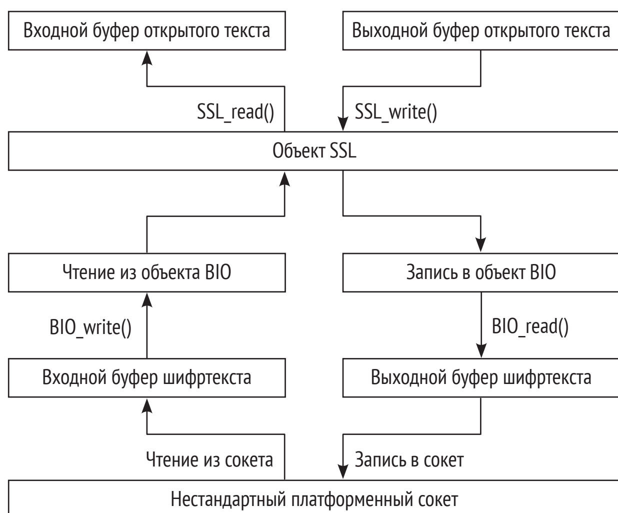


Рис. 11.1 ❖ Входящий и исходящий из нестандартного сокета потоки данных при использовании BIO памяти в TLS

Когда программа хочет передать открытый текст по TLS-подключению, обработка данных происходит в противоположном направлении.

1. Программа помещает открытый текст, подлежащий передаче, в выходной буфер открытого текста.
2. OpenSSL шифрует открытый текст по протоколу TLS и помещает результирующий шифртекст в пишущий BIO памяти.
3. Программа читает шифртекст из пишущего BIO памяти в выходной буфер шифртекста.
4. Программа отправляет шифртекст в нестандартный платформенный сокет.

В следующем разделе мы научимся реализовывать TLS на нестандартных сокетах в программе.

Использование TLS на нестандартных сокетах

Для знакомства с использованием TLS на нестандартном сокете напишем небольшую программу `tls-client-memory-bio`.

В основу `tls-client-memory-bio` положим программу `tls-client` из главы 9 и адаптируем ее код под работу с BIO памяти.

В исходный код `tls-client` придется внести много изменений. Например, мы больше не будем использовать SSL BIO – обертку вокруг объекта `SSL`.

В предыдущих примерах это было удобно, потому что он автоматически сцеплялся с BIO подключения. Но теперь нам это не нужно, мы будем осуществлять ввод-вывод непосредственно через объект `SSL`, используя функции `SSL_read()` и `SSL_write()` вместо `BIO_read()` и `BIO_write()`. Прямой ввод-вывод через объект `SSL` не только упростит код, но и позволит продемонстрировать применение некоторых новых функций.

Поскольку мы собираемся использовать новые функции OpenSSL, будет полезно ознакомиться с их страницами руководства:

```
$ man SSL_new
$ man SSL_set_bio
$ man SSL_connect
$ man SSL_want_read
$ man SSL_read
$ man SSL_write
$ man SSL_shutdown
$ man SSL_free
$ man BIO_new_connect
$ man BIO_set_mem_eof_return
$ man BIO_pending
```

Нам также понадобится присоединить два BIO памяти к объекту `SSL`. Один из них будет использоваться для чтения, а другой – для записи шифртекста. Объект `SSL` будет автоматически читать и записывать шифртекст в эти BIO. Мы не станем присоединять к `SSL` объекты BIO для открытого текста. Вместо этого для чтения и записи открытого текста будут использоваться функции `SSL_read()` и `SSL_write()`.

Нестандартный сетевой сокет будет представлен BIO подключения. На самом деле BIO подключения не является нестандартным сокетом, но его вполне достаточно для демонстрации того, как передаются данные между BIO памяти и сетевым подключением. И в отличие от настоящих нестандартных сокетов, BIO подключения работают на обычном ПК, на котором мы запускаем все примеры программ.

Реализация функции service_bios()

Прежде всего реализуем функцию для передачи данных между BIO памяти, используемым объектом `SSL` и сетевым сокетом, представленным BIO подключения.

1. Эта функция будет вызываться до `SSL_read()` или после всех остальных функций ввода-вывода `SSL_*()`. Поскольку функция обслуживает три BIO, назовем ее `service_bios()`:

```
int service_bios(
    BIO* mem_rbio,
    BIO* mem_wbio,
    BIO* tcp_bio,
    int want_read);
```

Функция принимает следующие параметры:

- mem_rbio – BIO памяти, из которого объект SSL читает шифртекст;
- mem_wbio – BIO памяти, в который объект SSL записывает шифртекст;
- tcp_bio – BIO подключения, который читает и записывает в сеть;
- want_read – булев флаг. Если true, то функция service_bios() не только записывает ожидающие данные в сеть, но и читает из сети. Если false, то service_bios() только записывает в сеть.

Функция service_bios() возвращает 1 в случае успеха и –1 в случае ошибки.

Код функции начинается с определения кода возврата по умолчанию и выделения памяти для буферов:

```
int err = 1;
const size_t BUF_SIZE = 16 * 1024;
char* in_buf = malloc(BUF_SIZE);
char* out_buf = malloc(BUF_SIZE);
```

2. Следующий шаг – записать ожидающие данные в сеть:

```
while (BIO_pending(mem_wbio)) {
    int nbytes_read =
        BIO_read(mem_wbio, out_buf, BUF_SIZE);
    int nbytes_written_total = 0;
    while (nbytes_written_total < nbytes_read) {
        int nbytes_written =
            BIO_write(tcp_bio, out_buf, nbytes_read);
        if (nbytes_written > 0) {
            nbytes_written_total += nbytes_written;
            continue;
        } else {
            goto failure;
        }
    }
}
```

Как видим, мы просто читаем данные из mem_wbio и записываем в tcp_bio.

3. Далее читаем данные из сети, если это действие было запрошено:

```
if (want_read) {
    int nbytes_read =
        BIO_read(tcp_bio, in_buf, BUF_SIZE);
    if (nbytes_read > 0) {
        BIO_write(mem_rbio, in_buf, nbytes_read);
    } else {
        goto failure;
    }
}
```

На этот раз мы читаем из tcp_bio и записываем в mem_rbio, откуда их впоследствии заберет объект SSL.

4. В конце производим очистку и возвращаем код ошибки:

```
    goto cleanup;
failure:
    err = -1;
cleanup:
    free(out_buf);
    free(in_buf);
    return err;
```

Переработанная функция run_tls_client()

Теперь модифицируем функцию `run_tls_client()`, в которой сосредоточена большая часть кода.

1. В начале объявим переменные и выделим память для буферов:

```
int exit_code = 0;
int err = 1;
SSL_CTX* ctx = NULL;
BIO* tcp_bio = NULL;
BIO* mem_rbio = NULL;
BIO* mem_wbio = NULL;
SSL* ssl = NULL;
const size_t BUF_SIZE = 16 * 1024;
char* in_buf = malloc(BUF_SIZE);
char* out_buf = malloc(BUF_SIZE);
```

2. Следующий шаг – создание объекта `SSL_CTX`, загрузка надежных сертификатов и разрешение проверки сертификата другой стороны. Код такой же, как в оригинальной программе `tls-client`:

```
ERR_clear_error();
ctx = SSL_CTX_new(TLS_client_method());
if (trusted_cert_fname)
    err = SSL_CTX_load_verify_locations(
        ctx, trusted_cert_fname, NULL);
else
    err = SSL_CTX_set_default_verify_paths(ctx);
if (err <= 0) {
    if (error_stream)
        fprintf(
            error_stream,
            “Could not load trusted certificates\n”);
    goto failure;
}
SSL_CTX_set_verify(ctx, SSL_VERIFY_PEER, NULL);
```

3. Следующий шаг – установление *TCP*-подключения (не TLS!) к серверу. TLS-подключение будет установлено позже. Как уже было сказано, для моделирования нестандартного сокета мы воспользуемся BIO подключения:

```
tcp_bio = BIO_new_connect(hostname);
BIO_set_conn_port(tcp_bio, port);
err = BIO_do_connect(tcp_bio);
if (err <= 0) {
    if (error_stream)
        fprintf(
            error_stream,
            "Could not connect to server %s on port %s\n",
            hostname,
            port);
    goto failure;
}
```

4. Следующий шаг – создание и настройка читающего и записывающего BIO памяти для объекта SSL:

```
mem_rbio = BIO_new(BIO_s_mem());
BIO_set_mem_eof_return(mem_rbio, -1);
mem_wbio = BIO_new(BIO_s_mem());
BIO_set_mem_eof_return(mem_wbio, -1);
```

   Вызов `BIO_set_mem_eof_return()` необходим, чтобы избежать ошибки конца файла в случае пустого BIO.

5. Далее создадим объект SSL, представляющий TLS-подключение, и присоединим к нему только что созданные читающий и записывающий BIO памяти:

```
ssl = SSL_new(ctx);
SSL_set_bio(ssl, mem_rbio, mem_wbio);
```

   Заметим, что после вызова `SSL_set_bio()` объект `SSL` становится владельцем присоединенных BIO. При освобождении объекта `SSL` функцией `SSL_free()` память, выделенная BIO, освобождается функцией `BIO_free_all()`.

6. Следующий шаг – задание имени сервера для расширения TLS «**индикация имени сервера**» (Server Name Indication – **SNI**) и для проверки имени сервера в сертификате. Этот код тот же, что в программе `tls-client`:

```
SSL_set_tlsext_host_name(ssl, hostname);
SSL_set1_host(ssl, hostname);
```

7. Далее идет TLS-квитирование:

```
while (1) {
    err = SSL_connect(ssl);
    int ssl_error = SSL_get_error(ssl, err);
    if (ssl_error == SSL_ERROR_WANT_READ
        || ssl_error == SSL_ERROR_WANT_WRITE
        || BIO_pending(mem_wbio)) {
        int service_bios_err = service_bios(
            mem_rbio, mem_wbio, tcp_bio, SSL_want_read(ssl));
```

```
        if (service_bios_err != 1) {
            if (error_stream)
                fprintf(
                    error_stream,
                    "Socket error during TLS handshake\n");
            goto failure;
        }
        continue;
    }
    break;
}
if (err <= 0) {
    if (error_stream)
        fprintf(
            error_stream,
            "TLS error %i during TLS handshake\n",
            SSL_get_error(ssl, err));
    goto failure;
}
```

Заметим, что мы в цикле вызываем `SSL_connect()` и `service_bios()`, пока процедура TLS-квитирования не завершится. Во время квитирования и позже, при чтении и записи, мы проверяем ошибки `SSL_ERROR_WANT_READ` и `SSL_ERROR_WANT_WRITE`. Как объяснялось выше, это необходимо, потому что протокол TLS может читать или записывать свои служебные данные в любой момент. В зависимости от версии TLS и от существования кешированных сеансовых данных, оставшихся от прежних подключений, TLS-квитирование может потребовать от одного до двух сетевых оборотов. Поэтому в функции `service_bios()` нам, возможно, придется прочитать и записать данные в сокет ОС дважды в процессе квитирования.

8. Далее создадим HTTP-запрос, который будет отправлен серверу. Это тот же код, что в программе `tls-client`:

```
snprintf(
    out_buf,
    BUF_SIZE,
    "GET / HTTP/1.1\r\n"
    "Host: %s\r\n"
    "Connection: close\r\n"
    "User-Agent: Example TLS client\r\n"
    "\r\n",
    hostname);
int request_length = strlen(out_buf);
```

9. Следующий шаг – отправка запроса серверу:

```
printf("*** Sending to the server:\n");
printf("%s", out_buf);
int nbytes_written_total = 0;
while (nbytes_written_total < request_length) {
```

```
    int nbytes_written = SSL_write(
        ssl,
        out_buf + nbytes_written_total,
        request_length - nbytes_written_total);
    if (nbytes_written > 0) {
        nbytes_written_total += nbytes_written;
        continue;
    }
    int ssl_error = SSL_get_error(ssl, err);
    if (ssl_error == SSL_ERROR_WANT_READ
        || ssl_error == SSL_ERROR_WANT_WRITE
        || BIO_pending(mem_wbio)) {
        int service_bios_err = service_bios(
            mem_rbio, mem_wbio, tcp_bio, SSL_want_read(ssl));
        if (service_bios_err != 1) {
              if (error_stream)
                  fprintf(
                      error_stream,
                      "Socket error while sending data "
                      "to the server\n");
              goto failure;
        }
        continue;
    }
    if (error_stream)
        fprintf(
            error_stream,
            "TLS error %i while reading data "
            "to the server\n",
            ssl_error);
    goto failure;
}
printf("*** Sending to the server finished\n");
```

По аналогии с предыдущим кодом мы в цикле вызываем `SSL_write()` и `service_bios()`, пока весь запрос не будет отправлен. И хотя мы производим только запись по TLS-подключению, проверяются оба кода ошибки, `SSL_ERROR_WANT_READ` и `SSL_ERROR_WANT_WRITE`, потому что протокол TLS может в любой момент читать и записывать служебные данные, а мы должны удовлетворить эти запросы в функции `service_bios()`.

10. После отправки запроса необходимо прочитать ответ сервера:

```
printf("*** Receiving from the server:\n");
while ((SSL_get_shutdown(ssl) & SSL_RECEIVED_SHUTDOWN)
          != SSL_RECEIVED_SHUTDOWN) {
    int service_bios_err = 1;
    if (!BIO_pending(mem_rbio))
        service_bios_err = service_bios(
            mem_rbio, mem_wbio, tcp_bio, 1);
    if (service_bios_err != 1) {
        if (error_stream)
            fprintf(
```

```
                error_stream,
                "Socket error while reading data "
                "from the server\n");
        goto failure;
    }
    int nbytes_read = SSL_read(ssl, in_buf, BUF_SIZE);
    if (nbytes_read > 0) {
        fwrite(in_buf, 1, nbytes_read, stdout);
        continue;
    }
    int ssl_error = SSL_get_error(ssl, err);
    if (ssl_error == SSL_ERROR_NONE
        || ssl_error == SSL_ERROR_WANT_READ
        || ssl_error == SSL_ERROR_WANT_WRITE
        || BIO_pending(mem_wbio))
        continue;
    if (ssl_error == SSL_ERROR_ZERO_RETURN)
        break;
    if (error_stream)
        fprintf(
            error_stream,
            "TLS error %i while reading data "
            "from the server\n",
            ssl_error);
    goto failure;
}
printf("*** Receiving from the server finished\n");
```

По аналогии с предыдущим кодом мы в цикле вызываем `SSL_read()` и `service_bios()`, пока весь ответ не будет получен. Но обратите внимание, что теперь `service_bios()` вызывается перед функцией ввода-вывода `SSL`. Мы также проверяем, существуют ли уже ожидающие данные в читающем BIO, `mem_rbio`. Такой вызов функции имеет смысл, потому что сначала мы должны прочитать из сети, а только потом можем обработать полученный шифртекст с помощью функции `SSL_read()`. Как и раньше, мы должны обрабатывать обе ошибки `SSL_ERROR_WANT_READ` и `SSL_ERROR_WANT_WRITE`, хотя только читаем данные приложения.

11. После чтения ответа сервера TLS-подключение можно разомкнуть:

```
while (1) {
    err = SSL_shutdown(ssl);
    int ssl_error = SSL_get_error(ssl, err);
    if (ssl_error == SSL_ERROR_WANT_READ
          || ssl_error == SSL_ERROR_WANT_WRITE
          || BIO_pending(mem_wbio)) {
          int service_bios_err = service_bios(
              mem_rbio, mem_wbio, tcp_bio, SSL_want_read(ssl));
          if (service_bios_err != 1) {
              if (error_stream)
                  fprintf(
                      error_stream,
```

```
                    "Socket error during TLS shutdown\n");
                goto failure;
            }
            continue;
        }
        break;
    }
    if (err != 1) {
        if (error_stream)
            fprintf(
                error_stream,
                "TLS error during TLS shutdown\n");
        goto failure;
    }
```

12. И завершим функцию `run_tls_client()` очисткой, сообщениями об ошибках и возвратом кода ошибки:

```
failure:
    exit_code = 1;
cleanup:
    if (ssl)
        SSL_free(ssl);
    if (tcp_bio)
        BIO_free_all(tcp_bio);
    if (ctx)
        SSL_CTX_free(ctx);
    free(out_buf);
    free(in_buf);
    if (ERR_peek_error()) {
        exit_code = 1;
        if (error_stream) {
            fprintf(
                error_stream,
                "Errors from the OpenSSL error queue:\n");
            ERR_print_errors_fp(error_stream);
        }
        ERR_clear_error();
    }
    return exit_code;
```

Полный исходный код программы `tls-client-memory-bio` находится на GitHub по адресу https://github.com/PacktPublishing/Demystifying-Cryptography-with-OpenSSL-3/blob/main/Chapter11/tls-client-memory-bio.c.

Выполнение программы tls-client-memory-bio

Как и в предыдущем примере, нам понадобится программа `tls-server` из главы 9, а также файл `ca_cert.pem`, содержащий сертификат УЦ, для проверки сертификата сервера.

1. Откройте два окна терминала.
   I. В первом окне запустите программу tls-server:

```
$ ./tls-server 4433 server_keypair.pem server_cert.pem
*** Listening on port 4433
```

   II. Во втором окне запустите программу tls-client-memory-bio:

```
$ ./tls-client-memory-bio localhost 4433 ca_cert.pem
*** Sending to the server:
GET / HTTP/1.1
Host: localhost
Connection: close
User-Agent: Example TLS client
*** Sending to the server finished
*** Receiving from the server:
HTTP/1.0 200 OK
Content-type: text/plain
Connection: close
Server: Example TLS server
Hello from the TLS server!
*** Receiving from the server finished
TLS communication succeeded
```

   Как видим, обмен данными с localhost по TLS с помощью BIO памяти, имитирующего нестандартный сокет, завершился успешно.
2. Можем также попробовать подключиться к серверу www.example.org и убедиться, что программа tls-client-memory-bio работает и через интернет:

```
$ ./tls-client-memory-bio www.example.org 443
*** Sending to the server:
GET / HTTP/1.1
Host: localhost
Connection: close
User-Agent: Example TLS client
*** Sending to the server finished
*** Receiving from the server:
… much text …
*** Receiving from the server finished
TLS communication succeeded
```

   Как видим, взаимодействие с www.example.org также не вызывает нареканий. Мы можем сделать вывод, что наша программа tls-client-memory-bio работает, как мы и ожидали.

Резюме

В этой главе мы узнали, что такое закрепление сертификата в TLS и в каких случаях это может быть полезно. Мы также узнали, как реализовать закрепление с помощью функции обратного вызова для проверки сертификата. Затем поговорили о различиях между блокирующими и неблокирующими сокетами и о том, как работать с неблокирующими сетевыми подключениями. Далее мы видели, как использовать BIO памяти для установления TLS-подключений по нестандартным сокетам. Все эти знания помогают разрабатывать более безопасные, отзывчивые и производительные приложения, а также адаптировать программы к встраиваемым ОС.

В следующей главе мы узнаем, как настроить в организации мини-УЦ, выпускающий сертификаты для внутреннего пользования.

Часть V

МИНИ-УЦ

В этой части мы узнаем, как командные утилиты OpenSSL позволяют создать и эксплуатировать миниатюрный **удостоверяющий центр** (**УЦ**), выпускающий сертификаты для внутреннего использования в организации. Мини-УЦ может быть полезен для создания внутренней инфраструктуры открытых ключей, контроля над сертификатами для внутреннего пользования и экономии затрат на заказ сертификатов у коммерческих УЦ. Работа с мини-УЦ иллюстрируется на примерах конфигурационных файлов и команд.

Эта часть содержит всего одну главу:

- главу 12 «Эксплуатация мини-УЦ».

Глава 12

Эксплуатация мини-УЦ

В главе 8 мы узнали об **инфраструктуре открытых ключей** (**PKI**), основанной на сертификатах X.509. В этой главе мы поговорим о том, как с помощью подкоманды `openssl ca` эксплуатировать мини-УЦ, выпускающий сертификаты для использования внутри организации.

Рассматриваются следующие темы:

- подкоманда `openssl ca`;
- генерирование сертификата корневого УЦ;
- генерирование сертификата промежуточного УЦ;
- генерирование сертификата веб-сервера;
- генерирование сертификата веб-клиента и почтового клиента;
- отзыв сертификатов и генерирование CRL;
- извещение о состоянии отзыва сертификатов по протоколу OCSP.

Технические требования

В этой главе описываются команды, запускаемые из командной строки. Понадобятся программа `openssl` и динамические библиотеки OpenSSL.

Нам встретится много конфигурационных файлов и команд в виде скриптов оболочки. Их можно найти по адресу https://github.com/PacktPublishing/Demystifying-Cryptography-with-OpenSSL-3/tree/main/Chapter12.

Подкоманда openssl ca

Подкоманда `openssl ca` бывает полезна для эксплуатация мини-УЦ внутри организации. Такой УЦ может, например, выпускать сертификаты для внутренних серверов. Использование внутреннего УЦ экономит затраты на приобретение сертификатов у коммерческих УЦ. Но это не единственное преимущество. Многие внутренние серверы вообще не должны быть доступны из интернета. Это ограничение препятствует автоматической проверке сервера

со стороны внешних УЦ, а без такой проверки не обойтись при выпуске дешевых или бесплатных сертификатов. Кроме того, в ряде случаев нежелательно раскрывать само существование или имена внутренних серверов. Заказывая сертификат у внешнего УЦ, вы должны сообщить ему имя внутреннего сервера. Более того, УЦ может опубликовать информацию о сертификате в журнале **прозрачности сертификатов** (Certificate Transparency – **CT**), что ведет к еще более широкому и, следовательно, нежелательному распространению информации о внутренних серверах компании.

Еще одна причина завести внутренний УЦ – выпуск клиентских сертификатов. Использовать сертификат, выпущенный внутренним УЦ, на публичном веб-сервере не очень удобно, потому что каждый пользователь этого сервера должен будет установить сертификат этого УЦ в хранилище надежных сертификатов браузера. Но клиентский сертификат, выпущенный внутренним УЦ, лишен данного недостатка, поскольку обычно используется только для аутентификации небольшим числом серверов.

В некоторых организациях использование собственного УЦ для внутреннего пользования может дать большую свободу по сравнению с внешним УЦ. Например, получение внешнего сертификата нужно согласовать с отделом информационной безопасности, а его оплату – с бухгалтерией. При наличии собственного УЦ вы можете свободно выпускать сертификаты для своих целей, ни с кем не согласовывая.

Подкоманда `openssl ca` может принимать параметры из командной строки или из конфигурационного файла OpenSSL. В конфигурационном файле можно хранить различные параметры библиотеки и утилит OpenSSL. Файл по умолчанию, `openssl.cnf`, входит в состав дистрибутива OpenSSL. В процессе установки OpenSSL этот файл устанавливается в каталог `ssl`, например `/opt/openssl-3.0.0/ssl/openssl.cnf` или `/etc/ssl/openssl.cnf`. Для работы с `openssl ca` обычно создается специальный конфигурационный файл, представляющий конкретный УЦ, например корневой или промежуточный, взамен файла по умолчанию.

Рекомендуется задавать большинство параметров для подкоманды `openssl ca` в конфигурационном файле, чтобы не повторять их каждый раз в командной строке при выпуске разных сертификатов. Кроме того, некоторые параметры можно задавать только в конфигурационном файле. Если один и тот же параметр задан и в конфигурационном файле, и в командной строке, то командная строка имеет преимущество.

Параметры `openssl ca` и конфигурационные файлы документированы на следующих страницах руководства:

```
$ man openssl-ca
$ man 5ssl config
$ man x509v3_config
```

Наряду с конфигурационным файлом `openssl ca` использует другие обязательные и необязательные файлы, например индексные файлы сертификатов, файлы серийных номеров, файлы случайных начальных значений и файлы номеров CRL.

Подкоманда `openssl ca` может выпускать и отзывать сертификаты, а также генерировать **списки отозванных сертификатов** (**CRL**). При выпуске и от-

зыве сертификатов `openssl ca` обновляет индексный файл, или базу данных о сертификатах.

`openssl ca` может выпускать сертификаты с последовательными или случайными серийными номерами. Если используются последовательные номера, то следующий номер сохраняется в файле серийного номера и обновляется автоматически. Но по соображениям безопасности лучше выпускать сертификаты со случайными серийными номерами.

CRL можно генерировать с **номером CRL** или без него. Номер CRL – это, по сути дела, версия CRL. Это целое положительное число, которое увеличивается на единицу всякий раз, как УЦ генерирует следующий полный CRL. Лучше включать в CRL номер, потому что это позволяет проверить непрерывность выпуска CRL и генерировать **дельты CRL**. Дельтой называется CRL, содержащий только отличия от конкретной версии обычного полного CRL. Полный CRL, от которого отталкивается дельта, называется **базовым CRL**. Однако дельты CRL пока не поддерживаются `openssl ca` – если вы хотите их генерировать, то придется использовать какой-нибудь другой инструмент. Если генерируется CRL с номером, то сам номер хранится в файле номера CRL и автоматически обновляется подкомандой `openssl ca`.

Посмотрим, как используется подкоманда `openssl ca`. Начнем с генерирования сертификата корневого УЦ.

Генерирование сертификата корневого УЦ

Сначала создадим несколько обязательных каталогов и файлов, а затем сгенерируем сертификат корневого УЦ.

1. Создайте каталог `mini-ca`, в котором будут находиться все относящиеся к нашему УЦ файлы:

   ```
   $ mkdir mini-ca
   $ cd mini-ca
   ```

2. В этом каталоге создайте подкаталог `root` для файлов, относящихся к корневому УЦ;

   ```
   $ mkdir root
   $ cd root
   ```

3. В каталоге `root` создайте подкаталог для выпущенных сертификатов, индексный файл сертификатов и файл номера CRL:

   ```
   $ mkdir issued
   $ echo -n >index.txt
   $ echo 01 >crlnumber.txt
   ```

4. Теперь нужно создать конфигурационный файл для корневого УЦ, назовем его `root.cnf`. Для начала поместим в него несколько обязательных параметров:

```
[ca]
default_ca = CA_default
[CA_default]
database = index.txt
new_certs_dir = issued
certificate = root_cert.pem
private_key = private/root_keypair.pem
```

Параметр `database` определяет индексный файл, или базу данных сертификатов. Это текстовый файл, в котором каждая строка содержит информацию об одном сертификате. Параметр `new_certs_dir` определяет каталог, в который будут записываться новые сертификаты. Параметры `certificate` и `private_key` определяют сертификат текущего УЦ (в нашем случае корневого) и соответствующий ему закрытый ключ.

5. Далее определим несколько необязательных параметров. Наш корневой УЦ будет выпускать только сертификаты промежуточных УЦ, поэтому мы можем установить срок действия сертификата по умолчанию 10 лет:

```
default_days = 3650
```

6. Следующий параметр задает алгоритм вычисления хеша сообщения, который будет использоваться для подписания сертификатов, выпущенных нашим УЦ. Для него существует специальное значение по умолчанию, `default_md`, которое означает, что в подписанном сертификате будет использоваться тот же алгоритм вычисления хеша, что использован в подписывающем сертификате. Например, если подписывающий сертификат подписан с помощью функции хеширования SHA-256, то и для подписания выпущенного сертификата будет использована функция SHA-256. В нашем случае этот параметр не имеет значения, потому что сертификаты типа ED448 всегда используют для подписания других сертификатов алгоритм SHAKE256. Мы включаем этот параметр, просто чтобы продемонстрировать возможности подкоманды `openssl ca`:

```
default_md = default
```

7. Следующий параметр инструктирует `openssl ca` выпускать сертификаты со случайными, а не последовательными серийными номерами:

```
rand_serial = yes
```

8. Следующий параметр позволяет выпускать несколько сертификатов с одним и тем же субъектом. Это может быть полезно для замены сертификатов с истекшим сроком действия:

```
unique_subject = no
```

9. Следующие параметры определяют способ показа сведений о сертификате перед подписанием. В документации рекомендуется задавать

эти параметры, иначе будет использоваться старый дефектный формат отображения:

```
name_opt = ca_default
cert_opt = ca_default
```

10. Следующий параметр обязателен и задает секцию конфигурационного файла, в которой определена подразумеваемая по умолчанию политика субъекта выпущенного сертификата. Мы поговорим об этом подробнее, когда дойдем до самой секции:

```
policy = policy_intermediate_cert
```

11. Следующий параметр задает секцию, содержащую расширения X509v3:

```
x509_extensions = v3_intermediate_cert
```

12. Следующий параметр означает, что расширения X509v3, которые не добавлены УЦ, нужно скопировать из **запроса на подписание сертификата** (**CSR**) в выпущенный сертификат. Избирательное копирование позволяет включать в CSR полезные расширения, например *subjectAltName*, но запрещает переопределять важные расширения, установленные УЦ, например *basicConstraints* или *keyUsage*:

```
copy_extensions = copy
```

13. Следующие параметры относятся к CRL. Они задают подразумеваемую по умолчанию секцию конфигурационного файла, содержащую расширения CRL, файлы номеров CRL и срок действия выпущенного CRL по умолчанию:

```
crl_extensions = crl_extensions_root_ca
crlnumber = crlnumber.txt
default_crl_days = 30
```

14. Следующие параметры определяют подразумеваемые по умолчанию субъект и расширения X509v3 для CSR, когда CSR создается с помощью данного конфигурационного файла. Но с помощью этого конфигурационного файла должен создаваться только CSR для сертификата корневого УЦ. Поэтому удобно определить субъект и расширения X509v3 сертификата корневого УЦ в этом файле – как для генерирования CSR, так и для последующего генерирования сертификата корневого УЦ. Заодно это будет напоминать о том, с сертификатом какого УЦ используется данный конфигурационный файл:

```
[req]
prompt = no
distinguished_name = distinguished_name_root_cert
x509_extensions = v3_root_cert
[distinguished_name_root_cert]
countryName = NO
```

```
stateOrProvinceName = Oslo
localityName = Oslo
organizationName = TLS Experts
commonName = Root CA
```

15. В следующей секции конфигурационного файла определяется подразумеваемая по умолчанию политика субъекта выпущенного сертификата:

```
[policy_intermediate_cert]
countryName = match
stateOrProvinceName = match
localityName = match
organizationName = match
organizationalUnitName = optional
commonName = supplied
emailAddress = optional
```

    Согласно этой политике, субъект выпущенного сертификата промежуточного УЦ должен иметь такое же местоположение и организацию, как субъект сертификата корневого УЦ. Это разумно, потому что мини-УЦ эксплуатируется в пределах одной организации.
16. В следующих секциях конфигурационного файла определяются расширения X509v3 для сертификата корневого УЦ и выпущенного сертификата промежуточного УЦ. Нужную секцию можно задать с помощью параметра `-extensions` в командной строке:

```
[v3_root_cert]
subjectKeyIdentifier = hash
authorityKeyIdentifier = keyid:always, issuer
basicConstraints = critical, CA:TRUE
keyUsage = critical, digitalSignature, cRLSign, keyCertSign
crlDistributionPoints = URI:http://crl.tls-experts.no/root_crl.der
[v3_intermediate_cert]
subjectKeyIdentifier = hash
authorityKeyIdentifier = keyid:always, issuer
basicConstraints = critical, CA:TRUE, pathlen:0
keyUsage = critical, digitalSignature, cRLSign, keyCertSign
crlDistributionPoints = URI:http://crl.tls-experts.no/root_crl.der
```

17. В последних секциях определены расширения CRL, которые следует добавлять в выпускаемые CRL:

```
[crl_extensions_root_ca]
authorityKeyIdentifier = keyid:always, issuer
crlDistributionPoints = URI:http://crl.tls-experts.no/root_crl.der
```

    Полный конфигурационный файл корневого УЦ находится на GitHub по адресу https://github.com/PacktPublishing/Demystifying-Cryptography-with-OpenSSL-3/blob/main/Chapter12/mini-ca/root/root.cnf.
18. Наш конфигурационный файл корневого УЦ готов. Перейдем к следующему шагу – генерированию пары ключей корневого УЦ:

```
$ mkdir private
$ chmod 0700 private
$ openssl genpkey \
    -algorithm ED448 \
    -out private/root_keypair.pem
```

19. Следующий шаг – генерирование CSR для корневого УЦ:

```
$ openssl req \
    -config root.cnf \
    -new \
    -key private/root_keypair.pem \
    -out root_csr.pem \
    -text
```

Заметим, что мы не задали субъект корневого сертификата в командной строке. Подкоманда `openssl req` возьмет субъект из конфигурационного файла `root.cnf`, точнее из секций `req` и `distinguished_name_root_cert`.

20. Подготовив CSR, мы можем выпустить сертификат корневого УЦ:

```
$ openssl ca \
    -config root.cnf \
    -extensions v3_root_cert \
    -selfsign \
    -in root_csr.pem \
    -out root_cert.pem
Using configuration from root.cnf
Check that the request matches the signature
Signature ok
Certificate Details:
    Serial Number:
        ... (длинное шестнадцатеричное число) ...
    Validity
        Not Before: Jul 18 18:54:11 2022 GMT
        Not After : Jul 15 18:54:11 2032 GMT
    Subject:
        countryName               = NO
        stateOrProvinceName       = Oslo
        localityName              = Oslo
        organizationName          = TLS Experts
        commonName                = Root CA
X509v3 extensions:
    X509v3 Subject Key Identifier:
        ... (длинное шестнадцатеричное число) ...
    X509v3 Authority Key Identifier:
        ... (длинное шестнадцатеричное число) ...
    X509v3 Basic Constraints: critical
        CA:TRUE
    X509v3 Key Usage: critical
        Digital Signature, Certificate Sign, CRL Sign
    X509v3 CRL Distribution Points:
        Full Name:
```

```
            URI:http://crl.tls-experts.no/root_crl.der
Certificate is to be certified until Jul 15 18:54:11 2032 GMT (3650 days)
Sign the certificate? [y/n]:y
1 out of 1 certificate requests certified, commit? [y/n]y
Write out database with 1 new entries
Data Base Updated
```

Обратите внимание, что мы использовали для выпуска сертификата подкоманду `openssl ca`, а не `openssl x509`. Кроме того, большинство параметров заданы в конфигурационном файле, а не в командной строке. Мы не задавали в командной строке ни закрытый ключ, ни срок действия, ни расширения X509v3 – все они берутся из конфигурационного файла. Мы задали флаг `-selfsigned`, инструктирующий `openssl ca` выпустить самоподписанный сертификат. При выпуске других сертификатов этот флаг задаваться не будет.

После выпуска сертификата `openssl ca` сохраняет его копию в каталоге `issued` (определен параметром `new_certs_dir` в конфигурационном файле) и обновляет индексный файл сертификатов (определен параметром `database` в конфигурационном файле).

Мы успешно сгенерировали сертификат корневого УЦ, а в следующем разделе сгенерируем сертификат промежуточного УЦ.

Генерирование сертификата промежуточного УЦ

Сертификат промежуточного УЦ генерируется аналогично сертификату корневого УЦ. Для промежуточного УЦ мы будем использовать другой каталог, назовем его `intermediate` и поместим на том же уровне, что каталог `root` внутри каталога `mini-ca`.

1. Создадим каталог `intermediate`, а в нем – необходимые файлы и подкаталог:

```
$ cd mini-ca
$ mkdir intermediate
$ cd intermediate
$ mkdir issued
$ echo -n >index.txt
$ echo 01 >crlnumber.txt
```

2. Теперь надо создать конфигурационный файл для промежуточного УЦ, назовем его `intermediate.cnf`. Этот файл будет похож на файл корневого УЦ. Запишем в него те же самые параметры, что в конфигурационный файл корневого УЦ:

```
[ca]
default_ca = CA_default
[CA_default]
```

```
database = index.txt
new_certs_dir = issued
certificate = intermediate_cert.pem
private_key = private/intermediate_keypair. pem
default_days = 365
default_md = default
rand_serial = yes
unique_subject = no
name_opt = ca_default
cert_opt = ca_default
policy = policy_server_cert
x509_extensions = v3_server_cert
copy_extensions = copy
crl_extensions = crl_extensions_intermediate_ca
crlnumber = crlnumber.txt
default_crl_days = 30
```

Отметим, что параметр `default_days` в этом файле меньше такого же параметра в конфигурационном файле корневого УЦ, потому что промежуточный УЦ выпускает листовые сертификаты, срок действия которых обычно меньше, чем у сертификатов корневого и промежуточного УЦ.

3. Продолжим добавлять в конфигурационный файл параметры, которые помогут построить субъект сертификата промежуточного УЦ:

```
[req]
prompt = no
distinguished_name = distinguished_name_intermediate_cert
[distinguished_name_intermediate_cert]
countryName = NO
stateOrProvinceName = Oslo
localityName = Oslo
organizationName = TLS Experts
commonName = Intermediate CA
```

4. Наш промежуточный УЦ будет выпускать сертификаты для клиентов и серверов. Поэтому необходимо определить политики субъекта для обеих категорий:

```
[policy_server_cert]
countryName = optional
stateOrProvinceName = optional
localityName = optional
organizationName = optional
organizationalUnitName = optional
commonName = supplied
emailAddress = optional
[policy_client_cert]
countryName = optional
stateOrProvinceName = optional
localityName = optional
```

```
organizationName = optional
organizationalUnitName = optional
commonName = supplied
emailAddress = supplied
```

Легко видеть, что политики, определенные в конфигурационном файле промежуточного УЦ, не так ограничительны, как для корневого УЦ. Это связано с тем, что потенциально мы можем выпустить сертификат для любого юридического лица вне нашей организации, например для заказчика или партнера, а также с желанием продемонстрировать разные случаи.

5. В следующих разделах конфигурационного файла промежуточного УЦ определены расширения X509v3, которые будут добавляться в выпускаемые сертификаты разных видов: серверные, клиентские и для ответчиков по протоколу OCSP:

```
[v3_server_cert]
subjectKeyIdentifier = hash
authorityKeyIdentifier = keyid:always, issuer
basicConstraints = critical, CA:FALSE
nsCertType = server
keyUsage = critical, digitalSignature, keyEncipherment
extendedKeyUsage = serverAuth
crlDistributionPoints = URI:http://crl.tls-experts.no/intermediate_crl.der
authorityInfoAccess = OCSP;URI:http://ocsp.tls-experts.no/
[v3_client_cert]
subjectKeyIdentifier = hash
authorityKeyIdentifier = keyid:always, issuer
basicConstraints = critical, CA:FALSE
nsCertType = client, email
keyUsage = critical, nonRepudiation, digitalSignature, keyEncipherment
extendedKeyUsage = clientAuth, emailProtection
crlDistributionPoints = URI:http://crl.tls-experts.no/intermediate_crl.der
authorityInfoAccess = OCSP;URI:http://ocsp.tls-experts.no/
[v3_ocsp_cert]
subjectKeyIdentifier = hash
authorityKeyIdentifier = keyid:always, issuer
basicConstraints = critical, CA:FALSE
keyUsage = critical, digitalSignature
extendedKeyUsage = critical, OCSPSigning
crlDistributionPoints = URI:http://crl.tls-experts.no/intermediate_crl.der
authorityInfoAccess = OCSP;URI:http://ocsp.tls-experts.no/
```

6. В последней секции конфигурационного файла определены расширения, которые будут добавляться в выпускаемые CRL:

```
[crl_extensions_intermediate_ca]
authorityKeyIdentifier = keyid:always, issuer
crlDistributionPoints = URI:http://crl.tls-experts.no/intermediate_crl.der
authorityInfoAccess = OCSP;URI:http://ocsp.tls-experts.no/
```

Полный конфигурационный файл промежуточного УЦ находится на GitHub по адресу https://github.com/PacktPublishing/Demystifying-Cryptography-with-OpenSSL-3/blob/main/Chapter12/mini-ca/intermediate/intermediate.cnf.

7. Конфигурационный файл нашего промежуточного УЦ готов. Теперь сгенерируем для него пару ключей:

```
$ mkdir private
$ chmod 0700 private
$ openssl genpkey \
    -algorithm ED448 \
    -out private/intermediate_keypair.pem
```

8. Далее сгенерируем CSR для промежуточного УЦ:

```
$ openssl req \
    -config intermediate.cnf \
    -new \
    -key private/intermediate_keypair.pem \
    -out intermediate_csr.pem \
    -text
```

9. Как и для корневого УЦ, компоненты субъекта сертификата берутся из конфигурационного файла, а не из командной строки.
10. Следующий шаг – выпуск сертификата промежуточного УЦ, но выпустить его должен корневой УЦ. Поэтому делать это нужно из каталога `root`:

```
$ cd ../root/
$ openssl ca \
    -config root.cnf \
    -extensions v3_intermediate_cert \
    -in ../intermediate/intermediate_csr.pem \
    -out ../intermediate/intermediate_cert.pem
Using configuration from root.cnf
Check that the request matches the signature
Signature ok
Certificate Details:
        ...
Certificate is to be certified until Jul 15 18:55:11 2032 GMT (3650 days)
Sign the certificate? [y/n]:y
1 out of 1 certificate requests certified, commit? [y/n]y
Write out database with 1 new entries
Data Base Updated
```

На этот раз мы использовали расширения X509v3 из секции `v3_intermediate_cert` файла `root.cnf file`, а не из секции `v3_root_cert`, как при генерировании сертификата корневого УЦ.

Сертификат промежуточного УЦ сгенерирован. В следующем разделе мы воспользуемся им для выпуска листового сертификата.

Генерирование сертификата для веб-сервера

Далее мы сгенерируем сертификат для веб-сервера. Как и в предыдущих случаях, создадим для него отдельный каталог и конфигурационный файл.

1. Однако, поскольку мы собираемся выпустить листовый сертификат, нам не нужно создавать в этом каталоге файлы, относящиеся к УЦ. Нужно только создать сам каталог и перейти в него.

   ```
   $ cd mini-ca
   $ mkdir server
   $ cd server
   ```

   Конфигурационный файл сертификата сервера будет гораздо короче предыдущих:

   ```
   [req]
   prompt = no
   distinguished_name = distinguished_name_server_cert
   req_extensions = v3_server_cert
   [distinguished_name_server_cert]
   countryName = NO
   stateOrProvinceName = Oslo
   localityName = Oslo
   organizationName = TLS Experts
   commonName = internal.tls-experts.no
   [v3_server_cert]
   subjectAltName = DNS:mirror1.tls-experts.no, DNS:mirror2.tls-experts.no
   ```

   На этот раз мы определили расширение X509v3, `subjectAltNames`, которое будет добавлено в CSR. Мы задали в расширении несколько значений, разделив их запятыми. Определить несколько значений можно и по-другому – заведя для каждого значения отдельную строку. Этот способ мы рассмотрим ниже при создании клиентского сертификата. Рассматриваемого расширения не было в конфигурационном файле промежуточного УЦ, зато в нем была строка `copy_extensions = copy option`. Следовательно, расширение X509v3 из CSR будет скопировано в результирующий сертификат вместе с другими расширениями, определенными в конфигурационном файле промежуточного УЦ. Конечно, при выпуске сертификата мы сможем просмотреть все сведения о сертификате и увидим все включенные в него расширения X509v3.
2. Следующий шаг – генерирование пары ключей и CSR для сертификата сервера:

   ```
   $ mkdir private
   $ chmod 0700 private
   $ openssl genpkey \
       -algorithm ED448 \
       -out private/server_keypair.pem
   $ openssl req \
       -config server.cnf \
   ```

```
    -new \
    -key private/server_keypair.pem \
    -out server_csr.pem \
    -text
```

Здесь нам не нужно задавать параметр `-reqexts v3_server_cert` в командной строке, потому что секция расширений `v3_server_cert` подразумевается по умолчанию, т. к. в конфигурационном файле имеется строка `req_extensions = v3_server_cert`.

3. Пора генерировать сертификат сервера. Поскольку он будет выпущен промежуточным УЦ, подкоманду `openssl ca` нужно выполнять, находясь в каталоге `intermediate`:

```
$ cd ../intermediate/
$ openssl ca \
    -config intermediate.cnf \
    -in ../server/server_csr.pem \
    -out ../server/server_cert.pem
Using configuration from intermediate.cnf
Check that the request matches the signature
Signature ok
Certificate Details:
          ...
Certificate is to be certified until Jul 19 15:36:44 2023 GMT (365 days)
Sign the certificate? [y/n]:y
1 out of 1 certificate requests certified, commit? [y/n]y
Write out database with 1 new entries
Data Base Updated
```

Команда выпуска сертификата совсем простая – всего три аргумента. Остальные параметры берутся из конфигурационного файла, в т. ч. политика по умолчанию и расширения X509v3. Задав все параметры в конфигурационном файле, мы можем ограничиться простой командой и без труда выпускать сертификаты серверов.

Клиентский сертификат выпускается немного иначе, но тоже без особых трудностей. Как это делается, описано в следующем разделе.

Генерирование сертификата веб-клиента и почтового клиента

Процедуры генерирования сертификатов клиента и сервера похожи.

1. Как обычно, создадим отдельный каталог для сертификата и перейдем в него.

```
$ cd mini-ca
$ mkdir client
$ cd client
```

2. Для генерирования клиентского сертификата воспользуемся следующей конфигурацией, сохраненной в файле `client.cnf`:

```
[req]
prompt                  = no
distinguished_name      = distinguished_name_client_cert
req_extensions          = v3_client_cert
[distinguished_name_client_cert]
countryName             = NO
stateOrProvinceName     = Oslo
localityName            = Oslo
organizationName        = TLS Experts
commonName              = Thor Odinson
emailAddress            = thor@tls-experts.no
[v3_client_cert]
subjectAltName = @subject_alt_names
[subject_alt_names]
email.1 = postmaster@tls-experts.no
email.2 = hostmaster@tls-experts.no
```

Как и в прошлый раз, мы задали расширение `subjectAltName` с несколькими значениями, но определили каждое в отдельной строке.

3. Сгенерируем пару ключей и CSR:

```
$ mkdir private
$ chmod 0700 private
$ openssl genpkey \
    -algorithm ED448 \
    -out private/client_keypair.pem
$ openssl req \
    -config client.cnf \
    -new \
    -key private/client_keypair.pem \
    -out client_csr.pem \
    -text
```

4. Следующий шаг – генерирование клиентского сертификата:

```
$ cd ../intermediate/
$ openssl ca \
    -config intermediate.cnf \
    -policy policy_client_cert \
    -extensions v3_client_cert \
    -in ../client/client_csr.pem \
    -out ../client/client_cert.pem
Using configuration from intermediate.cnf
Check that the request matches the signature
Signature ok
Certificate Details:
          ...
Certificate is to be certified until Jul 19 15:37:44 2023 GMT (365 days)
Sign the certificate? [y/n]:y
1 out of 1 certificate requests certified, commit? [y/n]y
```

```
Write out database with 1 new entries
Data Base Updated
```

Команда выпуска клиентского сертификата немного длиннее, чем серверного, потому что пришлось добавить параметры `-policy` и `-extensions`. Подразумеваемые по умолчанию политика и секция расширений зарезервированы для серверных сертификатов, потому что они выпускаются чаще, чем клиентские. Но если в вашей организации дело обстоит иначе, то ничто не мешает использовать по умолчанию политики и секцию расширения для клиентов.

5. И еще одно: при создании клиентского сертификата полезно упаковать сертификат, его закрытый ключ и цепочку проверки в контейнер типа **PKCS #12**:

```
$ cd ../client/
$ cat \
    ../intermediate/intermediate_cert.pem \
    ../root/root_cert.pem \
    >certfile.pem
$ openssl pkcs12 \
    -export \
    -inkey private/client_keypair.pem \
    -in client_cert.pem \
    -certfile certfile.pem \
    -passout 'pass:SuperPa$$w0rd' \
    -out client_cert.p12
```

Как видим, выпускать сертификаты с помощью подкоманды `openssl ca` нетрудно. А как насчет отзыва сертификатов и генерирования CRL? Разберемся в следующем разделе.

Отзыв сертификатов и генерирование CRL

Прежде чем отзывать сертификат, его нужно выпустить. Выпустим тестовый сертификат по аналогии с тем, что делали выше.

1. Как обычно, первый шаг – создание каталога для файлов сертификатов:

```
$ cd mini-ca
$ mkdir server2
$ cd server2
```

2. Затем создадим следующую конфигурацию в файле `server2.cnf`:

```
[req]
prompt                  = no
distinguished_name      = distinguished_name_server_cert
[distinguished_name_server_cert]
countryName             = NO
stateOrProvinceName     = Oslo
```

```
localityName            = Oslo
organizationName        = TLS Experts
commonName              = server2.tls-experts.no
```

3. Далее создадим пару ключей, CSR и выпустим сертификат:

```
$ mkdir private
$ chmod 0700 private
$ openssl genpkey \
    -algorithm ED448 \
    -out private/server2_keypair.pem
$ openssl req \
    -config server2.cnf \
    -new \
    -key private/server2_keypair.pem \
    -out server2_csr.pem \
    -text
$ cd ../intermediate/
$ openssl ca \
    -config intermediate.cnf \
    -in ../server2/server2_csr.pem \
    -out ../server2/server2_cert.pem
Using configuration from intermediate.cnf
Check that the request matches the signature
Signature ok
Certificate Details:
          ...
Certificate is to be certified until Jul 19 15:38:44 2023 GMT (365 days)
Sign the certificate? [y/n]:y
1 out of 1 certificate requests certified, commit? [y/n]y
Write out database with 1 new entries
Data Base Updated
```

Мы сгенерировали новый сертификат и все еще находимся в каталоге `intermediate`. Сгенерируем CRL, пока сертификат `server2` еще не отозван.

4. Вот как генерируется CRL для промежуточного УЦ:

```
$ openssl ca \
    -config intermediate.cnf \
    -gencrl \
    -out intermediate_crl.pem
```

5. Посмотрим, как выглядит сгенерированный CRL в текстовом виде:

```
$ openssl crl \
    -in intermediate_crl.pem \
    -noout \
    -text
Certificate Revocation List (CRL):
        Version 2 (0x1)
        Signature Algorithm: ED448
        Issuer: C = NO, ST = Oslo, L = Oslo, O = TLS Experts,
CN = Intermediate CA
        Last Update: Jul 19 16:56:47 2022 GMT
```

```
        Next Update: Aug 18 16:56:47 2022 GMT
        CRL extensions:
            X509v3 Authority Key Identifier:
              ... (шестнадцатеричные значения) ...
            X509v3 CRL Distribution Points:
              Full Name:
                URI:http://crl.tls-experts.no/ intermediate_crl.der
            Authority Information Access:
                OCSP - URI:http://ocsp.tls-experts.no/
            X509v3 CRL Number:
                1
No Revoked Certificates.
    Signature Algorithm: ED448
    Signature Value:
        ... (шестнадцатеричные значения) ...
```

Мы только что сгенерировали свой первый CRL с номером 1. Этот номер был взят из файла `crlnumber.txt`, имя которого было задано в строке `crlnumber = crlnumber.txt` в конфигурационном файле промежуточного УЦ. Этот файл содержит номер следующего CRL в шестнадцатеричном виде. После выпуска CRL подкоманда `openssl ca` увеличивает номер в этом файле на 1.

6. Как видим, в CRL написано, что никакие сертификаты не были отозваны. Теперь отзовем сертификат `server2` и сгенерируем CRL повторно. Вот как отзывается сертификат:

```
$ openssl ca \
    -config intermediate.cnf \
    -revoke ../server2/server2_cert.pem \
    -crl_reason keyCompromise
Using configuration from intermediate.cnf
Revoking Certificate 1651F3139172DEE541B914DFCB371D8E11BA209F.
Data Base Updated
```

Сведения об отзыве хранятся в индексном файле сертификатов, `index.txt`. Состояние сертификата в этом файле изменилось с `V` (действителен) на `R` (отозван).

7. Отозвав сертификат, сгенерируем CRL еще раз:

```
$ openssl ca \
    -config intermediate.cnf \
    -gencrl \
    -out intermediate_crl.pem
```

8. Снова посмотрим на сгенерированный CRL:

```
$ openssl crl \
    -in intermediate_crl.pem \
    -noout \
    -text
Certificate Revocation List (CRL):
        Version 2 (0x1)
        Signature Algorithm: ED448
```

```
        Issuer: C = NO, ST = Oslo, L = Oslo, O = TLS Experts,
CN = Intermediate CA
        Last Update: Jul 19 17:43:49 2022 GMT
        Next Update: Aug 18 17:43:49 2022 GMT
        CRL extensions:
              X509v3 Authority Key Identifier:
                  ... (шестнадцатеричные значениЯ) ...
              X509v3 CRL Distribution Points:
                  Full Name:
                    URI:http://crl.tls-experts.no/ intermediate_crl.der
              Authority Information Access:
                  OCSP - URI:http://ocsp.tls-experts.no/
              X509v3 CRL Number:
                  2
Revoked Certificates:
    Serial Number: 1651F3139172DEE541B914DFCB371D8E11BA209F
        Revocation Date: Jul 19 17:41:39 2022 GMT
        CRL entry extensions:
            X509v3 CRL Reason Code:
                Key Compromise
    Signature Algorithm: ED448
    Signature Value:
        ... (шестнадцатеричные значения) ...
```

Теперь номер CRL равен 2, а сам CRL содержит один отозванный сертификат. Мы заключаем, что отзыв сертификатов и генерирование CRL работают, как положено.

9. Еще одна полезная вещь – преобразование выпущенного CRL из формата **почты повышенной секретности** (Privacy-Enhanced Mail – **PEM**) в формат **особых правил кодирования** (Distinguished Encoding Rules – **DER**), поскольку точки распространения CRL обычно отдают CRL в формате DER:

```
$ openssl crl \
    -in intermediate_crl.pem \
    -out intermediate_crl.der \
    -outform DER
```

Итак, мы научились отзывать сертификаты и генерировать CRL. Как мы знаем, раздача CRL – не единственный способ распространения информации об отзыве сертификатов. Другой метод дает протокол OCSP. О нем мы и поговорим в следующем разделе.

Информирование о состоянии отзыва сертификатов по протоколу OCSP

Ответы по протоколу OCSP должны быть подписаны. OCSP-ответ о сертификате может быть подписан издателем сертификата. Тот же самый издатель

может выпустить другой сертификат для подписания OCSP-запросов. В этом сертификате должно присутствовать поле `OCSPSigning` в расширении X509v3 `extendedKeyUsage`.

Создавая конфигурационный файл промежуточного УЦ, мы включили в него следующую секцию расширений X509v3:

```
[v3_ocsp_cert]
subjectKeyIdentifier = hash
authorityKeyIdentifier = keyid:always, issuer
basicConstraints = critical, CA:FALSE
keyUsage = critical, digitalSignature
extendedKeyUsage = critical, OCSPSigning
crlDistributionPoints = URI:http://crl.tls-experts.no/ intermediate_crl.der
authorityInfoAccess = OCSP;URI:http://ocsp.tls-experts.no/
```

Эта секция поможет нам сгенерировать сертификат для OCSP-ответчика. Создадим его.

1. Как обычно, начнем с подготовки каталога:

   ```
   $ cd mini-ca
   $ mkdir ocsp
   $ cd ocsp
   ```

2. Затем создадим конфигурационный файл `ocsp.cnf` для сертификата OCSP-ответчика:

   ```
   [req]
   Prompt                  = no
   distinguished_name      = distinguished_name_ocsp_cert
   [distinguished_name_ocsp_cert]
   countryName             = NO
   stateOrProvinceName     = Oslo
   localityName            = Oslo
   organizationName        = TLS Experts
   commonName              = OCSP Responder
   ```

3. Воспользовавшись сохраненной конфигурацией, создадим сертификат OCSP-ответчика:

   ```
   $ mkdir private
   $ chmod 0700 private
   $ openssl genpkey \
       -algorithm ED448 \
       -out private/ocsp_keypair.pem
   $ openssl req \
       -config ocsp.cnf \
       -new \
       -key private/ocsp_keypair.pem \
       -out ocsp_csr.pem \
       -text
   $ cd ../intermediate/
   $ openssl ca \
   ```

```
    -config intermediate.cnf \
    -in ../ocsp/ocsp_csr.pem \
    -out ../ocsp/ocsp_cert.pem
Using configuration from intermediate.cnf
Check that the request matches the signature
Signature ok
Certificate Details:
          ...
Certificate is to be certified until Jul 19 17:45:44 2023 GMT (365 days)
Sign the certificate? [y/n]:y
1 out of 1 certificate requests certified, commit? [y/n]y
Write out database with 1 new entries
Data Base Updated
```

Сгенерированный сертификат может использоваться OCSP-ответчиком. Протестируем его. Утилита `openssl` включает подкоманду `openssl ocsp`, которая может играть роль простого OCSP-сервера. Им-то мы и воспользуемся для тестирования. Этот OCSP-сервер берет данные о сроке действия сертификата из индексного файла сертификатов, `index.txt`. Для тестирования, на мой взгляд, очень удобно.

4. Прочитать документацию о подкоманде `openssl ocsp` можно на странице руководства:

```
$ man openssl-ocsp
```

5. Тестовый OCSP-сервер запустим, находясь в каталоге `ocsp`:

```
$ cd ../ocsp/
$ openssl ocsp \
    -port 4480 \
    -index ../intermediate/index.txt \
    -CA ../intermediate/intermediate_cert.pem \
    -rkey private/ocsp_keypair.pem \
    -rsigner ocsp_cert.pem
ACCEPT 0.0.0.0:4480 PID=124271
ocsp: waiting for OCSP client connections...
```

Мы предъявили сертификат OCSP-ответчика и его закрытый ключ подкоманде `openssl ocsp` для подписания OCSP-ответов.

6. Но подкоманда `openssl ocsp` может играть роль не только OCSP-сервера, но и OCSP-клиента. Воспользуемся ей в режиме клиента, чтобы проверить действительность недавно выпущенного сертификата `server`. Для этого нам понадобится открыть другое окно терминала, перейти в нем в каталог `mini-ca/ocsp` и выполнить следующую команду:

```
openssl ocsp \
    -url http://localhost:4480 \
    -sha256 \
    -CAfile ../root/root_cert.pem \
    -issuer ../intermediate/intermediate_cert.pem \
    -cert ../server/server_cert.pem
Response verify OK
```

```
../server/server_cert.pem: good
        This Update: Jul 19 19:59:06 2022 GMT
```

Как видим, OCSP-сервер подтвердил действительность сертификата `server`.

7. Теперь проверим состояние сертификата `server2`, который недавно был выпущен, а затем отозван:

```
$ openssl ocsp \
    -url http://localhost:4480 \
    -sha256 \
    -CAfile ../root/root_cert.pem \
    -issuer ../intermediate/intermediate_cert.pem \
    -cert ../server2/server2_cert.pem
Response verify OK
../server2/server2_cert.pem: revoked
        This Update: Jul 19 20:03:56 2022 GMT
        Reason: keyCompromise
        Revocation Time: Jul 19 17:41:39 2022 GMT
```

OCSP-ответчик сообщил, что сертификат `server2` отозван, как мы и рассчитывали. Мы заключаем, что OCSP-сервер работает правильно.

Мы многое узнали об эксплуатации мини-УЦ и познакомились с многочисленными конфигурационными файлами и командами. Эти файлы и команды, сохраненные в виде скриптов оболочки, находятся на GitHub по адресу https://github.com/PacktPublishing/Demystifying-Cryptography-with-OpenSSL-3/tree/main/Chapter12.

Резюме

В этой главе мы научились эксплуатировать мини-УЦ. Сначала мы объяснили, почему мини-УЦ может оказаться полезен для организации. Мы также узнали о подкоманде `openssl ca` и о создании конфигурационных файлов для нее. После этого мы научились выпускать сертификаты с помощью `openssl ca`, а затем отзывать сертификаты и генерировать CRL. И завершили главу рассказом о том, как выпустить сертификат для OCSP-ответчика и сообщить о состоянии отзыва сертификата по протоколу OCSP с помощью подкоманды `openssl ocsp`. Все это поможет вам настроить и эксплуатировать мини-УЦ, получив полный контроль над PKI в своей организации.

Это была последняя глава. Надеюсь, она вам понравилась, как и вся книга. Надеюсь также, что вы узнали много нового и полезного и что приобретенные знания помогут вам лучше разобраться в криптографии и технологиях защиты сетей, разрабатывать более безопасные приложения и перейти на следующую ступень в своей карьере.

Предметный указатель

Нумерованный

3DES шифр, 39

A

Advanced Encryption Standard (AES), 25, 35, 79
  в режиме счетчика с аутентификацией Галуа с синтетическим вектором инициализации (AES-GCM-SIV), 51
  в режиме счетчика с аутентификацией Галуа (AES-GCM), 29
  обзор шифра, 38
  шифрование и расшифрование в командной строке, 56
Advanced Encryption Standard New Instructions (AES-NI), 39
AES из программы
  расшифрование, 68
  шифрование, 63
ARIA алгоритм, 41
ARIA шифр, 42
ARM, 23
ARMv8.2-SHA, 80
ARMv8 криптографическое расширение, 39, 79
ASN.1, 26, 119, 151

B

BLAKE2, семейство, 83
BLAKE2b, 83, 85
BLAKE2s, 83, 85
BLAKE3, 83
Blowfish шифр, 41
BoringSSL, сравнение с OpenSSL, 30
Botan, сравнение с OpenSSL, 27
Browser Exploit Against SSL/TLS (BEAST), 40, 178
BSD семейство систем, 29

C

C, 27
C++11, 27
C++17, 27
Camellia алгоритм, 41
Camellia шифр, 42
CAST5 шифр, 41
CAST-128, 41
Certificate Status Request, расширение TLS, 218
ChaCha20 шифр, 40
ChaCha потоковый шифр, 83
ChaCha-Poly1305, 29
Common Name (CN), 188
Compression Ratio Info-Leak Made Easy (CRIME), 178
CRYPTO_BUFFER функциональность, 30

D

Data Encryption Standard (DES), 39
Datagram Transport Layer Security (DTLS), 174
Digital Signature Algorithm (DSA), 82, 113
  выбор, 139
  обзор, 135
Distinguished Encoding Rules (DER), формат, 119, 151, 208, 288
Distinguished Name (DN), формат, 151
DTLS протокол, 23

E

ElGamal алгоритм, 114
Elliptic Curve Digital Signature Algorithm (ECDSA), 113
Envelope API, 64
Ethereum, 81
EtM-расширение протокола TLS, 95
EVP API, 86

G

Git, 76
GMP, 26
GNU проект, 25
GNU экосистема, 26
GNU Privacy Guard (GnuPG), 42, 81
GnuTLS, сравнение с OpenSSL, 25

H

Hard Core Library (HCL), 26
HashEdDSA, 138
HMAC-SHA-256, 92
Hypertext Transfer Protocol (HTTP), 173

I

IDEA шифр, 41
Intel SHA расширения, 79
International Data Encryption Algorithm (IDEA), 43
IPsec, 95

K

KangarooTwelve (K12), 80
KECCAK Message Authentication Code (KMAC), 92

L

LibreSSL, сравнение с OpenSSL, 29
Libtasn1, 26

M

MAC, затем шифрование (MtE), схема, 94
MatrixSSL, 28
Mbed TLS, 28
MD4, 82
MD5, 82
MD семейство функций, 82
Mercurial, 76
Modification Detection Code 2 (MDC-2), 84

N

Nettle, 26
Network Security Services (NSS), библиотека, сравнение с OpenSSL, 26

O

OCSP-ответчик, 217
OpenBSD, 29
OpenSSH, 29, 81
OpenSSL 3.0, 24, 25
  доступные виды асимметричного шифрования, 113
  история, 24
  компиляция и компоновка, 61
  компоненты, 55
  особенности, 25
  скачивание и установка, 54
  сравнение с облегченными библиотеками TLS, 27
  сравнение с BoringSSL, 30
  сравнение с Botan, 27
  сравнение с GnuTLS, 25
  сравнение с LibreSSL, 29
  сравнение с NSS, 26
openssl ca подкоманда, эксплуатация мини-УЦ, 271
openssl dgst подкоманда, 95
openssl mac подкоманда, 95

P

Padding Oracle On Downgraded Legacy Encryption (POODLE), 178
PGP, 85
PKCS #12 контейнер, 285
  упаковка клиентских сертификатов, 229

Poly1305, 28
POWER8, 80
Power ISA, 80
Pretty Good Privacy (PGP), 42, 82, 112
Privacy Enhanced Mail (PEM), 119, 151, 288
PureEdDSA, 138
Python, 27

R

RC2, 41
RC4, 40
RC5, 41
RC семейство шифров, 43
Rijndael алгоритм, 39
RIPEMD-160, 84
RSA криптостойкость, 115
RSA обмен ключами, 173
RSA пара ключей, генерирование, 118

S

S-блоки, 42
scrypt алгоритм, 105
Secure Multipurpose Mail Extension (S/MIME), 82
Secure Sockets Layer (SSL), 23
SEED алгоритм, 41
SEED шифр, 42
SHA-0, 81
SHA-1, 81
SHA-2 семейство, 78
SHA3-256, 85
SHA-3 семейство функций хеширования, 79
SHA-256, 77
SHA-512, 79
SHAKE128, 80
SHAKE256, 80
Shang Mi 2 (SM2), 138
SM3, 84
SM4 алгоритм, 41
SM4 шифр, 43
SSH, 85, 95
SSL 3.0, 83
SSLeay библиотека, 24
Subject Alternative Names (SAN), 188
SVE/SVE2, 80

T

TLS, 85, 94
TLS 1.0, 83
TLS 1.1, 83
TLS-квитирование, 174
TLS клиентские сертификаты
  выполнение программы, 233
  генерирование, 228
  загрузка, 237
  запрос, 230
  проверка, 231
  реализация функции построения ответа, 232
TLS на неблокирующих сокетах, 252
TLS на нестандартных сокетах, 258
TLS-подключение, 201
TLS протокол, история, 177
TLS-сокет, 25
TrueCrypt, 84

U

User Datagram Protocol (UDP), 174

V

VeraCrypt, 84

W

Whirlpool, 84
wolfSSL, 28

X

X.509, 81, 85, 150
X.509 сертификат, 22, 75, 172
  выпуск, 156
  инфраструктура открытых ключей (PKI), 159
  поля, 151
XOR (исключающее ИЛИ), 36

А

Абстрактная синтаксическая нотация версии 1, 26, 119

Агентство национальной безопасности (АНБ), 78
Алгоритмы цифровой подписи, поддерживаемые OpenSSL
  обзор, 135
  ECDSA, 136
  EdDSA, 137
  RSA, 135
  SM2, 138
Асимметричного шифрования алгоритмы, 23, 111, 133
Атака
  на основе подобранного открытого текста, 46
  нахождения прообраза, 77
  с удлинением сообщения, 92
  с человеком посередине (MITM), 51, 111, 152, 172, 243
    встреча для подписания ключей, 79, 112
    личная встреча, 112
    проверка цифрового отпечатка ключа по телефону, 112
    разделение ключа, 113
Аутентификационный жетон, 89
Аутентификация сообщения, 90

Б

Базовый объект ввода-вывода (BIO), 183, 251
  источники и стоки, 184
  фильтр, 184
Базовый CRL, 273
Биты стойкости, 38
Блокирующие сокеты, 251
Блокчейн, 76
Блочный шифр
  дополнение, 52
  сравнение с потоковым шифром, 35

В

Вектор инициализации (IV), 35, 175
Встреча для подписания ключей, 112
Входной ключевой материал (IKM), 101
Выходной ключевой материал (OKM), 101

Г

Гамма, 36
Генератор
  псевдослучайных чисел (ГПСЧ), 36
  случайных чисел (ГСЧ), 137
Гибридная схема шифрования, 115
Гибридное шифрование открытым ключом (HPKE), 114
Глубина
  проверки, 202
  сертификата, 202
ГОСТ12, 84
ГОСТ89, 42, 84
ГОСТ94, 84
ГОСТ2012, 84
ГОСТ2015, 42
Графический процессор (GPU), 103

Д

Движковый API, 24
Дельта CRL, 273
Динамическая компоновка, 26
Диффи–Хеллмана метод обмена ключами, 48, 106, 113, 172
Диффи–Хеллмана на эллиптических кривых метод (ECDH), 114
Доказательство выполнения работы, 76
Дополнение PKCS7, 52
  недостаток, 53
Дополнительные данные, 204

З

Закрепление сертификата, 243
Закрытый ключ, 90, 111
Запрос на подписание сертификата (CSR), 157, 182, 245, 275

И

Избирательная атака с подлогом, 90
Имитовставка на основе функции хеширования (HMAC), 74, 91, 175, 228

Имитовставка (MAC), 40, 75, 90, 172
  сравнение с цифровой подписью, 134
Индикация имени сервера (SNI, 176, 263
Инженерный совет интернета (IETF), 178
Интегральные схемы специального назначения (ASIC), 103
Интегрированная схема шифрования (IES), 114
Интернет вещей, 27, 179
Интерфейс прикладного программирования (API), 23, 56, 96, 183
  нестабильность, 31
  привязки, 27
  совместимость, 30
Инфраструктура открытых ключей (PKI), 113, 159, 243
Исчерпывающий поиск, 37

К

Картера–Вегмана + CTR (CWC) режим, 51
Квантовые вычисления, 38, 117
Квитирования, 115
Квитированный секрет, 175
Клиентский сертификат
  генерирование, 283
  упаковка в контейнер PKCS #12, 229
Ключ шифрования, 34
Коллизионная атака, 41, 77
Коллизия, 73
Контекст (CTX), 65
Корневого УЦ конфигурационный файл, 276
Корневого УЦ сертификат, 153
  генерирование, 273
Криптовалюты, 76
Криптографические функции хеширования, 73
  выбор, 85
  оценка стойкости, 77
  свойства, 73
  MDC-2, 84
  RIPEMD-160, 84
  Whirlpool, 84
Криптографически стойкий генератор псевдослучайных чисел (КСГПСЧ), 54
Криптография на эллиптических кривых (ECC), 117
Кузнечик, 42

Л

Лавинный эффект, 73
Листовый сертификат, 153
Лицензии BSD с двумя оговорками, 27
Локальные сети (LAN), 43

М

Магма, 42
Мастер-секрет, 175
Меркла дерево, 83
Мини-УЦ
  подкоманда openssl ca, 271
  эксплуатация, 271

Н

Национальные шифры, 41
Национальный институт стандартов и технологий (NIST), 38, 78
Неблокирующие сокеты, 251
Необратимость, 102
Неотрицаемость, 90
Непрерывная интеграция, 62
Несамоподписанный сертификат, генерирование, 162
Номер CRL, 273
Нулевое оборотное время (0-RTT), 179

О

Облегченные библиотеки TLS, 27
  MatrixSSL, 28
  Mbed TLS, 28
  wolfSSL, 28
Обмен ключами, 23, 95, 115, 172

Одноразовое число, 36
Оппортунистический TLS, 174
Оптимальное дополнение асимметричного шифрования (OAEP), 121
Оракул, 49
Отказ в обслуживании, 76
Открытый или закрытый ключ (PKEY), 118
Открытый ключ, 111
Открытый текст, 34
Отравление DNS, 152

П

Парадокс дней рождения, 41
Пара ключей, 111
Пароль, 100
  сравнение с ключом шифрования, 101
  формирование ключа в командной строке, 105
  формирование ключа из программы, 105
Парольная фраза, 100
Полный перебор, 37
Поразрядный циклический сдвиг, 36
поставщики реализации операций OpenSSL, 24
Потоковый шифр, 36
Поток псевдослучайных цифр шифра, 36
Предварительного хеширования функция, 138
Предварительный мастер-секрет, 175
Принцип криптографической обреченности, 94
Проверка целостности данных, 74
Программируемые пользователем вентильные матрицы (ППВМ), 104
Промежуточных УЦ сертификаты, 153
  генерирование, 278
  причины существования, 154
Протокол онлайновой проверки состояния сертификата (OCSP), 152, 244, 280
  вшивание, 218, 245
  информирование о состоянии отзыва сертификатов, 288
  использование, 217
Псевдослучайная функция (PRF), 95

Р

Рабочая группа по TLS (TLS WG), 178
Расширения X509v3, 159, 289
Режим гаммирования
  с аутентификацией Галуа (GCM), 48
  с имитовставкой (CCM), 51
  с обратной связью по выходу (OFB), 51
  с обратной связью по шифртексту (CFB), 51
  CTR, 47
Режим простой замены
  с зацеплением (CBC), 45
  ECB, 44
Режим работы шифра, 35, 44
  выбор, 52
  режим гаммирования с аутентификацией Галуа (GCM), 48
  режим гаммирования (CTR), 47
  режим простой замены с зацеплением (CBC), 45
  режим простой замены (ECB), 44
  AES-GCM-SIV, 51
Режим CBC с распространением (PCBC), 51
Ривеста–Шамира–Адлемана (RSA) алгоритм, 113, 135
  расшифрование из программы, 129
  шифрование и расшифрование в командной строке, 120

С

Самоподписанные сертификаты, генерирование, 160
Свободное программное обеспечение с открытым исходным кодом (FOSS), 26
Сеансовый ключ, 114
Секретный ключ, 90

Сертификат
генерирование для веб-клиента и почтового клиента, 283
генерирование для веб-сервера, 282
отзыв, 285
Сеть доверия, 26, 113
Симметричная криптография, 23
Симметричного шифрования алгоритмы, 23, 34, 172
Симметричного шифрования ключ, 101
Симметричные криптографические алгоритмы, 38
Симметричные шифры
национальные, 41
поддерживаемые OpenSSL, обзор, 35
семейство RC, 43
Синтетический вектор инициализации (SIV), 51
Совершенная прямая секретность (PFS), 172
Совместная выработка ключа, 173
Соль, 102
Состояние отзыва сертификата, информирование по протоколу OCSP, 288
Софи Жермен режим гаммирования с аутентификацией, 51
Спам, 76
Список отозванных сертификатов (CRL), 152, 208, 244, 272
генерирование, 285
С подтверждением физического лица (IV) сертификаты, 158
С подтверждением юридического лица (OV) сертификаты, 158
С подтвержденным доменом сертификаты, 157
С расширенным подтверждением сертификаты, 158
Стандартное дополнение блока, 52
Стойкость
к полному перебору, 103
криптографических хеш-функций, оценка, 77
симметричного шифра, 37
Стрибог, 84
Счетчик блоков, 41

У

Удостоверяющий центр (УЦ), 139, 153, 175, 181, 208, 243
Универсальная атака с подлогом, 90
Управление исходным кодом (SCM), 76
Уровни совместимости с OpenSSL, 23

Ф

Файловый дескриптор, 184
Функции с удлиняемым результатом (XOF), 80
Функция
вычисления HMAC, 91
вычисления MAC, 90
стойкость, 90
формирования ключа на основе пароля (PBKDF), 101
свойства, 102
формирования ключа на основе HMAC (HKDF), 175
формирования ключа (KDF), 101
параметры, 102
поддерживаемые OpenSSL, 104

Х

Хеш-значение сообщения, 23, 72, 82
блокчейн, 76
вычисление в командной строке, 85
вычисление из программы, 86
доказательство выполнения работы, 76
идентификатор содержимого, 75
как основа HMAC, 74
проверка пароля, 75
проверка целостности данных, 74
сетевые протоколы, 75
цифровые подписи, 75
Хеширование паролей, 83
Хеширования и подписания парадигма, 133

Ц

Центры доверия, 153
Цепочка
  доверия, 152
  проверки сертификатов, 152
  сертификатов, 152
Цифровые подписи, 23, 75, 90
  подписание из программы, 142
  проверка из программы, 145
  свойства, 133
  сравнение с имитовставками, 134

Ч

Частичное обращение хеша, 76

Ш

Шифр, 34
Шифрование
  затем MAC (EtM), схема, 94
  и MAC (E&M), схема, 94
  с аутентификацией и присоединенными данными (AEAD), 49, 174
  с аутентификацией (AE), 49, 94, 172
Шифртекст, 34

Э

Эдвардса скрученные кривые, 137
Экзистенциальная атака
с подлогом, 91
  на основе подобранного сообщения, 91
Эллиптическое шифрование,
генерирование пары ключей, 139
эфемерный протокол
Диффи–Хеллмана (DHE), 172
  на эллиптических кривых (ECDHE), 172

Я

Ядерный TLS (KTLS), 25

Книги издательства «ДМК ПРЕСС»
можно купить оптом и в розницу
в книготорговой компании «Галактика»
(представляет интересы издательств
«ДМК ПРЕСС», «СОЛОН ПРЕСС», «КТК Галактика»).
Адрес: г. Москва, пр. Андропова, 38, оф. 10;
тел.: **(499) 782-38-89**, электронная почта: **books@alians-kniga.ru**.
При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: **http://www.galaktika-dmk.com/**.

Алексей Хлебников

OpenSSL 3. Ключ к тайнам криптографии

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Зам. главного редактора *Сенченкова Е. А.*
Перевод *Логунов А. В.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Гарнитура PT Serif. Печать цифровая.
Усл. печ. л. 24,38. Тираж 200 экз.

Веб-сайт издательства: **www.dmkpress.com**