

Анализ криптографических сетевых протоколов транспортного уровня

Владислав Суфьянов

sufyanov.as1504@asugubkin.ru

Исследованы криптографические сетевые протоколы транспортного уровня с акцентом на TLS/SSL. Рассмотрена логика их работы и последовательность изменений при переходе между версиями. Описаны основные известные на сегодняшний день уязвимости TLS/SSL, а также приведены профилактические меры по их устранению.

Разработан сканер поддерживаемых версий TLS/SSL и с помощью него исследовано более 100000 веб-сайтов. Сделан обзор основных инструментов исследования и библиотек TLS/SSL. Описана инструкция по конфигурации двух самых популярных веб-серверов. Приводятся рекомендации по настройке параметров протокола для обеспечения наивысшей степени безопасности.

Оглавление

Введение.....	3
1 Описание и логика работы протоколов TLS/SSL, DTLS	5
1.1 Общие сведения.....	5
1.2 DTLS (версия 1.2) [2].....	5
1.3 Логика работы TLS	7
1.4 TLS1.3 [5]	20
2 Уязвимости TLS/SSL.....	29
2.1 Модель угроз TLS.....	29
2.2 Атаки на TLS	31
3 Исследование и конфигурация.....	52
3.1 Обзор самых распространенных библиотек TLS/SSL, DTLS	52
3.2 Дополнительные механизмы проверки SSL сертификатов	54
3.3 Обзор и работа с инструментами для исследования TLS/SSL	55
3.4 Сканер, поддерживаемых версий TLS/SSL	57
3.5 Настройка веб-сервера.....	60
3.6 Общие рекомендации.....	65
Заключение.....	67
Список использованных источников.....	68

ВВЕДЕНИЕ

Протокол — это набор информационных сообщений определенного формата, которыми обмениваются два устройства или две программы, а также набор правил, определяющих логику обмена этими сообщениями [7, с. 41]. С точки зрения защиты информации существует несколько фундаментальных задач: обеспечение доступности информации, конфиденциальности, целостности и подлинности соединения. Криптографические протоколы призваны решать их с помощью криптографических алгоритмов. В данной работе рассмотрены протоколы защиты транспортного уровня, функции которых с точки зрения стека OSI можно разделить между двумя уровнями: сеансовым и представления.

Доминирующей технологией в данной области является SSL и TLS. Аббревиатура TLS появилась в качестве замены обозначения SSL после того, как протокол стал интернет-стандартом. Такая замена вызвана юридическими аспектами, так как спецификация SSL изначально принадлежала компании Netscape [1]. Изначально необходимость в TLS/SSL возникла из-за коммерческих транзакций, проводимых через интернет, и которые необходимо было каким-то образом защищать. В современных реалиях приложения данной технологии используются повсеместно, а самым распространенным вариантом является HTTPS, который стремительно вытесняет незащищенную версию HTTP. HTTPS и HTTP это протоколы прикладного уровня, являющиеся основой веба, и с которыми люди взаимодействуют каждый день, а это значит, что огромное количество личных данных пользователей может оказаться под угрозой, если не обеспечить достаточный уровень безопасности.

TLS решает проблемы конфиденциальности, целостности и подлинности (задача аутентификации) соединения. Из-за массовости использования, данная технология довольно хорошо изучена, но не всегда от этого есть практический толк и часто уязвимости исправляются только спустя большой промежуток времени и хорошо если проблема будет заключаться только в реализации, а не в самом протоколе, так как обратное может привести

к более тяжелым последствиям. Также большое число пользователей порождает другую проблему, связанную с снижением уровня безопасности в угоду совместимости, а именно использование устаревших, уязвимых версий протокола. Так на просторах интернета можно встретить сервисы поддерживающие протоколы SSLv2 (не имеет спецификации RFC) и SSLv3 (развитие SSLv2 с существенными доработками, имеет спецификацию RFC, но скорее в качестве исторического документа, так как с 2015 согласно RFC 7568 данный протокол следует исключить из разряда поддерживаемых клиентами и серверами), которые не должны использоваться в наше время, так как они не способны решать возложенные на них функции.

На данный момент существует четыре версии TLS: TLS1.0, TLS1.1, TLS1.2, TLS1.3, все они описаны в RFC. RFC содержит технические спецификации и стандарты Интернета. TLS1.3 является самой современной версией, вышедшей в 2018 году. Версии до 1.3 не имеют фундаментальных различий, а отличаются часто в небольших деталях (которые при этом не перестают быть очень важными), а вот версия 1.3 существенно отличается от предшественников и дает совершенно другой уровень обслуживания не только с точки зрения безопасности, но и с точки зрения скорости работы.

Основными целями ВКР является исследование TLS1.3, а также более ранних версий, для выявления последовательности и логики изменений, разбор основных известных на сегодняшний день уязвимостей, разработка программы для сканирования веб-сайтов на наличие поддержки различных версий TLS/SSL и анализ, полученных результатов, обзор инструментов для изучения и углубления знаний данной технологии, а также выработка рекомендаций по настройке параметров протокола для обеспечения наивысшей степени безопасности.

1 ОПИСАНИЕ И ЛОГИКА РАБОТЫ ПРОТОКОЛОВ TLS/SSL, DTLS

1.1 Общие сведения

TLS ставит своей целью создание защищенного от прослушивания (конфиденциальность) и замены передаваемых данных (целостность) канала связи между клиентом и сервером (протокол использует модель клиент/сервер), который подходит для передачи произвольных данных в двух направлениях, а также обеспечивает проверку того, что информация передается именно тем узлам, для которых и создавался канал (аутентификация).

Предполагается, что TLS работает поверх существующего между узлами "поточкового" соединения (с которым обычно связан "сокет" - откуда название SSL, Secure Sockets Layer) [1]. В большинстве случаев в качестве транспорта TLS использует TCP, который как известно обеспечивает надежную доставку сообщений, а также сохранение порядка их следования, но при этом никак не противодействует возможности прочесть или подменить данные третьей стороной (эту задачу как раз и решает TLS). Тот уровень обслуживания, который предоставляет TCP и требует TLS, без надежного канала протокол просто не будет корректно работать. TCP можно рассматривать, как потоковый псевдосокет, в который данные просто записываются как есть, а дальнейшая манипуляция с ними (разрезание на блоки, контроль за надежностью передачи их по сети) как раз и ложится на плечи протокола транспортного уровня. Но при этом существует разновидность TLS, работающая поверх UDP (не обеспечивает ни надежность передачи, ни порядок следования сообщений), и называется она DTLS. Рассмотрим данный протокол более подробно.

1.2 DTLS (версия 1.2) [2]

Данный протокол предназначен для защиты данных приложений. DTLS также, как и UDP не предоставляет сервиса для надежной доставки данных полезной нагрузки (т. е. не происходит компенсация потерянному или переупорядоченному трафика). Дейтаграммы используются в качестве транспорта приложениями чувствительными к задержкам трафика, но при

этом потеря небольшой части данных не приводит к значительным последствиям в их нормальной работе, примером могут выступать потоковые мультимедиа сервисы, интернет телефония и онлайн-игры. Поэтому для данных приложений полностью подходит DTLS. Так как TLS не способен работать с UDP, то целью DTLS является проведение минимальных изменений в TLS, которые необходимы для решения этой проблемы.

Проблемы, возникающие у TLS при работе с UDP:

1. Невозможность проверки целостности при потере или нарушении порядка следования (возможно также дублирование) сообщений.
2. Установление соединения (handshake) требует надежной доставки, а в случае потери данных происходит разрыв.

Первую проблему DTLS решает путем запрета потоковых шифров и добавлением порядковых номеров записей, а вторую с помощью таймеров повторной передачи и фрагментации данных, чтобы влезть в одну дейтаграмму. После отправки handshake сообщений устанавливается таймер и если следующее, ожидаемое сообщение не получено, то инициируется повторная передача (рис. 1). А сохранение информации о полученных ранее пакетах, позволяет отбрасывать дубликаты.

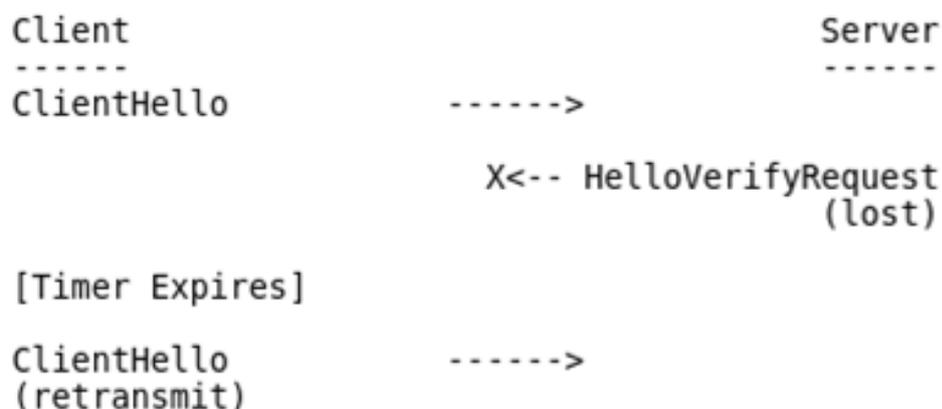


Рис. 1. Демонстрация использования таймеров повторной передачи. RFC

6347 [2, с. 6]

1.3 Логика работы TLS

TLS можно разбить на два уровня: уровень записей (records) и уровень сообщений (являющихся подпротоколами более высокого уровня чем records). Записи находятся в фундаменте иерархии протокола. Это нижний транспортный уровень TLS. Т. е. запись представляет из себя блок, состоящий из заголовка и данных. Спецификация позволяет сообщения верхнего уровня разбивать между несколькими записями, а также одна запись может содержать несколько сообщений. В TLS используется сетевой порядок байт (старший байт идет первым).

Размер заголовка записи составляет 5 байт:

Тип (1 байт)

Версия протокола (2 байта)

Длина данных (2 байта)

Тип — определяет тип записи. Существует четыре типа: 20 (0x14) - сообщение Change Cipher Spec (CCS), в версии 1.3 данное сообщение передаётся только как "фиктивное", и не имеет логической роли в самом протоколе; 21 (0x15) - сообщение Alert; 22 (0x16) - сообщение Handshake (установление соединения); 23 (0x17) - Application Data (запись содержит данные приложения - то есть, полезную нагрузку); трактовка типа Application Data в TLS1.3 существенно отличается от предыдущих версий. До версии 1.3 записи, отличные от Application Data, обычно передаются в открытом виде, но тоже могут быть зашифрованы, в зависимости от текущего состояния TLS-соединения [1].

Версия протокола — собственно и отражает, используемую версию: 03 00 - это SSLv3, 03 01 - TLS 1.0; 03 02 - TLS 1.1; 03 03 - TLS 1.2. Но в TLS 1.3 в это поле записывается значение 03 03, что соответствует TLS1.2, и при этом оно игнорируется, а указателем на использование версии 1.3 служат другие дополнительные поля.

Длина данных — количество байт, содержащих данные, которые следуют сразу после заголовка. Максимальная допустимая длина данных

записи в TLS 1.3 - 16640 байт ($2^{14} + (256 \text{ байт под MAC})$). В записях до версии 1.3 максимальная длина данных = 18432 байта, т. е. 16 килобайт + 2 килобайта. 2 килобайта зарезервированы под код аутентификации сообщения (MAC) и под возможное увеличение длины после сжатия (в TLS1.3 сжатие не используется, впрочем, и в предыдущих версиях TLS, из-за найденных уязвимостей, в качестве алгоритма сжатия обычно указывают null, который означает, что никакого сжатия на самом деле не будет).

Пример заголовка TLS1.2 (в шестнадцатеричной записи), полученный при сканировании веб-сайта:

```
16 03 03 00 3f
```

Раскодируем данный заголовок: сообщение типа Handshake (0x16); версия 3.3 (0x03, 0x03) - TLS 1.2; длина данных - 0x003f байт (63 байта).

Заголовок всегда передается в открытом виде. MAC в записях необходим для проверки целостности. В RFC используется понятие HMAC, которое означает тот же самый код аутентификации сообщения, но вычисленный с помощью криптографической хеш-функции, например SHA-1, SHA-256, SHA-512. Хеш-функция в качестве входа принимает блок байт (например сообщение), а на выходе выдает тоже блок байт, но определенной длины. Особенность работы данных алгоритмов заключается в том, что для одного и того же входного объекта будет получаться одинаковый результат, но при этом по нему вычислительно сложно определить исходный объект. Хорошие криптографические хеш-функции устойчивы к коллизиям первого (подбор объекта по результату работы алгоритма с той же самой хеш-суммой) и второго рода (поиск двух объектов с одинаковой хеш-суммой). Соответственно получатель сможет удостовериться, что сообщение не было изменено, посчитав для него хеш-сумму, и сравнив его с HMAC. Такой подход применялся в версиях до 1.3 (наряду и с другими, так как позже также был добавлен AEAD), в современном варианте используются шифры в режиме аутентифицированного шифрования (а именно — AEAD). Данный режим не нуждается в отдельном MAC, так как сам шифр гарантирует целостность

данных. Пример: блочный шифр AES в режиме GCM. Причем AEAD режим устраняет один из дефектов протокола, а именно, до версии 1.3 спецификация предполагала сначала вычисление MAC открытого текста, а затем шифрование всего (текст + MAC) вместе. Данный подход приводил к возможности использования так называемых «криптографических оракулов», определенных знаков от криптосистемы, позволяющих третьей стороне (злоумышленнику) расшифровать переданное сообщение. Это является возможным, так как происходит утечка информации при работе криптосистемы. Для решения данной проблемы еще до версии 1.3 было введено специальное поле расширения `encrypt_then_mac`, которое позволяет узлам договориться о порядке выполнения шифрования и вычисления MAC. AEAD режимы не нуждаются в отдельном MAC, так как на уровне шифра обеспечивается целостность данных.

Используемые криптосистемы в TLS объединяются в типовые шифронаборы (Cipher Suites). Поэтому в ходе handshake узлы должны сначала договориться о используемых шифронаборах. В TLS 1.2 и младше в шифронабор входили: криптосистема, используемая для аутентификации сервера и обмена сеансовым секретом; шифр, который послужит для защиты передаваемых данных; хеш-функция, являющаяся основой для HMAC. В TLS 1.3 криптосистемы, служащие для аутентификации узлов и получения общего секрета, отделены от, собственно, шифров, что потребовало изменения организации реестра и механизма нумерации шифронаборов (а каждый из них обозначается двухбайтовым индексом).

Рассмотрим типовые шифронаборы:

В TLS1.0: `TLS_RSA_WITH_AES_256_CBC_SHA` (0x0035) — для передачи сеансового секрета будет использоваться RSA, AES с 256-битным ключом в режиме CBC в качестве симметричного шифра, SHA-1 в качестве хеш-функции HMAC.

В TLS1.3: `TLS_CHACHA20_POLY1305_SHA256` (0x1303) — для защиты данных будет использоваться потоковый шифр ChaCha20, со схемой

аутентификации Poly1305, а в качестве хеш-функции применяется SHA256 (в том числе и для генерации ключей).

1.3.1 Установление соединения (handshake)

Для того, чтобы понять каким образом работает handshake в TLS1.3, потребуется понимание этого процесса в более ранних версиях. В связи с этим сначала будет рассмотрен вариант до 1.3 (рис. 2), но возможно с некоторыми комментариями, касающимися TLS1.3, а затем будет описан вариант самой современной версии на данный момент.

В процессе установления соединения клиент и сервер должны договориться о используемых шифрах, методах аутентификации, согласовать сеансовые ключи и т. д. Полный набор параметров, которые будут согласованы называется криптографическим контекстом.

Протокол TLS Handshake (до 1.3) включает в себя следующие этапы (RFC 5246 [3]): 1) обмен приветственными сообщениями для согласования алгоритмов, обмена случайными значениями, и проверки возможности возобновления сеанса; 2) обмен необходимыми криптографическими параметрами, чтобы клиент и сервер смогли согласовать общий секрет (так называемый *premaster secret*); 3) при необходимости обменяться сертификатами и криптографической информацией, чтобы произвести взаимную аутентификацию; 4) сгенерировать мастер-секрет (*master secret*) из *premaster secret* и случайных значений в приветствии (*ClientRandom*, *ServerRandom*), на основе мастер-секрета и будут вычислены сеансовые ключи и необходимые данные для шифров; 5) предоставить параметры безопасности уровню записей; 6) позволить клиенту и серверу удостовериться, что параметры соединения были согласованы без вмешательства злоумышленника.

Пояснение к рисунку 2: * - обозначены необязательные сообщения; в квадратных скобках указано сообщение, которое не входит в handshake, а принадлежит к отдельному типу - Change Cipher Spec (CCS).

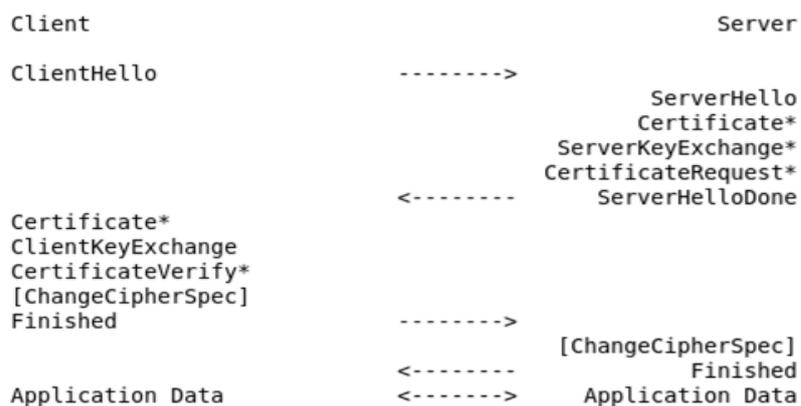


Рис. 2. Полный набор сообщений при установлении соединения до версии TLS1.3 RFC 5246 [3, с. 36]

Первым handshake сообщением в протоколе TLS является ClientHello. Поля данного сообщения: 1) версия протокола (Version) - максимальная версия, которую готов поддерживать клиент (в TLS1.3 игнорируется); 2) 32 байта случайных значений - ClientRandom. В старых версиях первые 4 байта могут быть таймстемпом. Но обычно все байты являются результатом работы генератора псевдослучайных чисел, такой подход закреплён в TLS1.3; 3) идентификатор TLS-сессии - SessionID: TLS позволяет возобновлять ранее установленные сессии, используя сокращённый вариант протокола установления соединения. Так до версии 1.3 клиент мог передать id сессии, о которой клиент и сервер договорились ранее и если сервер сохранил свой криптографический контекст, то соединение может быть возобновлено, если у клиента нет контекста для возобновления сессии, то он отправляет пустое значение, а в качестве длины указывается 0. В случае когда в TLS1.3 данное поле содержит данные, то оно выполняет другую роль: обнаружив в ClientHello поле SessionID, сервер включает его значение в свой ответ без изменений, при этом, как номер сессии - SessionID в 1.3 не используется, но может служить инструментом имитации сессий версии 1.2; 4) список шифронаборов (Cipher Suites), которые поддерживает клиент. Порядок шифронаборов в списке отражает их степень предпочтения клиентом (предпочтительные передаются первыми); 5) список поддерживаемых методов

сжатия - Compression Methods, обычно в этом поле лишь одно значение — null, в TLS1.3 данное поле сохранено по историческим причинам; б) данные нескольких расширений протокола, сюда входит большое количество важных полей, которые позволяют сторонам договориться например о формате, в котором будут передаваться точки эллиптической кривой (ec_point_formats), так как из-за особенностей возможны варианты с использованием сжатия и передачей вместо двух координат только одной, а для другой просто знак, а также расширение — server_name, делающее возможным расположение нескольких web-сайтов на одном ip адресе, внутренняя адресация основана на данном поле и т. д. Расширения появились позже того, как формат сообщения был закреплён. Они вносили коррективы, исправляющие недоработки, и добавляли некоторую новую функциональность. Если говорить о формате сообщений handshake, то стандартным полям переменной длины предшествует значение длины данных, а для обозначения полей расширения сначала указывается имя расширения, потом длина и затем соответственно данные.

После отправки ClientHello, клиент ожидает либо ответное сообщение ServerHello, если все прошло успешно, либо сообщение Alert в случае возникновения ошибок. Alert — это небольшие сообщения, содержащие информацию об уровне ошибки и о её типе.

Поля ServerHello: 1) версия протокола (Version), выбранная сервером для взаимодействия с клиентом; 2) 32 байта случайных значений - ServerRandom. С этой строкой ситуация такая же, как и с Client Random: первые четыре байта могут быть таймстемпом, а могут и не быть (в 1.3 последние 8 байт могут использоваться для защиты от downgrade attack); 3) идентификатор сессии - SessionID, присвоенный новой сессии сервером (в случае TLS 1.3 - данное поле повторяет SessionID клиента); 4) выбранный сервером шифронабор - Cipher Suite, будет использоваться в дальнейшем и клиентом, и сервером. Выбор основывается на вариантах, отправленных клиентом; 5) выбранный

сервером метод сжатия (Compression Method) - обычно это null; 6) набор расширений.

Сообщения handshake (до передачи Change Cipher Spec) не содержат никакой секретной информации, соответственно - передаются в открытом виде (в TLS 1.3 ситуация поменялась - практически сразу сообщения Handshake передаются в зашифрованном виде, тем не менее, ServerHello и ряд расширений всё равно открыты). Данные сообщения, особенно - ClientHello, повсеместно используются в системах DPI для обнаружения факта установления (или попытки установления) TLS-соединения.

В рамках каждой сессии TLS, клиент и сервер согласовывают следующие параметры (RFC 5246 [3]): 1) роли узлов: кто сервер, а кто клиент (сессию начинает клиент); 2) алгоритм PRF (псевдослучайная функция, PseudoRandom Function) - применяется для генерации мастер-секрета на основе premaster secret, ClientRandom и ServerRandom; 3) алгоритм шифрования, который будет применяться для шифрования данных внутри сессии, клиент и сервер согласуют тип шифра (поточковый, блочный), сам шифр, а также режим использования шифра, размер ключа, размер блока (для блочных шифров), инициализирующие векторы, а также дополнительные параметры (если они требуются, например, в случае с AEAD-режимами); 4) алгоритм вычисления кода аутентификации сообщения (MAC), для AEAD-режимов шифрования MAC не нужен; 5) алгоритм сжатия; 6) основной общий секрет (master secret) - массив из 48 секретных байтов, известный серверу и клиенту. Он необходим для генерации сеансовых ключей. Если третья сторона узнает master secret, то на основе его легко сможет вычислить сеансовые ключи и расшифровать весь трафик. В TLS 1.3 используется более сложная схема, она состоит из секретов разных уровней, которые соответствуют этапам работы протокола; 7) случайные данные клиента (ClientRandom) - 32 байта случайных значений, передаются в ClientHello; 8) случайные данные сервера (ServerRandom) - 32 байта случайных значений, передаются в ServerHello.

Данные параметры определяют контекст работы криптосистем, которые будут обрабатывать защищённые TLS-записи на стороне сервера и клиента. Помимо ClientHello и ServerHello, установление соединения подразумевает обмен несколькими другими сообщениями. После отправки ServerHello, со стороны сервера следует набор сообщений, состав и содержание которых зависят от выбранного сервером режима работы, так как многие из них являются необязательными (см. рис. 2).

Certificate — сообщение, которое содержит публичный ключ сервера, информацию о его владельце, сроки валидности сертификата, доменное имя, для которого он был выдан и т. д., плюс подпись центра сертификации, а также цепочку сертификатов удостоверяющих центров (УЦ). Целью сертификата является сопоставление открытого ключа и сетевого имени для защиты от атаки Man-in-the-middle (человек посередине). С помощью него пользователь может удостовериться в подлинности сервера. Открытому ключу соответствует закрытый приватный ключ сервера, именно на основе обладания им и выполняется аутентификация. Проверку владения данным ключом берет на себя центр сертификации, который с помощью своего закрытого ключа ставит подпись на открытом ключе сервера. Но здесь возникает проблема, чтобы проверить подпись необходимо иметь открытый ключ УЦ. Для ее решения существует иерархия удостоверяющих центров, корневые выдают сертификаты, разрешающие другим УЦ выпуск сертификатов, те в свою очередь дают право промежуточным центрам выдавать сертификаты конечным пользователям (описана упрощенная схема). И в итоге пользователю необходимо доверять только корневым центрам, а остальные сертификаты проверять. Открытые ключи корневых УЦ вшиты в OS и ПО (например, в браузеры). Сертификаты являются очень важной частью современного интернета, так как они не позволяют третьей стороне выдавать себя за другой узел. Однако в TLS возможен анонимный вариант, при котором не происходит аутентификация, но в вебе такую ситуацию вряд ли можно встретить. Одной из слабостей TLS до версии 1.3, если не выбран вариант с передачей общего

секрета с помощью открытого ключа сервера (что тоже является плохой практикой) выступает практически отсутствие механизма, позволяющего клиенту проверить, что сервер действительно обладает закрытым ключом (а не кто-то другой просто отправляет сертификат), кроме подписи на параметрах, используемых для обмена общим секретом. Очень важно уточнить, что сейчас самыми распространенными являются RSA и ECDSA сертификаты электронной подписи.

ServerKeyExchange. Это сообщение, содержащее серверную часть данных, необходимых для генерации общего сеансового ключа. Сообщение может отсутствовать, а в TLS 1.3 не используется, так как новая схема Handshake применяет для обмена криптографическими параметрами расширения Hello-сообщений. Данные, которые передает это сообщение обычно представляют собой параметры протокола Диффи-Хеллмана (DH). Если используется классический вариант DH, то в сообщении ServerKeyExchange передается значение модуля и серверный открытый ключ. В варианте на эллиптических кривых (ECDH) - идентификатор самой кривой и, аналогично DH, открытый ключ сервера. Параметры подписываются сервером (за исключением экзотических вариантов обмена, вроде анонимного DH), с помощью открытого ключа из TLS-сертификата клиент может проверить подпись. Подпись необходима для защиты от подмены данных параметров, так как в противном случае по ним могут быть легко вычислены сеансовые ключи. В зависимости от используемой криптосистемы, подпись может быть DSA (сейчас практически не встречается), RSA или ECDSA. В TLS 1.0, 1.1 (при использовании RSA, что является основным вариантом) — подпись вычисляется от объединения значений двух хеш-функций - MD5 и SHA-1, взятых от параметров обмена DH. В TLS 1.2 - от значения хеш-функции, заданной в используемом шифронаборе, вычисленного от объединения ServerRandom, ClientRandom и параметров обмена DH (ECDH).

CertificateRequest. Позволяет серверу запросить клиентский сертификат, который позволит аутентифицировать пользователя. Чаще всего клиентские

сертификаты используются при доступе к банковским, платёжным системам, к корпоративным веб-шлюзам различного назначения и т. д.

`ServerHelloDone`. В TLS 1.2 и раньше — это сообщение обозначает окончание передачи сообщений сервером, которые начались с `ServerHello`. Сообщение имеет нулевую длину, а служит лишь флагом о том, что сервер передал начальную часть данных и ждет ответ от клиента.

После клиент должен ответить своим набором сообщений:

`Certificate` — сообщение, содержащее клиентский сертификат, если он был запрошен сервером, с помощью `CertificateRequest`. В некоторых случаях данный сертификат может заменить пару «логин/пароль» для аутентификации в онлайн-ресурсе.

`ClientKeyExchange` — клиентская часть обмена данными, которые позволяют узлам получить общий сеансовый ключ. Содержательная часть данного сообщения зависит от выбранного шифронабора. В TLS до 1.3 существуют два основных варианта - RSA и несколько разновидностей протокола Диффи-Хеллмана. При использовании RSA, клиент генерирует 48-байтовый случайный секрет (при этом первые два байта содержат версию используемого протокола), зашифровывает его открытым RSA-ключом сервера (этот ключ передаётся в составе серверного TLS-сертификата или в сообщении `ServerKeyExchange`) и передаёт данные на сервер. Сервер может расшифровать значение, используя соответствующий секретный ключ. Данная схема запрещена в TLS 1.3, да и для других версий является скорее исторической. Дело в том, что она имеет некоторые недостатки. А именно, передача секрета по каналу связи является плохой практикой, так как если закрытый ключ сервера станет известен третьей стороне, то она сможет расшифровать весь записанный трафик данной сессии. Современный метод - использование протокола Диффи-Хеллмана. В этом случае, `ClientKeyExchange` содержит открытый ключ ДН.

`CertificateVerify`. Если после запроса сервера (`CertificateRequest`) клиент отправил свой сертификат, то он передает специальное сообщение,

содержащее подпись всех ранее переданных и полученных сообщений handshake. Данный механизм позволяет серверу удостовериться в том, что клиент действительно обладает закрытым ключом соответствующего сертификата. В версии 1.3 аналогичное сообщение отправляет и сервер.

Следом за сообщением ClientKeyExchange, если оно было единственным, либо за CertificateVerify, клиент должен передать сообщение ChangeCipherSpec. Данное сообщение не является членом handshake. В TLS1.3 данное сообщение игнорируется и передается только для маскировки соединения под предыдущие версии. ChangeCipherSpec является сигналом о том, что с данного момента клиент переходит на выбранный шифр и все последующие TLS-записи будут зашифрованы.

Со стороны клиента установление соединения завершается отправкой сообщения Finished. Так как оно передается после ChangeCipherSpec, то является зашифрованным. А также содержит отпечаток (хеш-сумму) от всех предыдущих сообщений handshake, который сервер может проверить. Таким образом подтверждается подлинность сообщений и, соответственно, оказываются криптографически защищены выбранный шифронабор и другие параметры сессии. В случае успешного установления соединения сервер отвечает своей парой сообщений ChangeCipherSpec и Finished (позволяет клиенту проверить целостность всего handshake).

Третьей стороне, активно перехватывающей канал связи, чтобы успешно произвести подмену сообщений handshake, необходимо вычислить сеансовый ключ и другие секретные параметры, после чего сфабриковать корректное сообщение Finished. Это вычислительно трудная задача, обычно неразрешимая на практике, если выбраны правильные настройки протокола и корректная реализация. Однако, в случае использования нестойких шифронаборов (примером являются экспортные варианты), атакующий может на лету вычислить секретный сеансовый ключ, благодаря раскрытию premaster secret, и - успешно подделать сообщения Finished, став посредником в соединении. После обмена между клиентом и сервером парами ChangeCipherSpec и

Finished, защищённое соединение считается успешно установленным и данные могут передаваться в защищённой форме.

В качестве симметричных ключей шифрования сессий, используются пары, один ключ для чтения, другой для записи, т. е. клиентскому read соответствует серверный write (это сделано для устранения проблем с некоторыми, используемыми симметричными алгоритмами, при применении одного ключа и для записи, и для чтения на обоих узлах).

TLS никак не скрывает сам факт установления соединения, а также некоторую метаинформацию о своем состоянии, что может быть использовано в системах DPI (по крайней мере до версии 1.3, которая стала более скрытной). При этом в версиях до 1.3 клиент может инициировать новую сессию в любой момент передав ClientHello (в TLS1.3 от этого отказались), в данном случае сообщения handshake будут зашифрованы.

1.3.2 Сокращенный вариант handshake до версии 1.3

Так как handshake требует немалых временных затрат, а именно до 1.3 - 2 RTT (round trip time), то была введена возможность сокращенного установления соединения (рис. 3).



Рис. 3. Набор сообщений при проведении сокращенного варианта handshake.

RFC5246 [3, с. 37]

Рассмотрим вариант, основанный на применении SessionID. Клиент и сервер какое-то время хранят криптографический контекст, а в случае опправки клиентом валидного значения id сессии, используют его для возобновления сеанса связи. В связи с этим на сервере возникает проблема, заключающаяся в хранении контекста сессий.

Существует еще один механизм, который немного разгружает сервер. Он основан на применении тикета сессии (TLS Session Ticket) — расширение, содержащее серверный криптографический контекст, может быть отправлено клиентом для возобновления соединения.

1.3.3 Отличия TLS1.0 от TLS1.1

TLS1.1 не сильно отличается от версии 1.0 и содержит небольшие улучшения безопасности. Основные изменения [4, с. 5]:

- Неявный вектор инициализации (IV) заменяется на явный IV для защиты от атак на CBC. В версии до 1.1 в качестве вектора инициализации использовался шифротекст последнего блока предыдущей записи.
- Обработка ошибок заполнения изменена на использование предупреждения `bad_record_mac`, а не `decryption_failed` для защиты от padding атак не режим CBC.
- Определены параметры протокола в реестре IANA.
- Преждевременное закрытие больше не приводит к невозможности восстановления сеанса.

1.3.4 Отличия TLS1.1 от TLS1.2

Основные изменения [3, с. 5-6]:

- Комбинация MD5 и SHA-1 в псевдослучайной функции (PRF) была заменена на PRF, которая определяется шифронабором. Все комплекты шифров в RFC 5246 используют P_SHA256.
- Комбинация MD5 и SHA-1, как часть цифровой подписи была заменена одним хешем. Подписанные элементы теперь включают поле, которое явно указывает используемый алгоритм хеширования.
- Добавление поддержки аутентифицированного шифрования.
- После `Certificate_request`, если нет доступных сертификатов, клиенты должны отправлять пустой список сертификатов.
- Добавлены наборы шифров HMAC-SHA256.

- Удалены наборы шифров, включающие устаревшие алгоритмы IDEA и DES.

1.4 TLS1.3 [5]

Так как TLS1.3 очень сильно отличается от предыдущих версий, то мы разберем его более подробно.

Основные отличия от TLS1.2:

- Из списка поддерживаемых алгоритмов симметричного шифрования были исключены все алгоритмы, которые считаются устаревшими. Остались только алгоритмы аутентифицированного шифрования, а именно AEAD алгоритмы. Концепция шифронаборов была изменена, так отделены механизмы аутентификации и обмена ключами от алгоритма защиты записи, а также хеш должен использоваться как с функцией генерации ключей, так и для установки кода аутентификации сообщений handshake (MAC).
- Добавлен режим 0-RTT.
- Статические наборы шифров RSA и Diffie-Hellman были удалены.
- Все сообщения после ServerHello теперь зашифрованы. Введенное сообщение EncryptedExtensions позволяет различным расширениям, которые раньше отправлялись в открытом виде в ServerHello, также пользоваться защитой конфиденциальности.
- Конечный автомат handshake был значительно реструктурирован в сторону большей последовательности и удаления лишних сообщений, таких как ChangeCipherSpec (кроме случаев, когда оно необходимо для совместимости).
- Алгоритмы, которые используют эллиптические кривые теперь в базовой спецификации, также добавлены новые алгоритмы подписи, такие как EdDSA. Определено единое представление при передаче формата точек эллиптических кривых. Для всех кривых закреплено значение генератора.

- Механизм согласования версии TLS1.2 считается устаревшим, применяются списки версий в расширении. Что улучшает совместимость с существующими серверами, в которых неправильно реализовано согласование версии.
- Возобновление сеанса с серверным состоянием и без него, а также основанные на PSK наборы шифров более ранних версий TLS были заменена единой новой PSK exchange.

Список шифронаборов TLS1.3: TLS_AES_128_GCM_SHA256, TLS_AES_256_GCM_SHA384, TLS_CHACHA20_POLY1305_SHA256, TLS_AES_128_CCM_SHA256, TLS_AES_128_CCM_8_SHA256. Для обмена общим секретом используется DHE или ECDHE.

TLS1.3 от предыдущих версий унаследовал базовые принципы установления соединения: сохранилась роль узлов (соединение инициирует клиент), не изменилась последовательность ClientHello и ServerHello, присутствует сообщение-сигнал Finished. Однако на этом сходство заканчивается. Сокращено общее число сообщений, а также сообщений, которые передаются в открытом виде, так как узлы практически сразу переходят на зашифрованное соединение (рис. 4).

Удалён сигнал ChangeCipherSpec - он больше не требуется, но при этом продолжает фиктивно использоваться. Теперь сервер передает сообщение CertificateVerify, которое удостоверяет сообщения первой части handshake, а также позволяет клиенту удостовериться, что сервер действительно обладает закрытым ключом, соответствующим предъявленному сертификату.

В TLS 1.3 появился новый механизм опознавания используемой версии протокола. Поля, содержавшие номер версии до 1.3, сохранены в статусе "исторических", их значения зафиксированы и не используются. Версия же 1.3 передаётся в специальном расширении сообщений ClientHello и ServerHello. Это расширение называется supported_versions, а его наличие является одним из признаков того, что клиент (или сервер) будут использовать версию TLS 1.3. Со стороны клиента, в ClientHello, supported_versions содержит список версий

протокола, которые готов поддерживать клиент. Сервер передаёт в этом расширении номер выбранной версии.

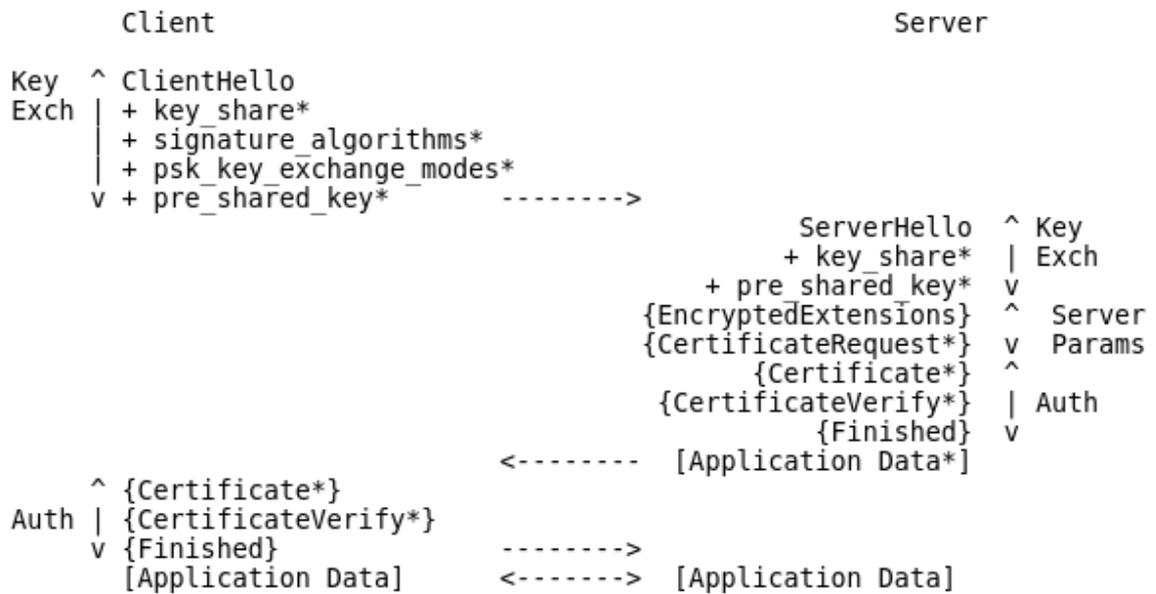


Рис. 4. Поток сообщений полного handshake TLS1.3. RFC 8446 [5]

+ Указывает важные расширения, входящие в отмеченное сообщение.

* Обозначает необязательные или зависимый от ситуации сообщения/расширения, которые не всегда отправляются.

{ } Выделяют сообщения, защищенные ключами полученными из [sender]_handshake_traffic_secret.

[] Выделяют сообщения, защищенные ключами полученными из [sender]_application_traffic_secret_N.

В протоколе Handshake TLS 1.3 выделяются три фазы (рис. 4): выработка начального значения общего криптографического секрета (ключей), определение параметров соединения, аутентификация (сервера и клиента).

В первую фазу входят ClientHello и ServerHello, в результате обмена данными сообщениями узлы согласовывают начальный секрет (handshake_traffic_secret), на основе которого и сообщений, уже переданных к этому моменту, клиент и сервер генерируют первый набор сеансовых ключей. Благодаря этому сообщение EncryptedExtension, следующее сразу после ServerHello, передается в зашифрованном виде. Это становится возможным в связи с расширением key_share, в котором клиент передает свои открытые

значения алгоритма Диффи-Хеллмана (т. к. генератор уже закреплён спецификацией и сервер его не выбирает, а вот выбор группы клиент берет на себя, но сервер может с ним не согласиться и потребовать рассчитать значение для другой группы, ответив сообщением HelloRetryRequest).

Аналогично в ServerHello поле key-share служит для передачи серверной части параметров Диффи-Хеллмана. Соответственно раннее используемые ClientKeyExchange и ServerKeyExchange заменены на расширение key_share в ServerHello и ClientHello.

Поле signature_algorithms содержит шифронаборы, которые поддерживает клиент, сервер из них выбирает какой-либо и использует его для подписи всех полученных и отправленных сообщений на данный момент, помещая полученное значение в CertificateVerify (содержит выбранный алгоритм подписи и собственно саму подпись). Signature_algorithms использовалось и в TLS1.2, но обозначало алгоритм вычисления цифровой подписи, которая ставилась сервером на своей части данных обмена общим ключом, т. е. в ServerKeyExchange. PSKKeyExchangeModes и PreSharedKey, предназначены для создания сессий, использующих ранее согласованные между узлами ключи. Определяют, соответственно, поддерживаемые режимы ("прямой" режим, или режим с дополнительным использованием протокола ДН) и идентификатор ключа. Очевидно, что секретный ключ не может передаваться в открытом виде, поэтому расширение PreSharedKey содержит только данные, позволяющие другой стороне найти нужный ключ у себя. Также PreSharedKey определяет срок действия ключа и ряд других параметров (криптосистему и т. д.).

Early_data в TLS1.3 позволяет клиенту и серверу возобновить соединение очень быстро и работать в так называемом режиме 0-RTT, с помощью которого клиент может сразу начать отправлять полезную нагрузку, например HTTP-запрос GET. О работе в данном режиме сервер узнает по наличию расширений early_data и PreSharedKey. Второе необходимо, чтобы сервер смог выбрать ранее согласованный сторонами секрет и сгенерировать

ключ, который нужен для расшифровки поступивших в зашифрованной TLS-записи (т. е. в Application Data) данных клиента.

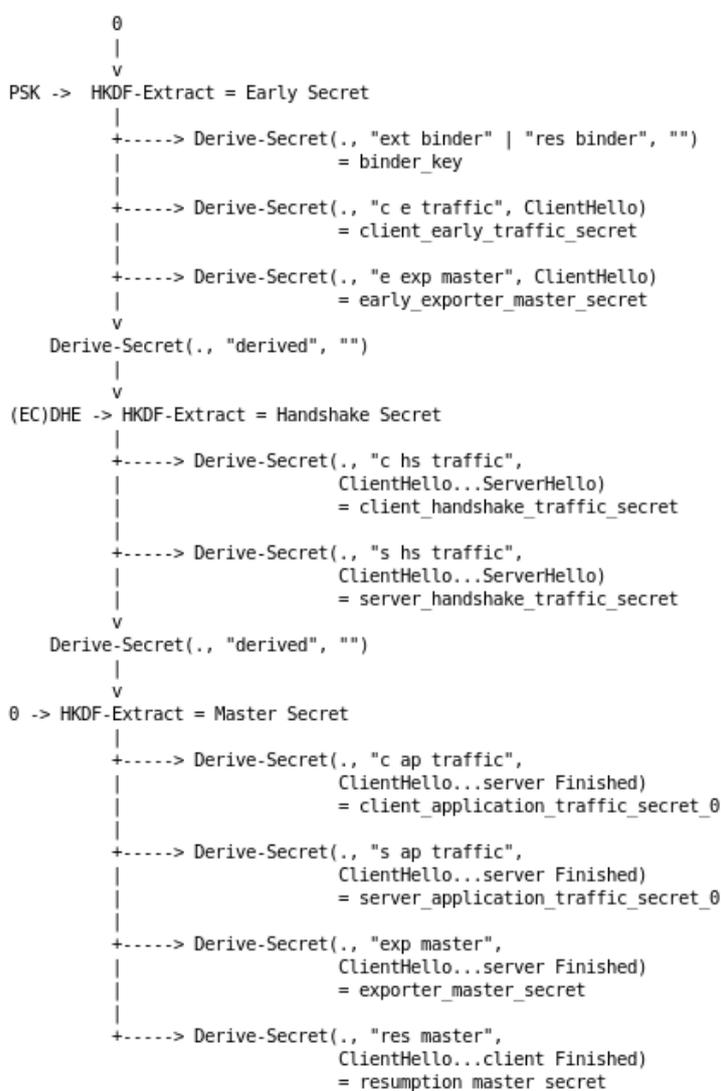
Расширение cookie, которое сервер может передать в HelloRetryRequest, позволяет ему удостовериться в том, что клиент действительно намерен установить соединение. Достигается это требованием в ответе ClientHello повторить содержимое поля cookie. Это сделано для защиты от флуда, так как установление TLS соединения требует немалого количества вычислительных ресурсов. Аналогичный механизм имеется в протоколе TCP, когда данные о соединении не сохраняются до тех пор, пока клиент не повторит значение cookie в ответном syn (защита от syn флуда). В целом, для защиты от подобных атак и должны срабатывать механизмы, находящиеся ниже уровня работы TLS протокола, например уровень протокола TCP, поэтому cookie в TLS это скорее избыточная защита, так как если происходят проблемы здесь, то ошибки были допущены на более низких уровнях. Первая фаза завершается передачей ServerHello, которым заканчивается часть открытых сообщений handshake. Вторая фаза включает серверные сообщения EncryptedExtensions и, при необходимости аутентификации клиента, CertificateRequest. Новое сообщение EncryptedExtensions дает возможность расширениям, которые раньше отправлялись в открытом виде в ServerHello, теперь передаваться в защищенной TLS-записи.

Третья фаза представляет собой группы сообщений обоих узлов Certificate (теперь серверный сертификат передается в защищенной TLS-записи и третья сторона не сможет узнать ни его содержимое, ни факт его передачи), CertificateVerify и Finished, с помощью которых происходит аутентификация сервера и клиента. Как и в предыдущих версиях TLS, аутентификация может быть полностью исключена (анонимный режим), однако типичный сценарий использования подразумевает аутентификацию, по крайней мере, сервера клиентом. При этом, в TLS 1.3 сервер может перейти к отправке данных полезной нагрузки непосредственно после серверного

Finished — это возможно, так как к этому моменту сервером уже получен первый набор симметричных ключей.

1.4.1 Управление сеансовыми ключами в TLS 1.3

Основой общего секрета является значение, полученное в рамках обмена ДН, также (в дополнение к ДН) могут применяться секретные значения, которые узлы согласовали заранее (PSK схемы). Поколения ключей соответствуют этапу соединения, при этом они вычисляются на основе соответствующего поколения секрета. На каждом шаге генерируется новый секрет с добавлением дополнительной информации из сообщений протокола (рис. 5). В предыдущих версиях ничего похожего не было.



PSK (предварительно согласованный общий ключ либо извне, либо полученный из значения `resumption_master_secret` предыдущего соединения). (EC)DHE общий секрет (значение, рассчитанное для конечных групп полей алгоритма Диффи-Хеллмана). Функция `Derive-Secret` обеспечивает трансформацию секретов при переходе между этапами.

Рис. 5. Схема преобразования ключей в TLS1.3. RFC 8446 [5]

На рис. 5 этапы преобразований идут сверху вниз, начиная от "пустого" значения, обозначенного 0 (в протоколе это строка нулевых байтов заданной длины). Центральная вертикальная линия соответствует движению ключевых данных, где к ним последовательно применяется функция генерации ключей (HKDF). Слева - изображены источники входных данных с ключевой информацией. Справа - промежуточные результаты (общие секреты), сообщения Handshake, используемые в качестве дополнительной подмешиваемой информации, и симметричные "этапные" секреты, которые получают на каждом шаге. На рис. 5 отражён алгоритм генерации симметричных секретов, которые еще не являются сеансовыми ключами. Ключи, а также соответствующие векторы инициализации, рассчитываются на основе секретов в результате применения функции из семейства HKDF, а именно — HKDF-Expand-Label(). Клиент и сервер могут периодически заменять ключи защиты трафика приложений, выполняя ещё одну итерацию преобразования.

Данные необходимые для генерации ключей и векторов инициализации: общий секрет; целевое значение (key или iv); длина генерируемого ключа.

```
[sender]_write_key = HKDF-Expand-Label(Secret, "key", "", key_length) [5]
```

```
[sender]_write_iv = HKDF-Expand-Label(Secret, "iv", "", iv_length) [5]
```

1.4.2 Сокращённый вариант установления соединения в TLS 1.3. Схема 0-RTT

Одно из самых больших преимуществ TLS 1.3 по сравнению с более ранними версиями заключается в том, что для установки соединения требуется 1 RTT, как для полного варианта, так для сокращенного. Это обеспечивает значительный прирост скорости при установлении новых соединений, но не для их возобновления. Очень большое количество HTTPS-соединений происходят с помощью механизма возобновления сеанса (либо с помощью SessionID, либо с TLS Session Ticket), а из-за того, что в данном случае версии TLS до 1.3 имеют 1 RTT, то современный протокол не показывает выигрыша в скорости.

Браузеру, загружающему веб-страницу по протоколу HTTPS, необходимо выполнить четыре этапа при первой попытке доступа к сайту: 1) DNS запрос; 2) TCP Handshake (1 RTT); 3) TLS Handshake (RTT зависит от версии); 4) HTTP (1 RTT). Теперь можно сравнить версии TLS по значению RTT для работы по HTTPS:

TLS1.2 (и более ранние версии).

- Новое соединение: 4 RTT + DNS;
- Возобновление сеанса: 3 RTT + DNS.

TLS1.3

- Новое соединение: 3 RTT + DNS
- Возобновление сеанса: 3 RTT + DNS

TLS1.3 + 0-RTT

- Новое соединение: 3 RTT + DNS
- Возобновление сеанса: 2 RTT + DNS

При использовании 0-RTT имеем большой прирост скорости.

Для использования схемы 0-RTT необходимо, чтобы стороны заранее согласовали общий секрет. Если данный секрет известен, то клиент может начать соединение отправкой ClientHello, с указанием общего секрета, за которым сразу же следуют данные полезной нагрузки. В случае с HTTPS, в виде данных может выступать, например, запрос HTTP GET. Клиент сразу же, не дожидаясь ответа сервера, приступает к отправке запросов уровня приложения. Что позволяет зашифрованные HTTPS-запросы сделать такими же быстрыми, как незашифрованные HTTP-запросы. Если сервер принял соединение, он отвечает сообщением ServerHello, а также Certificate и т. д., заканчивает же фазу открытия сессии сообщением Finished, и сразу после этого отправляет данные полезной нагрузки, которые являются ответом на запрос приложения клиента. В случае HTTPS, это будет HTTP-ответ на клиентский GET-запрос. Клиентский запрос в схеме 0-RTT не обладает прогрессивной секретностью: он шифруется не выделенным сеансовым

ключом, а общим секретным ключом, полученным в другом сеансе. Следовательно, если ключ будет скомпрометирован, получившая его сторона сможет расшифровать клиентскую часть Handshake ранее записанного трафика. Но это относится только к клиентскому запросу, так как далее трафик приложения шифруется с использованием сессионных ключей, которые уже могут быть сгенерированы с обеспечением прогрессивной секретности. В отличие от любых других запросов, отправляемых по TLS, запросы, посланные клиентом как часть схемы 0-RTT, уязвимы к так называемой атаке воспроизведения. Если злоумышленник имеет доступ к вашему зашифрованному соединению, он может взять копию данных 0-RTT, которая содержит первый запрос, и снова отправить ее на сервер, притворившись клиентом. Это может привести к тому, что сервер обработает повторные запросы, при том, что клиент отправил только один. Собственно, из-за данных причин, схема 0-RTT в целом считается менее безопасной (это отмечено в спецификации). TLS1.3 — является большим шагом вперед с точки зрения повышения производительности и безопасности. Комбинируя TLS1.3 с 0-RTT, выигрыш в производительности становится еще более впечатляющим. А если объединить это с HTTP/2, то зашифрованная передача данных становится как никогда быстрой.

2 УЯЗВИМОСТИ TLS/SSL

2.1 Модель угроз TLS

Приведем модель угроз SSL, разработанную Иваном Ристичем (рис. 6). Несмотря на то, что она была опубликована в 2009 году, актуальность ее еще сохраняется.

Итак, перечислим основные угрозы безопасности TLS:

- Уязвимости конечных узлов (клиента и/или сервера). Например, неправильная настройка сервера (варианты конфигурации, приводящие к уязвимостям), использование устаревшего, а следовательно, возможно уязвимого ПО (как со стороны клиента, так и со стороны сервера).
- Необходимость доверия третьей стороне для проверки (основана на инфраструктуре открытых ключей: PKI) сертификатов. Так как никто не мешает удостоверяющему центру сертификации выпустить абсолютно валидный сертификат, который может быть потом использован в MITM атаке (здесь нужно оговориться, что сейчас проводятся попытки создания и внедрения механизмов, которые должны это изменить). Кроме этого, тоже самое может проделать злоумышленник, получив доступ к закрытому ключу УЦ. Нельзя забывать и том, что возможны ошибки программного обеспечения при проверке цепочки сертификатов, так браузер Microsoft IE длительное время, до 2003 года, производил некорректную проверку статуса сертификата, которая определяла его возможность подписывания других сертификатов, это позволяло выпускать валидные, с точки зрения IE, сертификаты для любого имени, просто получив обычный серверный сертификат у УЦ.
- Уязвимости на уровне протокола: ошибки, неточности в спецификации. Как вариант могут приводить к возможности понижения версии протокола третьей стороной до уязвимой, так называемая Downgrade attack.

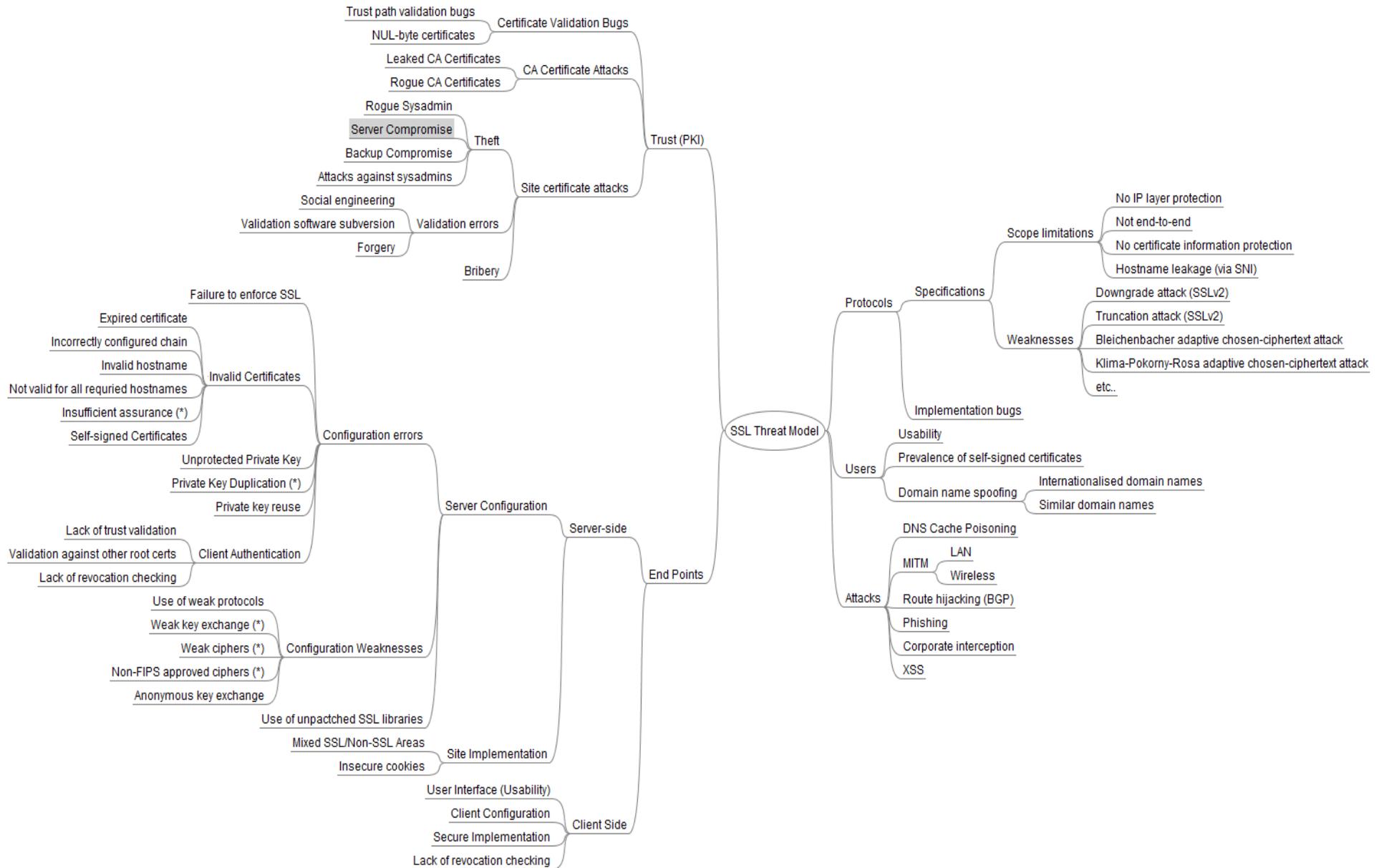


Рис. 6. Модель угроз SSL [6]

- Баги реализации. Собственно, на них основано очень большое количество атак, например использующих утечку информации о состоянии криптосистемы (по различным ответам Alert о уровне и состоянии ошибки, по времени обработки сообщений).
- Использование невнимательности пользователя: похожие доменные имена, мошеннические сайты идентичные официальным, но работающие по протоколу HTTP и т. д.
- Различные виды атак, как пассивных (возможно просто запись всего трафика для дальнейшего анализа), так и активных. Сюда можно отнести MITM атаки, фишинг, XSS уязвимости и т. п.

2.2 Атаки на TLS

Большой проблемой является то, что до сих пор можно нередко встретить поддержку заведомо нестойких, устаревших шифронаборов, которые используют ключи длиной менее 128 бит. Иногда встречается даже SSLv2, который давно не является хоть сколь-нибудь защищённым на практике. Поддержка старых версий протокола, а также слабых параметров криптосистем позволяет атакующему произвести понижение уровня защиты до такого шифра, который он сможет без труда взломать на лету и благодаря этому подделать сообщения Finished, став посредником в сессии. Однако современные версии браузеров вводят ограничения и не позволяют использовать заведомо нестойкие сочетания криптосистем, а также не поддерживают устаревшие версии TLS/SSL. При использовании алгоритмов блочного шифрования необходимо выполнять дополнение данных, что приводит к возникновению криптографических оракулов, которые помогают злоумышленнику раскрывать зашифрованные сообщения. Есть целый класс таких оракулов, которые в англоязычной литературе называются Padding oracle, породивший несколько нашумевших атак. Одним из первых использований оракула, связанного с дополнением данных, относится к 2003 году. Атака, сконструированная Сержем Воденэ (Serge Vaudenay), основывалась на том факте, что реализация протокола SSL возвращала разные

сообщения об ошибках в зависимости от того, удалось ли после расшифровки записи обнаружить корректное дополнение, но не совпал код аутентификации (MAC), или корректного дополнения данных обнаружить не удалось. Причину, делающую данную атаку возможной, устранили только спустя семь лет после ее теоретического описания в веб-сервере IIS.

2.2.1 BEAST

Browser Exploit Against TLS/SSL (BEAST) была обнаружена в сентябре 2011 года. Применима атака к SSL 3.0 и TLS 1.0, поэтому ее целью являются браузеры, поддерживающие протокол TLS1.0 или более ранние версии. Злоумышленник может расшифровать данные, которыми обмениваются две стороны, воспользовавшись уязвимостью в реализации режима шифрования Cipher Block Chaining (CBC) в TLS 1.0, которая заключается в использовании в качестве вектора инициализации очередного сообщения последнего блока шифротекста предыдущего сообщения. Уязвимость BEAST зарегистрирована в базе данных NIST NVD (национальная база данных уязвимостей США) как CVE-2011-3389 [8]. Выполняется данная атака на стороне клиента с использованием MITM. Злоумышленник использует MITM для вмешательства в поток TLS. Это позволяет ему угадать вектор инициализации (IV), используемый с введенным сообщением, а затем просто сравнить результаты с результатами блока, который он хочет расшифровать. Для успешности проведения BEAST атаки, злоумышленник должен иметь некоторый контроль над браузером жертвы. Рассмотрим атаку более подробно. Как уже говорилось для успешного ее проведения атакующему необходимо иметь возможность перехватывать трафик, передаваемый браузером жертвы, а также ему как-то нужно заставить передавать специальным образом сформированные данные (например заманив жертву на свой сайт с JS кодом) по уже установленному защищенному каналу между клиентом и сервером (это является довольно сложной задачей и напомним, что передавать сообщения по установленному соединению необходимо, так как в процессе атаки нужно влиять на открытый текст, а также с другого домена нет доступа к ресурсам (кукам) атакуемого

сайта, которые нужно раскрыть, мешает SOP (Same Origin Policy), данную проблему авторам уязвимости удалось решить, найдя в виртуальной машине Java 0day-уязвимость и написав для нее работающий спloit).

Предположим, что между компьютерами Алисы и Боба установлено безопасное соединение, злоумышленник (Ева) перехватывает сообщение и вычленяет оттуда i -блок (является зашифрованным, обозначим его C_i) который по ее предположению содержит секретную информацию (например пароль или данные куки). C_i -му соответствует открытый текст P_i , содержащий секрет. Согласно режиму CBC: $C_i = E(\text{Key}, P_i \text{ xor } C_{i-1})$, где E — блочный шифр, Key — ключ, C_{i-1} — предыдущий блок шифротекста. Ева может предположить, что секрет — это S и с помощью атаки BEAST проверить свою гипотезу. Так как весь трафик перехватывается, то Ева знает, какой блок шифротекста будет использоваться, как инициализирующий вектор при шифровании первого блока следующего сообщения (последний зашифрованный блок предыдущего) — обозначим его IV .

Далее формируется сообщение таким образом, чтобы первый блок был равен:

$$P_1 = C_{i-1} \text{ xor } IV \text{ xor } S$$

Если сообщение получилось передать по защищенному каналу между Алисой и Бобом, то его первый блок будет таким:

$$C_1 = E(\text{Key}, P_1 \text{ xor } IV) = E(\text{Key}, (C_{i-1} \text{ xor } IV \text{ xor } S) \text{ xor } IV) = E(\text{Key}, C_{i-1} \text{ xor } S) = C_i$$

Это равенство получается благодаря свойствам операции XOR: $IV \text{ xor } IV = 0$, а $U \text{ xor } 0 = U$, для любого U .

Следовательно равенство зашифрованного блока C_1 и ранее перехваченного C_i , означает истинность гипотезы о значении секрета. А если C_1 не равен C_i , то предположение неверное. На практике угадывали не весь секрет целиком, а значение каждого символа отдельно, это достигалось благодаря смещению секрета так, чтобы сначала только первый символ попадал в C_i (тогда в C_i он находится в самом конце). При этом считается, что остальные значения байтов блока известны, это могут быть стандартные поля.

Отсюда получается, что для гарантированного подбора первого символа необходимо перебрать 256 вариантов, что не так уж и много. После этого секрет сдвигается и теперь два символа его находятся в C_i , а первый уже известен, следовательно опять перебираем 256 вариантов и так до тех пор, пока не вычислим весь секрет. Авторам уязвимости удалось таким образом подобрать куки PayPal за 103 секунды.

Так как BEAST является чисто клиентской уязвимостью, то для ее экстренного предотвращения были устранены неисправности, позволяющие отправлять данные по установленному защищенному каналу, а также введены некоторые браузерные механизмы, заключающиеся в отправке фиктивных сообщений перед всеми настоящими. В TLS начиная с версии 1.1 введены изменения значений, используемых векторов инициализации (последние блоки шифротекста в качестве IV более не применяются).

Профилактические меры: обновить ПО; использовать TLS1.1 либо более новые версии.

2.2.2 CRIME

Compression Ratio Info-leak Made Easy (CRIME) использует уязвимость сжатия TLS. Методы сжатия, поддерживаемые клиентом, включены в сообщение приветствия (хотя на сегодняшний момент в качестве значения данного поля обычно передается null, означающее отсутствие сжатия), они были введены в TLS/SSL для уменьшения объема передаваемых данных. Соединение может быть установлено без сжатия. DEFLATE - наиболее распространенный алгоритм сжатия. Уязвимость CRIME зарегистрирована в базе данных NIST NVD как CVE-2012-4929 [9]. Одним из основных методов, используемых алгоритмами сжатия, является замена повторяющихся байтовых последовательностей указателем на первый экземпляр этой последовательности. Чем больше повторяющихся элементов, тем выше степень сжатия.

Требованиями к реализации CRIME выступают MITM и возможность отправлять запросы на сервер от имени клиента. Предположим, что

злоумышленник хочет получить cookie. Ему известно, что целевой веб-сайт (example.com) создает файл cookie для сеанса с именем key. Злоумышленник также знает, что применяется метод сжатия DEFLATE. Для проведения атаки он добавляет Cookie:key=0 в запрос жертвы. Потом отправляет его на сервер, после чего измеряет длину зашифрованного сообщения, далее злоумышленник заменит 0 на другой символ и снова сделает запрос с проверкой длины ответа. Его целью выступает поиск такого символа, при добавлении которого длина сообщения становится меньше (так как это означает совпадение символа с символом в реальном куке). Таким образом подобрав первый символ, он переходит к следующему и так до тех пор, пока не будет раскрыт весь cookie. В марте 2013 года в Black Hat (EU) было представлено расширение CRIME под названием TIME, которое не требовало MITM, а использовало размеры окон TCP.

Профилактические меры: обновление браузера до последней версии; не использовать сжатие TLS; отключение в браузере сторонних куки (third cookies).

2.2.3 BREACH

Уязвимость Browser Reconnaissance and Exfiltration via Adaptive Compression of Hypertext (BREACH) очень похожа на CRIME, но BREACH нацелена на сжатие HTTP, а не на сжатие TLS. Эта атака возможна, даже если сжатие TLS отключено. Злоумышленник заставляет жертву подключиться к веб-сайту (при помощи своего ресурса с вредоносным JS кодом), подверженному уязвимости, и отслеживает трафик между клиентом и сервером с помощью MITM. Уязвимость BREACH зарегистрирована в базе данных NIST NVD как CVE-2013-3587. Уязвимое веб-приложение должно удовлетворять следующим условиям: взаимодействовать с сервером, использующим сжатие на уровне HTTP; отражать пользовательский ввод в теле HTTP ответа (так как сжатие HTTP не затрагивает заголовки); передавать секреты также в теле ответа HTTP (поэтому значения в заголовках HTTP защищены от этой атаки). Т. е. атакующий специальным образом, формируя

запросы, следит за изменением длины ответа и на основе этого может постепенно раскрыть секрет (который для реализации атаки должен передаваться в теле ответа вместе с пользовательским вводом), например CSRF токен.

Профилактические меры: отключение сжатия HTTP; отделение секрета от пользовательского ввода; рандомизирование секрета по запросу; маскировка секрета; сокрытие длины добавлением случайных чисел к ответам; ограничение частоты запросов.

2.2.4 Lucky13

Зарегистрирована в базе данных NIST NVD как CVE-2013-0169. Использовала уязвимость реализации TLS1.1 и 1.2 (и к более ранним), а также DTLS1.0 и 1.2. В атаках, которые основаны на подмене дополнения, отмечено, что сигналом о верной интерпретации байтов дополнения могут служить как разные коды ошибок, возвращаемые сервером в alert сообщениях, так и сам факт успешной обработки запроса протоколом, находящимся на уровень выше TLS (HTTPS в случае с POODLE). Однако злоумышленник может извлекать полезную информацию путём анализа времени обработки данных на стороне сервера. На использовании данного информационного канала работает атака Lucky13. Основная идея Lucky13 состоит в отправке на сервер специальным образом подготовленных TLS-записей, содержащих тщательно сформированные дефекты. Обработка этих записей на сервере и, в частности, вычисление кода аутентификации, приводит к тому, что сообщение об ошибке поступает с различной задержкой по времени. То есть, уязвимые реализации используют один универсальный код ошибки, но время генерации этого кода коррелирует с данными открытого текста, расположенного внутри TLS-записей. Получив статистику времени обработки запросов, злоумышленник сможет вычислить открытый текст. В данном случае в качестве криптографического оракула выступает время выполнения операций на сервере. В своей простейшей форме данная атака может восстановить полный блок открытого текста с шифрованием TLS, используя около 2^{23} сеансов TLS,

при условии, что злоумышленник находится в той же локальной сети, что и атакуемая машина, а также используется в качестве алгоритма вычисления MAC HMAC-SHA1. Количество необходимых сессий может быть уменьшено до 2^{19} , если известно, что открытый текст кодируется в base64. До 2^{13} сеансов TLS на байт можно достичь, если байт открытого текста в одной из двух последних позиций в блоке уже известен. Сложность атаки различна для разных алгоритмов MAC.

Профилактические меры: обновить ПО; использовать AEAD шифрование.

2.2.5 Heartbleed

Heartbleed являлась критической уязвимостью, обнаруженной в расширении heartbeat популярной библиотеки OpenSSL. Это расширение используется для поддержания связи между клиентом и сервером. Уязвимость Heartbleed зарегистрирована в базе данных NIST NVD как CVE-2014-0160 [11]. Клиент отправляет сообщение запрос с полезной нагрузкой, которая включает данные и их размер (плюс заполнение) на сервер. Сервер должен ответить тем же запросом, содержащим данные и размер данных, отправленных клиентом. Основой уязвимости Heartbleed был тот факт, что, если клиент отправлял неверную длину данных, сервер отвечал данными, полученными клиентом, и случайными данными из его памяти, чтобы сообщение соответствовало требованиям длины, указанным отправителем. Утечка незашифрованной информации из памяти сервера может иметь катастрофические последствия. Были проведены пробные варианты эксплуатации этой уязвимости, в которых злоумышленник смог получить закрытый ключ сервера. Память сервера может содержать учетные данные пользователей, конфиденциальные документы, номера кредитных карт и т. д.

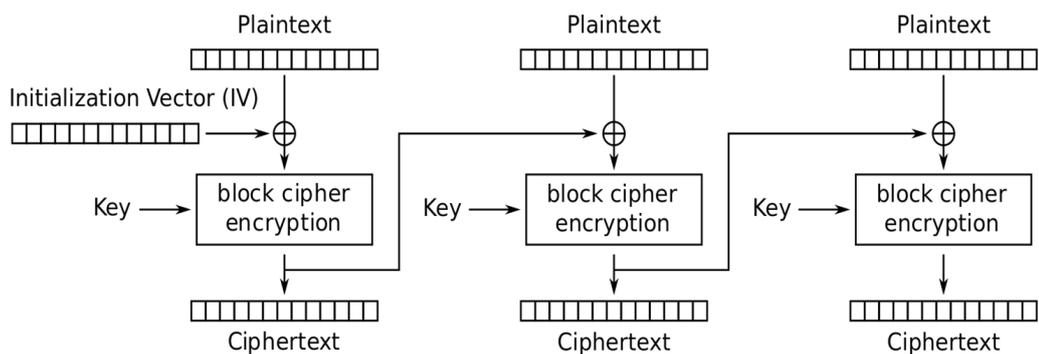
Профилактические меры: обновление до последней версии OpenSSL, если это невозможно, то необходимо перекомпилировать уже установленную версию с параметром `-DOPENSSL_NO_HEARTBEATS`.

2.2.6 POODLE

Зарегистрирована в базе данных NIST NVD как CVE-2014-3566 [14]. POODLE использует уязвимость реализации. Для проведения данной атаки предполагается, что атакующий способен переставлять блоки шифротекста внутри защищённых сообщений, а также может изменять часть открытого текста. Для понимания данной атаки необходимо более подробно рассмотреть режим шифрования CBC (рис. 7). Основные этапы шифрования в данном режиме:

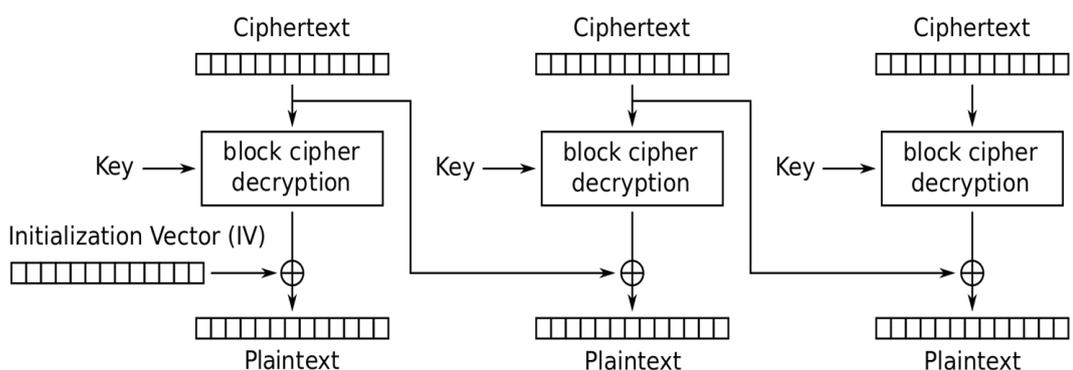
1. Текстовое сообщение разбивается на дискретные 16-байтные блоки данных.
2. Случайным образом генерируется вектор инициализации (IV).
3. Первый блок шифруется с использованием выбранного блочного шифра и ключа.
4. Над результатом работы блочного шифра и вектором инициализации выполняется операция XOR.
5. Результат шага 4 сохраняется как первый блок зашифрованного текста.
6. Следующий блок открытого текста передается на вход шифра.
7. Над результатом работы блочного шифра и предыдущим блоком зашифрованного текста выполняется операция XOR.
8. Результат шага 7 сохраняется как следующий блок зашифрованного текста.
9. Шаги 6-8 повторяются, пока не закончится открытый текст.

Дешифрование использует обратный алгоритм шифра и обратную схему следования блоков (рис. 8). При этом, если для шифрования следующего блока нужно было знать результат шифрования предыдущего, то для расшифрования произвольного блока достаточно знать значение шифротекста предыдущего: то есть, расшифрование может работать параллельно.



Cipher Block Chaining (CBC) mode encryption

Рис. 7. Шифрование в режиме CBC [12]



Cipher Block Chaining (CBC) mode decryption

Рис. 8. Дешифрование в режиме CBC [13]

Поскольку AES в режиме CBC работает только с 128 битными блоками, то может случиться так, что длина входных данных не будет кратна 128. В таком случае потребуются дополнительные биты, которые называются заполнением, а TLS предписывает специальный формат для них. Если сервер каким-то образом выдает информацию о соответствии заполнения определенному формату, то он передает злоумышленнику необходимую информацию для расшифровки сообщений. Атака POODLE использует тот факт, что SSLv3 не предписывает детерминированный формат заполнения и, следовательно, невозможно полностью проверить все эти байты. Это означает, что злоумышленник, имеющий возможность перемещать интересующий его блок зашифрованного текста, сможет заменить им блок заполнения, в следствии чего он будет способен расшифровать последний байт целевого блока, в случае подтверждения сервером правильности заполнения. POODLE

затрагивает как системы, использующие SSLv3, так и те, которые неправильно использовали функцию заполнения SSLv3 в своих реализациях TLS.

В процессе атаки злоумышленник должен создавать HTTPS-запросы с браузера жертвы на сайт, в котором была пройдена аутентификация, причем так, чтобы cookie сеанса были выровнены по концу блока зашифрованного текста, и чтобы общая длина открытого текста являлась кратной размеру блока (эта длина сообщения гарантирует, что последний блок зашифрованного текста содержит только байты заполнения). Добиться этого можно, например при помощи манипуляции с адресом документа (path), а также добавлением дополнительных полей заголовка. При шифровании запросов, клиент должен рассчитать код аутентификации сообщения (MAC), присоединить это значение к сообщению, а затем добавить заполнение, которое будет являться последним блоком, так как для этого URL и заголовки были специально подобраны. Стоит отметить, что байты заполнения не учитываются при вычислении кода аутентификации сообщения. После отправки зашифрованного запроса клиентом, злоумышленник заменяет последний блок шифротекста на блок, который, как ожидается, будет содержать секретные сеансовые cookie. После получения сообщения уязвимый сервер сначала расшифрует данные в соответствии с CBC режимом. При этом получившийся открытый текст из последнего блока будет результатом побитового XOR зашифрованного текста предпоследнего блока с расшифрованным блоком, содержащим секретные куки. Это означает, что последний блок открытого текста фактически будет случайным, так как из свойств CBC шифрования TLS шифротекст предпоследнего блока будет изменяться, что и обеспечивает возможность перебора. Последний байт, получившегося открытого текста будет интерпретироваться как значение длины заполнения, обозначим его n . При правильной реализации TLS сервер проверит, что все последние $n+1$ байтов имеют значение n . Если это не так, то он, как ожидается, отклонит сообщение с отправкой общего предупреждения (alert) TLS «Bad Record MAC». Уязвимые же реализации TLS не выполняют эту проверку и вместо этого просто удаляют

последние $n+1$ байт. Далее проверяется MAC и единственным вариантом, при котором он будет валидным является значение последнего байта после расшифровки и операции XOR — $0x0F$ (т. е. 15). Злоумышленнику необходимо наблюдать и на основании того, отклоняет сервер или принимает измененную запись, делать выводы. Если запись принята, то атакующий теперь знает, что после расшифровки последнего блока и применении к нему операции XOR с предпоследним, получилось равенство последнего байта $0x0F$:

$E(\text{Key}, \text{Clast})[15] \text{ xor } \text{Clast-1}[15] = 0x0F = 15$, где E — блочный шифр; Key — ключ шифрования; Clast — последний блок шифротекста; $E(\text{Key}, \text{Clast})[15]$ — последний байт последнего блока, получившийся после дешифрования; $\text{Clast-1}[15]$ — последний байт предпоследнего блока шифротекста.

Откуда злоумышленник сможет вычислить один байт секретного cookie:

$P[15] = 15 \text{ XOR } \text{Clast-1}[15] \text{ XOR } \text{Cdc}[15]$, где $P[15]$ — значение последнего байта сеансового cookie; $\text{Cdc}[15]$ — последний байт блока шифротекста, находящегося перед блоком с зашифрованными куки (копию которых злоумышленник переместил на место заполнения).

Затем происходит корректировка URL-адреса и заголовков в контролируемом HTTPS-запросе так, чтобы другой байт открытого текста (с куки) совпадал с концом блока. Например, `path` может быть обрезан на один байт, чтобы сместить позицию значения cookie при одновременном добавлении 15-байтового заголовка, для сохранения длины заполнения в 16 байт (15-байтовый заголовок добавит 17 байт из-за двух байт окончания строки CRLF). Этот процесс повторяется до тех пор, пока все байты cookie не будут расшифрованы. В среднем, каждый байт требует 256 запросов.

Профилактические меры: не использовать SSLv3; отказаться от шифронаборов, содержащих режим CBC.

2.2.7 Logjam

Зарегистрирована в базе данных NIST NVD как CVE-2015-4000 [16]. Данной атаке подвержен протокол TLS 1.2 и более ранние версии, при условии поддержки шифронабора DHE_EXPORT на сервере. Так как ранее браузеры

принимали любые параметры Диффи-Хеллмана, то этот факт, позволял злоумышленнику, находящемуся в середине, произвести атаку с понижением шифра путем переписывания поддерживаемых шифронаборов клиентом (Cipher Suites) в ClientHello на шифронаборы, содержащие DHE_EXPORT, а затем в ServerHello заменял DHE_EXPORT на DHE. Это позволяло заставить использовать группы DH малой разрядности (512 бит), что в свою очередь могло привести к раскрытию общего секрета, а следовательно, и к раскрытию сеансовых ключей. В популярных браузерах проблема была исправлена летом 2015 года. Logjam напоминает атаку FREAK (CVE-2015-2319), которая использовала понижение безопасности соединения до использования EXPORT_RSA (разрядность модуля 512 бит), т. е. в качестве сеансового шифра использовался RSA малой разрядности.

Профилактические меры: обновить используемые браузеры; на стороне сервера исключить шифронаборы, содержащие DHE_EXPORT (а для противодействия FREAK EXPORT_RSA).

2.2.8 DROWN

Decrypting RSA with Obsolete and Weakened eNcryption (DROWN) зарегистрирована в базе данных NIST NVD как CVE-2016-0800 [17]. Атака основана на использовании между узлами алгоритма RSA для передачи Premaster Secret от клиента серверу (клиент шифрует секрет с помощью открытого ключа сервера и отправляет), а также на поддержке сервером SSLv2. DROWN демонстрирует, что даже простая поддержка SSLv2 представляет угрозу для серверов и клиентов, не смотря на установление ими соединения по более современному протоколу. Атака позволяет злоумышленнику расшифровать данные, передающиеся по TLS соединению между клиентами и серверами, отправляя сообщения-зонды на сервер, который поддерживает SSLv2 и использует тот же закрытый ключ.

С технической точки зрения DROWN - это новая форма атаки оракула Блейхенбахера, описанной в 1998 году, которая заключалась в том, что атакующий может просто направлять TLS-серверу случайные ключи,

спрашивая, верны ли они (при условии использования RSA, как описано выше). Тем самым происходит утечка информации и в конечном итоге раскрытие Premaster Secret. После обнаружения в 1998 году данной уязвимости отказ от использования RSA не произошел, а лишь были введены меры усложняющие брутфорс. Примером того, что этого оказалось недостаточно служит атака DROWN и ряд других, которые используют те же самые принципы, что и атака Блейхенбахера.

Профилактические меры: отключить поддержку SSLv2 на сервере; обновить OpenSSL; отказаться от использования шифронаборов, в которых применяется передача общего секрета с помощью RSA.

2.2.9 Sweet32

Зарегистрирована как CVE-2016-2183 [17]. Данная атака использует фундаментальные особенности протокола и поэтому не является уязвимостью какой-то конкретной реализации. Для реализации Sweet32 требует использование блочного шифра в режиме CBC с размером блока в 64 бита. Атака основана на неизбежном возникновении коллизии в результате работы шифра, что означает появления двух одинаковых зашифрованных блоков. Так, для 64-битного блока количество данных, которые должны пройти через шифр будет около 32 гигабайт.

Следствие коллизии: $C_i == C_k$ (C_i и C_k , совпавшие блоки) $\rightarrow M_i == M_k$ (M_i и M_k блоки, поступающие на вход шифру после операции XOR с C_{i-1} и C_{k-1} соответственно) $\rightarrow C_{i-1} \text{ XOR } T_i == C_{k-1} \text{ XOR } T_k$

(T_i и T_k блоки открытого текста) $\rightarrow T_i \text{ XOR } T_k == C_{i-1} \text{ XOR } C_{k-1}$, а так как C_{i-1} и C_{k-1} известны, то нам теперь известно значение операции XOR между T_i и T_k , зная один из них или его часть, можно раскрыть другой либо целиком, либо частично, что и является утечкой конфиденциальных данных.

С вероятностью около $1/2$ коллизия блоков произойдет на выборке в $2^{n/2} = 2^{32}$ блоков (n - разрядность блока, 64 бита).

Профилактические меры: отказаться от использования шифронаборов, которые включают блочные шифры в режиме CBC (отличным вариантом будет применение аутентифицированного шифрования).

2.2.10 HEIST

HTTP Encrypted Information can be Stolen through TCP-windows (HEIST) зарегистрирована как CVE-2016-7153 [18].

Атака, названная TIME, впервые была представлена на Black Hat EU в 2013 году, но она не привлекла к себе большого внимания. Вскоре на конференции Black Hat USA в августе 2016 года она опять появилась с некоторыми дополнениями и с новым названием. HEIST позволяет CRIME и BREACH обходиться без MITM, а ограничиваться только манипуляциями в браузере жертвы. В ее основе лежит атака на длину сообщения с помощью размера tcp окна. А именно найдя возможность делать запрос на сервер с параметром (происходить это может, например с помощью JavaScript кода, встроенного в страницу взломщика и на которую была заманена жертва), который будет включен и в ответ. Злоумышленник сможет отслеживать поместился ли ответ в одно tcp окно или же было подтверждение и оставшиеся данные были переданы уже в другом (это происходит с помощью отслеживания времени получения первого и последнего байта ответа, благодаря API браузера (просто так к ответу нельзя получить доступ)). Меняя длину сообщения, необходимо будет найти граничное значение размера параметра, при котором будет достаточно одного окна. После чего не составит проблем вычислить длину ответного сообщения. Т. е. HEIST не дает новый инструмент для взлома TLS, а лишь расширяет возможности других атак, в основе которых лежит отслеживание длины после сжатия. Поэтому для продолжения атаки можно воспользоваться, например BREACH. Но при этом HEIST предоставляет возможность с помощью инфицированного сайта получить большое количество конфиденциальной информации основываясь только на длине ответов к различным сервисам, не применяя MITM, на которых клиент аутентифицирован. Атака пригодна как для HTTP/1.1, так и

для HTTP/2 даже не смотря на применение в данном протоколе HPACK, который противодействует атакам, использующим сжатие HTTP (примером может выступать BREACH, рассчитанный на применение DEFLATE).

Профилактические меры: обновление браузера до последней версии; отключение в браузере сторонних куки (third cookies); отключить поддержку сжатия на сервере.

2.2.11 ROBOT

Return Of Bleichenbacher's Oracle Threat (ROBOT) зарегистрирована в базе данных NIST NVD как CVE-2017-13099 [19]. Данная атака является возвращением атаки, описанной в 1998 году Даниэлем Блейхенбахером, собственно, она еще раз демонстрирует, что те меры, которые были применены для исправления уязвимости (а именно усложнение брутфорса) не были достаточными. ROBOT также использует уязвимость дополнения RSA, но в отличии от оригинальной работы, где в качестве оракулов использовались alert сообщения, здесь применяют различные виды сигналов, такие как тайм-ауты, сброс подключения, дубликаты сообщений TLS alert. Т. е. были найдены новые источники утечки информации, что позволяет злоумышленнику расшифровать ранее записанный трафик, при условии использования RSA клиентом и сервером для обмена общим секретом. Вариант атаки MITM (с подменой шифронаборов) с возможностью подделать сообщения Finished, является более сложным в исполнении и зависит от скорости подбора. Авторы ROBOT получили награду Pwnie на Black Hat 2018 в категории «Лучшая криптографическая атака».

Профилактические меры: отказаться от использования шифронаборов, в которых применяется передача общего секрета с помощью RSA.

2.2.12 Downgrade Attack on TLS 1.3

Новое применение атаки Блейхенбахера (снова эксплуатирует применение RSA в режиме PKCS#1 v1.5 для обмена общим секретом), благодаря найденным уязвимостям в TLS библиотеках (CVE-2018-12404, CVE-2018-19608, CVE-2018-16868, CVE-2018-16869, and CVE-2018-16870).

Но основной интерес в данных исследованиях представляет то, что применение TLS1.3 (в которой RSA не используется) не спасает при поддержке более старых версий протокола, которые позволяют применять обмен premaster secret с помощью RSA, так как была продемонстрирована возможность понижения до нужной версии (downgrade attack), а также атака посредством TLS1.2 предоставляющая злоумышленнику возможность стать либо посредником в сессии, либо выдавать себя за валидный сервер и отвечать от его имени. Рассмотрим первый вариант. Для реализации данной атаки необходима поддержка клиентом и сервером RSA для обмена общим секретом. Злоумышленник, находящийся посередине, перехватывает сообщение ClientHello(v1.3) и отправляет на сервер свою версию ClientHello(v1.2) с шифронаборами, содержащими RSA (для exchange). После получает и отправляет ServerHello(v1.2, RSA) клиенту. Далее без изменений ретранслирует ServerCertificate и ServerHelloDone. На данный набор клиент отвечает ClientKeyExchange (содержит зашифрованный premaster secret открытым ключом сервера из сертификата). С этого момента злоумышленник проводит атаку на сервер с целью расшифровки данного сообщения, а для того, чтобы браузер не прервал соединение применяется отправка сообщения ChangeCipherSpec (сбросит таймер) и Alert, игнорируемые клиентом, но сбрасывающие таймер (в браузере Firefox позволяет подвесить сессию на продолжительное время). Далее если удалось узнать premaster secret, то рассчитываются сессионные ключи и подделывается сообщение Finished. На TLS1.2 атака с понижением до использования RSA для exchange производится аналогичным образом, только подделывается одно сообщение — ClientHello. Теперь рассмотрим вариант, в котором посредник не понижает используемую версию в сессии (между клиентом и сервером подразумевается TLS1.3). Для ее реализации достаточна поддержка сервером RSA для обмена секретом, а так как в версиях TLS до 1.3 это является соответствием стандарту, то скорее всего любой сервер с поддержкой какой-либо версии TLS до 1.3 подойдет. Итак первым сообщением согласно схеме Handshake в TLS1.3 выступает

ClientHello(v1.3) (передается в открытом виде), на которое злоумышленник никак не влияет, а просто пересылает серверу. Далее следует ответ ServerHello(v1.3) (также в открытом виде), который атакующий заменяет на свою версию (со своими параметрами (EC)DHE). EncryptedExtensions и Certificate от сервера передаются в неизменном виде. И тут включается механизм TLS1.3 рассчитанный на предотвращение такого рода вмешательств, а именно CertificateVerify, обеспечивающий раннюю проверку целостности данных. Собственно, здесь перед злоумышленником встает задача подделки данного сообщения, сделать это возможно если в качестве алгоритма подписи применялся RSA. Подделка подписи и расшифровка RSA со схемой PKCS являются довольно похожими. Атакующий устанавливает TLS соединение версии 1.2 с сервером с использованием RSA алгоритма для обмена premaster secret. После чего проводит атаку оракула, получая после этого результат шифрования от хеш-суммы всех предыдущих сообщений, что и будет являться подделанной подписью. Потом завершается процедура Handshake с клиентом и теперь злоумышленник с точки зрения жертвы выступает в роли аутентифицированного сервера.

Профилактические меры: отказаться от использования шифронаборов, в которых применяется передача общего секрета с помощью RSA; обновить, используемые версии библиотек TLS; если требуется поддержка обмена ключами с помощью RSA, то для этого должен использоваться открытый ключ, который не позволяет ставить цифровую подпись; использовать RSA с разрядностью не меньшей чем 2048 бит; короткий таймаут в браузере клиента.

2.2.13 0-Length OpenSSL

Зарегистрирована как CVE-2019-1559 [20]. Уязвимость заключается в том, что если приложение сталкивается с фатальной ошибкой, а затем дважды вызывает SSL_shutdown() для закрытия соединения (один раз для отправки сообщения alert close_notify и второй раз для получения ответа от клиента close_notify, при этом стандарт разрешает вызывать данную функцию один раз и не дожидаться ответа), то в данном случае OpenSSL может по-разному

реагировать на вызывающее приложение, так если получена 0-байтовая запись с недопустимым заполнением будет одна реакция, а если пришла 0-байтовая запись с недопустимым MAC (но верным заполнением) другая. При этом если приложение ведет себя по-разному, основываясь на результатах работы OpenSSL, то оно предоставляет удаленным узлам оракул заполнения, с помощью которого можно расшифровать данные. Для реализации необходимо, чтобы приложение вызывало `SSL_shutdown()` дважды, даже если произошла ошибка протокола. Уязвимость исправлена в OpenSSL 1.0.2r (версии подверженные уязвимости: 1.0.2-1.0.2q). Данная атака не затрагивает симметричные шифры в режиме AEAD.

Профилактические меры: обновить OpenSSL; использовать шифронаборы, включающие симметричные шифры в режиме AEAD.

2.2.14 GOLDENDOODLE и Zombie POODLE

Зарегистрирована в базе данных NIST NVD как CVE-2019-6485 и CVE-2019-6593 [21, 22]. Набор уязвимостей на основе padding oracle attack, выявленных в ходе нового исследования и сканирования самых популярных веб-сайтов. Представлены были в марте 2019 года на Black Hat Asia Крейгом Янгом. Суть их заключалась в том, что была выявлена проблема в существующих сканерах для обнаружения уязвимости POODLE (в частности, исследовалось поведение не свойственное реальной атаке). И таким образом были обнаружены некоторые оракулы для определенных сценариев: Zombie POODLE (неверное заполнение с верным MAC), GOLDENDOODLE (верное заполнение с неверным MAC).

GOLDENDOODLE аналогично POODLE использует утечку информации из криптосистемы о правильном дополнении, которое заменяется на блок с секретом. Но срабатывает в том случае, если сервер на неправильный MAC отвечает каким-либо образом отличным от ошибки заполнения. И очень важным нововведением GOLDENDOODLE выступает использование определенного байта (или байтов) MAC (так как он все равно подразумевается, как невалидный), в качестве предположения о символе, этот подход был

описан еще в 2003 году Сержем Воденэ. Таким образом возможно ускорить подбор секрета, из-за того, что теперь не нужно полагаться на случай, как в POODLE с игральным кубиком из 256 граней (там не производится проверка предположения о значении символа, а просто выжидается момент, при котором последний байт заполнения будет 0). Если POODLE предполагал проверку сервером только последнего байта дополнения (иначе атака не срабатывала, так как заполнение с огромной вероятностью будет неверным), то Zombie POODLE использует специфический оракул, возникающий при проверки однородности остальных байт заполнения (так уязвимые системы реагируют на ошибки MAC или заполнения в Application Data путем сброса TCP-соединения, но если MAC-адрес правильный, сервер сначала отправит предупреждение MAC TLS Bad Record MAC). Также стоит упомянуть о так называемом Sleeping POODLE (термин введен автором уязвимости), когда проведена проверка, не выявившая уязвимости POODLE, но использовала она для этого сообщение Finished (единственное зашифрованное сообщение Handshake в TLS до 1.3), а не записи Application Data.

Профилактические меры: обновить ПО; отказаться от использования шифронаборов, которые включают блочные шифры в режиме CBC; переходить на TLS1.3.

Приведем сводную таблицу уязвимостей TLS/SSL (табл. 1). Большое число современных атак эксплуатируют в качестве основы уязвимости, которые были известны еще 20 лет назад. Это становится возможным с одной стороны из-за того, что веб представляет собой огромное разнообразие программного обеспечения, поддерживающего разные версии протокола (часто устаревшие), а с другой стороны те изменения вводимые до версии 1.3 зачастую не решали проблемы, а лишь усложняли возможность их эксплуатации. TLS1.3 имеет совершенно другой подход, убирая основу на которых держались все найденные уязвимости. Собственно, на данный момент TLS1.3 является наилучшим решением не только в плане безопасности, что подтверждается современными исследованиями [23], но и скорости работы.

Табл. 1. Уязвимости TLS/SSL.

	Основа атаки	Эксплуатируемая уязвимость реализации	Требования	Что позволяет	Устранена ли уязвимость реализации	Как устранить
BEAST	Шифрование в режиме CBC	Использование в качестве вектора инициализации очередного сообщения последнего блока шифротекста предыдущего сообщения	MITM, а также возможность отправлять по защищенному каналу специальным образом сформированные сообщения	Расшифровать секретную часть сообщения (узнать куки, пароли и т. д.)	Да, а в стандарте TLS1.1 описан другой подход для формирования векторов инициализации	Обновить ПО для устранения уязвимости реализации, а лишить атаку основы можно исключив шифрование в режиме CBC (как вариант использовать AEAD шифрование)
CRIME	Сжатие данных в TLS	Применение сжатия	MITM и возможность отправлять данные не сервер по защищенному каналу от имени клиента	Узнать секретную часть сообщения (куки, пароли и т. д.)	Да, на данный момент сжатие TLS не используется	Не применять алгоритмы сжатия в TLS
BREACH	Сжатие данных в HTTP	Передача сжатого ответа сервера с секретом и значением пользовательского ввода в теле HTTP	MITM и возможность отправлять данные на сервер по защищенному каналу от имени клиента, причем секрет должен располагаться в теле HTTP	Узнать секретную часть сообщения, содержащуюся в теле HTTP	Потенциально данная атака до сих пор реализуема, все зависит от конкретного случая	Кардинальное решение: не использовать сжатие HTTP и более мягкое: отделить секрет от пользовательского ввода
Lucky13	Шифрование в режиме CBC	Время обработки данных на стороне сервера	Злоумышленник находится в той же локальной сети, что и атакуемая машина, а также используется в качестве алгоритма вычисления MAC HMAC-SHA1	Частичное или полное раскрытие зашифрованного сообщения	Да	Обновить ПО для закрытия ошибки реализации; использовать AEAD шифрование
Heartbleed	Уязвимость расширения heartbeat	Утечка незашифрованной информации из памяти сервера	Использование уязвимой версии расширения heartbeat библиотеки OpenSSL	Возможность получения секретных данных от конфиденциальной информации пользователей, до приватного ключа сервера	Да	Обновить OpenSSL
POODLE	Шифрование в режиме CBC	Оракулы дополнения; не производилась проверка всего заполнения	Атакующий способен переставлять блоки шифротекста внутри защищённых сообщений, а также может изменять часть открытого текста	Узнать секретную часть сообщения (куки, пароли и т. д.)	Да	Не использовать SSLv3; отказаться от шифронаборов, содержащих режим CBC
Logjam	Поддержка сервером экспортных шифронаборов	Использование групп DH малой разрядности	Поддержка сервером DHE_EXPORT	Стать посредником в сессии	Да	Обновить используемые браузеры; на стороне сервера исключить экспортные шифронаборы

Табл. 1. Уязвимости TLS/SSL.

	Основа атаки	Эксплуатируемая уязвимость реализации	Требования	Что позволяет	Устранена ли уязвимость реализации	Как устранить
DROWN	Применение RSA для передачи Premaster Secret от клиента серверу	Оракул Блейхенбахера	RSA для передачи Premaster Secret, а также поддержка сервером SSLv2	Узнать сессионные ключи и расшифровать ранее записанный трафик	Да	Отключить поддержку SSLv2 на сервере; обновить OpenSSL
Sweet32	Шифрование в режиме CBC	Использование блока размером 64 бита	Прослушивание и запись трафика	Частично или полностью раскрыть блок с зашифрованными данными	Так как уязвимость располагается на уровне протокола, то полностью ее устранить невозможно при использовании CBC, но были приняты меры по усложнению реализации Sweet32	Либо для усложнения реализации применять блоки 128 бит, либо отказаться от CBC
HEIST	Атака на длину сообщения с помощью размера окна tcp	API браузера, позволяющее зафиксировать получение первого и последнего байта ответа	Возможность отправки запроса с параметром, который будет включен в ответ	Узнать длину ответа	Нет, так как дальнейший сценарий атаки предполагает использования CRIME или BREACH	Отключить в браузере сторонние куки (third cookies); отключить поддержку сжатия на сервере
ROBOT	Применение RSA для передачи Premaster Secret от клиента серверу	Оракулы, такие как тайм-ауты, сброс подключения, дубликаты сообщений TLS alert	MITM	Узнать сессионные ключи и расшифровать ранее записанный трафик	Да	Отказаться от использования шифроработов, в которых применяется передача общего секрета с помощью RSA
Downgrade Attack on TLS 1.3	Применение RSA для передачи Premaster Secret от клиента серверу	Оракулы RSA PKCS#1 v1.5	MITM	Позволяет либо узнать сессионные ключи, либо стать посредником в сессии	Да	Отказаться от использования шифроработов, в которых применяется передача общего секрета с помощью RSA; обновить, используемые версии библиотек TLS
0-Length OpenSSL	Уязвимая версия OpenSSL	Разная реакция на 0-байтовую запись	Возможность переставлять блоки шифротекста	Расшифровать секретную часть сообщения (узнать куки, пароли и т. д.)	Да	Обновить OpenSSL; использовать шифроработы, включающие симметричные шифры в режиме AEAD
GOLDENDOODLE, Zombie POODLE	Шифрование в режиме CBC	Оракулы дополнения	Возможность влиять на зашифрованный трафик (переставлять блоки и т. д.)	Расшифровать секретную часть сообщения (узнать куки, пароли и т. д.)	Да	Обновить ПО; отказаться от использования шифроработов, которые включают блочные шифры в режиме CBC

3 ИССЛЕДОВАНИЕ И КОНФИГУРАЦИЯ

3.1 Обзор самых распространенных библиотек TLS/SSL, DTLS

Приведем сводную таблицу библиотек TLS/SSL (табл. 2), а затем разберем каждую более подробно.

Табл. 2. Библиотеки TLS/SSL (на 30.05.2019).

	Самая последняя версия	Лицензия	Open source	TLS1.3	DTLS	DANE
GnuTLS	3.6.8	LGPLv2.1 +	Да	Да	Да	Да
wolfSSL	4.0.0	GPLv2 и стандартное коммерческое лицензирование	Да	Да	Да	Нет
mbed TLS	2.16.1	Apache License v2; GPLv2	Да	Нет	Да	Нет
BearSSL	0.6	MIT License	Да	Нет	Нет	Нет
OpenSSL	1.1.1c	Apache License v2	Да	Да	Да	Да

3.1.1 GnuTLS

GnuTLS - библиотека, реализующая протоколы SSL, TLS и DTLS, а также технологии с ними связанные. Предоставляет простой интерфейс прикладного программирования на языке C (API) для доступа к протоколам защищенной связи, а также API для чтения и записи X.509, PKCS #12 и других требуемых структур.

Поддерживает: все версии протокола TLS (включая 1.3), SSLv3, DTLS1.0, DTLS1.2; технологии DANE и TOFU; протокол OCSP. Работает на большинстве платформ Unix и Windows. Использует лицензию GNU Lesser General Public License версии 2.1 (LGPLv2.1 +).

3.1.2 wolfSSL

Библиотека wolfSSL - легкая библиотека TLS/SSL, написанная на языке C, предназначена для IoT (Интернет вещей) и RTOS (операционная система

реального времени) в связи с размером, скоростью и набором функций. Поддерживает отраслевые стандарты вплоть до текущих TLS 1.3 и DTLS 1.2, в 20 раз меньше OpenSSL, предоставляет простой API, поддерживает OCSP и CRL, работает на основе криптографической библиотеки wolfCrypt.

Программное обеспечение wolfSSL доступно в двух разных моделях лицензирования: с открытым исходным кодом (по GPLv2) и стандартное коммерческое лицензирование, в случае использования в ее в коммерческих целях.

3.1.3 mbed TLS

mbed TLS (ранее PolarSSL) - реализация протоколов TLS и SSL, а также соответствующих криптографических алгоритмов и необходимого кода поддержки. Имеет двойную лицензию Apache License версии 2.0 (также доступна версия GPLv2). Написана на языке C. В отличие от OpenSSL и других реализаций TLS, mbed TLS похожа на wolfSSL в том, что она предназначена для установки на небольшие встроенные устройства, с минимальным полным стеком TLS, требующим менее 64 КБ ОЗУ.

3.1.4 BearSSL

BearSSL является реализацией протоколов TLS/SSL, написанного на C. Особенности BearSSL: небезопасные версии протоколов и выбор алгоритмов не поддерживаются; минимальная реализация серверной части может занимать около 20 КБ скомпилированного кода и 25 КБ оперативной памяти; может применяться в небольших встроенных системах и даже в специальных контекстах, таких как загрузочный код. На данный момент поддерживает TLS1.0, TLS1.1 и TLS1.2. Пока существует только как бета-версия.

3.1.5 OpenSSL

OpenSSL — это надежный, коммерческий и полнофункциональный инструментарий для протоколов TLS и SSL. Также выступает в роли криптографической библиотеки общего назначения. Распространяется по лицензии Apache, что разрешает свободно получать и использовать ее в

коммерческих и некоммерческих целях при соблюдении условий лицензии. Поддерживает TLS1.3, TLS1.2, TLS1.1, TLS1.0, SSLv3, DTLS1.2, DTLS1.0.

3.2 Дополнительные механизмы проверки SSL сертификатов

В связи с тем, что безопасность современного веба очень сильно зависит от сертификатов, а соответственно и от инфраструктуры, позволяющей их проверить. Были созданы дополнительные механизмы проверки валидности сертификатов.

Trust on first use (TOFU) (применяется в SSH) - концепция, заключающаяся в том, что открытый ключ узла не проверяется при первом соединении, а сохраняется его отпечаток для последующей проверки, при повторном соединении с тем же самым узлом происходит проверка его отпечатка с отпечатком сохраненным до этого и выдвигается требование о их совпадении. Такая система в сочетании с обычной проверкой сертификата CA и проверкой отзыва OCSP может помочь обеспечить многофакторную проверку, для которой отказ одной составляющей не приведет к компрометации соединения. В HTTPS может применяться аналогичный механизм: HTTP Public Key Pinning (HPKP).

DANE (DNS-based Authentication of Named Entities) — технология, которая позволяет проверить валидность сертификата с помощью инфраструктуры DNSSEC. В набор ресурсных записей DNS вводится новый тип TLSA, обеспечивающий соответствие между именем хоста и его сертификатом. Т. е. клиент сравнивает отпечаток пришедшего сертификата с тем, что он получил от DNS сервера.

Протокол OCSP (Online Certificate Status Protocol) — позволяет проверить статус SSL сертификата (данный протокол пришел на замену используемых ранее списков CAC (или CRL списков), в которых содержалась информация о отозванных SSL сертификатах). Принцип работы: пользователь посылает запрос серверу для получения сведений о состоянии сертификата. В ответ он получает подписанное центром сертификации OCSP сообщение, которое может содержать следующее: good — означает, что SSL сертификат не

отозван и не заблокирован; `revoked` — сертификат отозван; `unknown` — не удалось установить статус сертификата, так как серверу не известен его издатель. При использовании OCSP Stapling сервер с сертификатом присылает его состояние, запрошенное у УЦ.

3.3 Обзор и работа с инструментами для исследования TLS/SSL

3.3.1 SageMath

SageMath — это бесплатное математическое программное обеспечение с открытым исходным кодом, распространяющаяся по лицензии GPL. Создано на основе многих существующих пакетов с открытым исходным кодом: NumPy, SciPy, matplotlib, SymPy, Maxima, GAP, FLINT, R и многих других. Собственно, для TLS/SSL SageMath интересен тем, что позволяет поработать с криптографическими объектами. Например, можно рассмотреть одну из стандартных групп, которые используются в TLS1.3, или попробовать придумать свою кривую и определить разрядность ей соответствующую и т. д.

3.3.2 Wireshark

Wireshark — программа анализатор трафика. Имеет удобный графический интерфейс. В Wireshark HTTPS трафик будет отображаться в зашифрованном виде, так как-будто вы находитесь в положении MITM. Поэтому для более продуктивного изучения TLS/SSL рекомендуется научиться добавлять секрет (о значении которого договорились браузер и сервер) в Wireshark для того, чтобы можно было увидеть истинное содержимое TLS-записей. Приступим к настройке. Для начала необходимо настроить логирование секретной информации (данную функцию поддерживает браузер Firefox). Чтобы это сделать нужно добавить переменную окружения `SSLKEYLOGFILE` со значением пути к файлу, в который будут записываться логи. В Windows это делается через `Advanced system settings`, а затем `Environment Variables`, добавляется новая переменная окружения. В Linux и Mac OS X необходимо открыть терминал, получить права суперпользователя, а затем ввести данную команду:

```
# export SSLKEYLOGFILE=/Путь_к_файлу/Имя_файла
```

```
# firefox
```

Важно отметить, что запуск Firefox в Linux нужно производить из той же сессии, поэтому сразу пишем `firefox` (в качестве аргумента можно тут же передать адрес веб-сайта, который хотим посетить).

После установки переменной окружения запускаем Wireshark, ловим пакеты, посещая нужные сайты в Firefox. Теперь у нас имеется зашифрованный трафик HTTPS, для его расшифровки укажем Wireshark, где находится лог-файл с секретом. Нажимаем Edit → Preferences → Protocols, находим TLS и в поле (Pre)-Master-Secret log filename указываем путь к файлу с логами Firefox, нажимаем ОК. После этого нам доступно расшифрованное содержимое TLS-записей. Это очень удобно особенно при работе с TLS1.3, так как шифрование начинается очень рано и без расшифровки не удастся рассмотреть даже полный механизм Handshake.

3.3.3 Сканеры уязвимостей

SSL Labs на своем сайте (www.ssllabs.com) предоставляет несколько интересных бесплатных онлайн-сервисов для анализа защищенности TLS/SSL. Так там представлен тест, выполняющий глубокий анализ конфигурации любого SSL веб-сервера в Интернете. Достаточно указать домен в поле `hostname` и через некоторое время будет получен отчет, сформированный в результате анализа. Отчет включает в себя итоговый рейтинг от A+ до F (обозначает, что сервер уязвим к одной из известных атак), информацию о сертификате, поддерживаемые версии TLS/SSL, поддерживаемые шифронаборы, совместимость с различным клиентским ПО, подверженность известным атакам и многое другое. Кроме уязвимых отмечаются слабые места, которые можно усилить. Также присутствует сканер браузера, показывающий поддерживаемые версии протокола, а также подверженность к различного рода уязвимостям. Стоит отметить, что на сайте SSL Labs приводится интересная статистика, отражающая качество поддержки TLS/SSL с течением времени на выборке из 150 000 веб-сайтов с

поддержкой SSL и TLS, согласно списку самых популярных сайтов в мире Alexa.

3.3.4 Openssl, nmap

Нельзя не упомянуть утилиту openssl, которая позволяет из командной строки подключаться к веб-серверам с определенными параметрами. А именно можно определить версию протокола, которая будет использоваться или список шифронаборов. Пример использования:

```
openssl s_client -connect www.google.com:443 -tls1
```

Также для проверки поддерживаемых версий протокола, шифронаборов и подверженности уязвимостям может выступать nmap. Пример использования:

```
nmap --script ssl-enum-ciphers -p 443 www.example.com
```

На сайтах, посвященных уязвимостям, авторы часто оставляют ссылки на сканеры, позволяющие проверить веб-сервер, а также по их исходному коду можно лучше разобраться в проблеме. Так для проверки на GOLDENDOODLE и Zombie POODLE можно использовать padcheck (написан на Go) [24], созданный автором атак.

3.4 Сканер, поддерживаемых версий TLS/SSL

Был написан сканер на языке C++ с использованием библиотеки OpenSSL. На вход он принимает файл, содержащий список веб-сайтов (можно ip адреса, а можно доменные имена), которые нужно просканировать на поддержку различных версий TLS/SSL. На выходе получаем строки вида:

```
Сетевое_имя 4:yes/no 3:yes/no 2:yes/no 1:yes/no 0:yes/no
```

где 4 — TLS1.3, 3 — TLS1.2, 2 — TLS1.1, 1 — TLS1.0, 0 — SSLv3 соответственно. Метка yes или no — показывает поддерживается ли данная версия протокола сервером или нет.

Рассмотрим подробнее как работает программа. Сначала считываются домены или ip адреса из файла, путь к которому пользователь передает через аргумент при запуске (в коде argv[1]). Далее необходимо сформировать контекст (в терминах OpenSSL SSL_CTX), представляющий из себя

практически тоже самое, что обсуждалось выше при описании контекста TLS. Т. е. он будет в себя включать поддерживаемые версии протокола, поддерживаемые шифронаборы, клиентский сертификат (если будет использоваться) и т. д. Соответственно в нашем случае загружаются все доступные варианты шифронаборов, указывается путь к файлу с сертификатами корневых удостоверяющих центров. Так как будет проводиться не только проверка версий TLS, но и SSL, то необходимо будет создать два контекста (возможны и другие подходы, реализуемые через один контекст, но в данном случае был выбран вариант с двумя). Первый контекст создается с `TLS_client_method`, а второй с `SSLv3_method`. После указания пути к файлу с сертификатами удостоверяющих центров (использовался файл из Mozilla доступный на официальном сайте) начинается цикл, проходящий по всем серверам, которые нужно проверить. В контексте TLS устанавливается максимальная поддерживаемая версия, сначала TLS1.3, а затем в зависимости от ответа сервера остальные, т. е. если сервер выбрал TLS1.1, то следующая максимальная версия будет TLS1.0. Далее происходит подключение к серверному порту 443 причем, необходимо отслеживать установление TCP соединения, так как может произойти заикливание, если сервер не отвечает на клиентское SYN, в этом случае соединение прерывается (т. к. сервер недоступен, в этом случае метка будет `timeout_error`). Непосредственно для осуществления передачи и получения данных от сервера используется BIO (Basic I/O abstraction) с вызовом `BIO_new_ssl_connect()` во время инициализации. Производится процедура Handshake, проверяется валидность сертификата и если все прошло успешно, то считается, что сервер поддерживает данную версию протокола. После тоже самое выполняется для контекста SSL только проверяется единственная версия SSLv3. В случае, когда не удалось удостовериться в валидности сертификата напротив идентификатора версии протокола будет `error-7` (т. е. возможно был прислан самоподписанный сертификат или срок его действия истек и т. п.).

3.4.1 Результаты сканирования

Сканированию подверглись топ 100000 веб-сайтов по версии рейтинга Alexa.

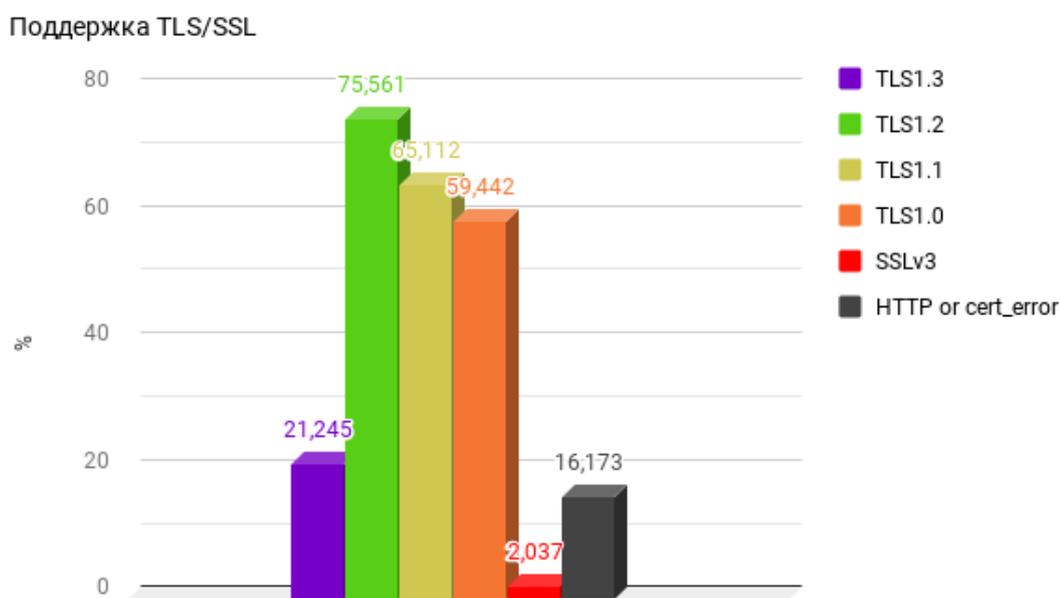


Рис. 9. Диаграмма поддерживаемых версий TLS/SSL топ 100000 Alexa

Рис. 9 отображает процентное соотношение поддерживаемых версий от общего числа на 25.05.2019. Можно заметить, что достаточно большое число сайтов либо вообще не поддерживают TLS/SSL (столбец HTTP or cert_error, либо во время проверки их сертификатов не удалось удостовериться в подлинности (возможно использовались самоподписанные или истекшие). Около двух процентов поддерживают SSLv3, из них 20 поддерживали только данный протокол и 297 TLS1.0 и SSLv3.

Также было проведено сканирование топ 1842 сайтов в России по версии liveinternet.ru (для которого был написан парсер на python). Приведем результаты на 24.05.2019 TLS1.3: 10,97 %; TLS1.2: 83,2 %; TLS1.1: 81,5 %; TLS1.0: 81,9 %; SSLv3: 2,8 %; HTTP or cert_error: 13,2 %. Сайтов, которые поддерживают только SSLv3 не было обнаружено, но были 10 только с TLS1.0 и SSLv3.

На основании этих данных можно сделать вывод, что достаточно большое количество сайтов либо поддерживают устаревшие версии протокола, либо вообще не поддерживают данную технологию. Между тем переход на

TLS1.3 идет не очень быстрыми темпами, даже несмотря на то, что сейчас последние версии веб-серверов по умолчанию его поддерживают.

3.5 Настройка веб-сервера

Будет представлено описание конфигурации веб-серверов Apache и Nginx с акцентом на TLS/SSL. Настройка производилась на Ubuntu 19.04. Так как для поддержки TLS серверу необходимо иметь сертификат, то мы сгенерируем его сами и сами же его подпишем, получится так называемый самоподписанный сертификат (в реальных условиях сертификат должен подписывать удостоверяющий центр), поэтому браузеры не будут ему доверять, из-за чего придется добавлять его в исключения или загружать страницу не смотря на «риски», собственно в исследовательских целях такой сценарий является приемлемым. Генерировать будем ECDSA сертификат, который в отличие от RSA обеспечивает тот же уровень безопасности при меньшем размере, а также проверка подписи ECDSA выполняется быстрее чем RSA. Но для сохранения совместимости с устаревшими устройствами рекомендуется использовать два сертификата один ECDSA другой RSA, в такой конфигурации в большинстве случаев будет применяться ECDSA, который быстрее, и очень редко RSA с устаревшими клиентами. Выводим список поддерживаемых эллиптических кривых:

```
openssl esparam -list_curves
```

Генерируем приватный ключ с использованием кривой prime256v1:

```
openssl esparam -out server.key -name prime256v1 -genkey
```

Создаем самоподписанный сертификат:

```
openssl req -new -key server.key -x509 -nodes -days 365  
-out server.pem
```

Перемещаем приватный и публичный ключ:

```
mv server.pem /etc/ssl/certs/  
mv server.key /etc/ssl/private/
```

В свете описанной ранее атаки Logjam рекомендуется генерировать и использовать свою группу (не менее 2048 бит) для алгоритма классического Диффи-Хеллмана, создаем:

```
openssl dhparam -out dhparams.pem 2048
```

3.5.1 Apache

Веб-сервер Apache является одним из самых популярных серверов в Интернете с 1996 года. Он имеет отличную документацию и поддерживает интеграцию с другими проектами. Apache выбирают за его гибкость и мощь, в нем очень многое доступно сразу из коробки. Приступим к настройке.

Первым делом необходимо установить Apache:

```
apt-get install apache2
```

Далее включаем ssl модуль, т. е. появляется поддержка TLS/SSL и перезапускаем веб-сервер:

```
a2enmod ssl && a2ensite default-ssl && systemctl restart apache2
```

Открываем файл `/etc/apache2/sites-enabled/default-ssl.conf` находим в нем `SSLCertificateFile`, `SSLCertificateKeyFile` и указываем в качестве их значений путь к сертификату и путь к приватному ключу соответственно. С этого момента наш сервер поддерживает TLS/SSL, для проверки можно открыть содержимое нашего сайта (которое в данный момент будет представлять из себя страницу по умолчанию) через браузер, вбив `https://ip_адрес:443` и игнорировав все предупреждения. Для того, чтобы отключить отображение информации о сервере при ошибках в файле `/etc/apache2/conf-available/security.conf` прописываем: `ServerTokens Prod` и `ServerSignature Off`. Теперь настроим редирект с 80 порта на 443, чтобы оставить только HTTPS: вводим команду `2enmod alias`, а затем в файл `/etc/apache2/sites-available/000-default.conf` добавляем строку `Redirect / https://ip_адрес_сервера:443/`

Все дальнейшие наши манипуляции будут происходить в файле `/etc/apache2/sites-enabled/default-ssl.conf`. На данный момент в версии Apache2

2.4.38 TLS1.3 поддерживается по умолчанию, а SSLv2 и SSLv3 отключены. Меня значения поля `SSLProtocol` можно настроить поддерживаемые версии (приведем пример: `SSLProtocol all -TLsv1 -SSLv3 -SSLv2`, что означает все версии кроме TLSv1, SSLv3 и SSLv2).

`SSLHonorCipherOrder on` — шифронаборы будут выбираться по предпочтению сервера.

`SSLOpenSSLConfCmd DHParameters "path_to_dhparams.pem"` — указываем путь к ранее сгенерированной группе классического Диффи-Хеллмана.

С помощью `SSLCipherSuite` можно настроить поддерживаемые шифронаборы (пример: `SSLCipherSuite HIGH:!aNULL:!MD5` — только стойкие шифры). Включение OCSP Stapling:

```
SSLUseStapling On
SSLStaplingCache "shmcb:logs/ssl_stapling(32768)"
```

Но при использовании самоподписанных сертификатов OCSP Stapling включать не нужно.

Включаем HSTS (заставляет браузер использовать только HTTPS при работе с сайтом в течении `time` секунд):

```
Header always set Strict-Transport-Security "max-age=time; includeSubdomains;"
```

3.5.2 Nginx

Nginx достиг популярности благодаря небольшому размеру (т. е. в нем изначально нет ничего, что может оказаться лишним), а также способности простого масштабирования на минимальном аппаратном обеспечении. Администраторы отдают предпочтение Nginx из-за эффективности использования ресурсов и малой скорости отклика под нагрузкой. Перейдем к настройке.

Установим Nginx:

```
apt-get install nginx
```

Теперь наш сайт с стандартной страницей работает и, чтобы его увидеть необходимо обратиться на 80 порт. Все дальнейшие настройки будут проводиться в файле `/etc/nginx/sites-available/default` (его можно переименовать например на `example.com`). Настроим поддержку TLS/SSL. Найдем блок `server` и в нем либо прокомментируем (добавляем `#` перед строкой), либо удалим строки:

```
listen 80 default_server;
listen [::]:80 default_server;
```

И добавим:

```
listen 443 ssl default_server;
listen [::]:443 ssl default_server;
ssl_certificate    "path_to_cert";
ssl_certificate_key "path_to_private_key";
```

Теперь наш веб-сервер поддерживает SSL. Настроим редирект на 443 порт при обращении к 80. Для этого добавим еще один блок `server`:

```
server {
    listen 80 default_server;
    listen [::]:80 default_server;
    return 301 https://$host$request_uri;
}
```

По умолчанию в Nginx версии 1.15.9 включен TLS1.3, а также не используются SSLv3 и SSLv2. Для настройки поддерживаемых версий, а также шифронаборов применяются директивы `ssl_protocols` и `ssl_ciphers` соответственно. Пример:

```
ssl_protocols    TLSv1.1 TLSv1.2 TLSv1.3;
```

Перечисляются только поддерживаемые версии, таким образом будут доступны TLS1.1, TLS1.2, TLS1.3.

```
ssl_ciphers    HIGH:!aNULL:!MD5;
```

Оставляет только сильные шифронаборы (в таком виде она присутствует неявно по умолчанию, поэтому ее можно не добавлять в данной вариации).

Для использования предпочтения сервера при выборе шифронаборов добавляем:

```
ssl_prefer_server_ciphers on;
```

Указываем путь к ранее сгенерированной группе классического Диффи-Хеллмана:

```
ssl_dhparam "path_to_dhparams.pem";
```

При желании можно добавить поддержку 0-RTT, что заметно увеличит скорость, но необходимо помнить, что данный режим не обладает прогрессивной секретностью. Для включения и защиты от replay attacks:

```
ssl_early_data on;
```

```
proxy_set_header Early-Data $ssl_early_data;
```

По умолчанию 0-RTT отключена `ssl_early_data off;`

Включение OCSP Stapling:

```
ssl_stapling on;
```

```
ssl_stapling_verify on;
```

Но при использовании самоподписанных сертификатов OCSP Stapling включать не нужно.

Включаем HSTS:

```
add_header Strict-Transport-Security "max-age=time;  
includeSubDomains";
```

Приведем ключевые моменты конфигурации веб-сервера Nginx, которые были выполнены с уклоном на совместимость, безопасность и скорость, и в совокупности дали рейтинг A+ при сканировании на сайте sslabs (рис. 10):

- Совместное использование двух сертификатов ECDSA (384 бит) и RSA (4096 бит).
- Поддерживаемые протоколы TLS1.0, TLS1.1, TLS1.2, TLS1.3.
- Отключены шифронаборы с RSA для передачи секрета.
- Выбор шифронабора основан на предпочтении сервера.
- Применение Strict Transport Security (HSTS).
- Не используется OCSP stapling и 0-RTT.

- Для DHE сгенерирована группа (3096 бит).
- Поддерживается восстановление сессии.



Рис. 10. Результаты сканирования на сайте sslslabs

3.6 Общие рекомендации

В свете описанных в главе 2 уязвимостей можно сделать определенные выводы. С точки зрения версий, абсолютно необходимо исключить SSLv3 и SSLv2, так как даже простая поддержка их на ряду с другими более свежими, может привести к серьезным проблемам с безопасностью, примером тому может выступать DROWN. С точки зрения шифронаборов, основные проблемы заключались с режимом CBC, передачей общего секрета с помощью RSA, а также с поддержкой сжатия как на уровне TLS, так и на уровне HTTP. Соответственно стоит исключить все шифронаборы, позволяющие передавать premaster secret за счет RSA, особенно вспомнив опыт атаки Блейхенбахера, DROWN, ROBOT и Downgrade Attack. Сжатие на данный момент не поддерживается, но нужно иметь ввиду, что ее использование небезопасно (CRIME, BREACH, HEIST). Режим CBC уже неоднократно подвергался успешным атакам BEAST, POODLE, GOLDENDOODLE и т. д. Что позволяет говорить о потенциальном риске при использовании шифрования в режиме CBC. В качестве серверного сертификата лучше применять ECDSA сертификаты, так как они меньше и проверка цифровой подписи требует меньше вычислений, ну и также исключается возможность их использования для передачи общего секрета. Для классического ДН необходимо генерировать новую группу, а не использовать стандартные, причем разрядность модуля должна быть не менее 2048 бит, в необходимость этого заставляет верить атака

Logjam. И конечно нужно обязательно переходить на поддержку TLS1.3, так как все описанные выше проблемы в нем устранены. Для пользователей можно рекомендовать включить проверку статуса сертификата посредством OCSP запросов, а в будущем, когда инфраструктура DNSSEC достигнет больших масштабов, то использовать технологию DANE.

ЗАКЛЮЧЕНИЕ

В данной работе был исследован протокол TLS1.3, а также более ранние версии, кроме того, приводится описание DTLS. Рассмотрена логика их работы и последовательность изменений при переходе между версиями, помимо этого представлены последствия применения устаревших версий. Описаны основные известные на сегодняшний день уязвимости и приведены профилактические меры по их устранению.

Разработан сканер серверов на поддержку различных версий TLS/SSL с использованием библиотеки OpenSSL, с помощью которого исследовано более 100000 веб-сайтов и произведен анализ результатов.

Дается обзор как инструментов для изучения и углубления знаний данной технологии, так и основных библиотек, позволяющих создавать свои приложения с поддержкой TLS/SSL.

Описана инструкция по настройке двух самых популярных веб-серверов Apache и Nginx с упором на TLS. Приводятся рекомендации по настройке параметров протокола для обеспечения наивысшей степени безопасности.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Александр Венедюхин. Ключи, шифры, сообщения: как работает TLS [Электронный ресурс]. URL: <https://tls.dxdt.ru/tls.html> (дата обращения 2.05.2019).
2. Datagram Transport Layer Security Version 1.2 [Электронный ресурс] // RFC 6347 January 2012. URL: <https://tools.ietf.org/html/rfc6347> (дата обращения 5.05.2019).
3. The Transport Layer Security (TLS) Protocol Version 1.2 [Электронный ресурс] // RFC 5246 August 2008. URL: <https://tools.ietf.org/html/rfc5246> (дата обращения 5.05.2019).
4. The Transport Layer Security (TLS) Protocol Version 1.1 [Электронный ресурс] // RFC 4346 April 2006. URL: <https://tools.ietf.org/html/rfc4346> (дата обращения 4.05.2019).
5. The Transport Layer Security (TLS) Protocol Version 1.3 [Электронный ресурс] // RFC 8446 August 2018. URL: <https://tools.ietf.org/html/rfc8446> (дата обращения 5.05.2019).
6. Ivan Ristić. SSL Threat Model [Электронный ресурс]. URL: <https://blog.ivanristic.com/2009/09/ssl-threat-model.html> (дата обращения 6.05.2019).
7. В. Олифер, Н. Олифер. Компьютерные сети. Принципы, технологии, протоколы: Учебник для вузов. 5-е изд. — СПб.: Питер, 2016. — 992 с.: ил. — (Серия «Учебник для вузов»).
8. NIST NVD CVE-2011-3389 [Электронный ресурс]. URL: <https://nvd.nist.gov/vuln/detail/CVE-2011-3389> (дата обращения 7.05.2019).
9. NIST NVD CVE-2012-4929 [Электронный ресурс]. URL: <https://nvd.nist.gov/vuln/detail/CVE-2012-4929> (дата обращения 7.05.2019).
10. NIST NVD CVE-2014-0160 [Электронный ресурс]. URL: <https://nvd.nist.gov/vuln/detail/CVE-2014-0160> (дата обращения 7.05.2019).
12. Encryption using the Cipher Block Chaining (CBC) mode [Электронный ресурс]. URL:

[https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation#/mediaFile:CBC_encryption.svg](https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation#/media/File:CBC_encryption.svg) (дата обращения 6.05.2019).

13. Decryption using the Cipher Block Chaining (CBC) mode [Электронный ресурс]. URL:

[https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation#/mediaFile:CBC_decryption.svg](https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation#/media/File:CBC_decryption.svg) (дата обращения 6.05.2019).

14. NIST NVD CVE-2014-3566 [Электронный ресурс]. URL: <https://nvd.nist.gov/vuln/detail/CVE-2014-3566> (дата обращения 7.05.2019).

15. NIST NVD CVE-2013-0169 [Электронный ресурс]. URL: <https://nvd.nist.gov/vuln/detail/CVE-2013-0169> (дата обращения 7.05.2019).

16. NIST NVD CVE-2016-0800 [Электронный ресурс]. URL: <https://nvd.nist.gov/vuln/detail/CVE-2016-0800> (дата обращения 7.05.2019).

17. NIST NVD CVE-2016-2183 [Электронный ресурс]. URL: <https://nvd.nist.gov/vuln/detail/CVE-2016-2183> (дата обращения 7.05.2019).

18. NIST NVD CVE-2016-7153 [Электронный ресурс]. URL: <https://nvd.nist.gov/vuln/detail/CVE-2016-7153> (дата обращения 7.05.2019).

19. NIST NVD CVE-2017-13099 [Электронный ресурс]. URL: <https://nvd.nist.gov/vuln/detail/CVE-2017-13099> (дата обращения 7.05.2019).

20. NIST NVD CVE-2019-1559 [Электронный ресурс]. URL: <https://nvd.nist.gov/vuln/detail/CVE-2019-1559> (дата обращения 7.05.2019).

21. NIST NVD CVE-2019-6485 [Электронный ресурс]. URL: <https://nvd.nist.gov/vuln/detail/CVE-2019-6485> (дата обращения 7.05.2019).

22. NIST NVD CVE-2019-6593 [Электронный ресурс]. URL: <https://nvd.nist.gov/vuln/detail/CVE-2019-6593> (дата обращения 7.05.2019).

23. Cas Cremers, Marko Horvat, Jonathan Hoyland, Sam Scott, Thyla van der Merwe. A Comprehensive Symbolic Analysis of TLS 1.3. March 7, 2019.

24. Padcheck: A TLS CBC Padding Oracle Scanner [Электронный ресурс]. URL: <https://github.com/Tripwire/padcheck> (дата обращения 8.05.2019).