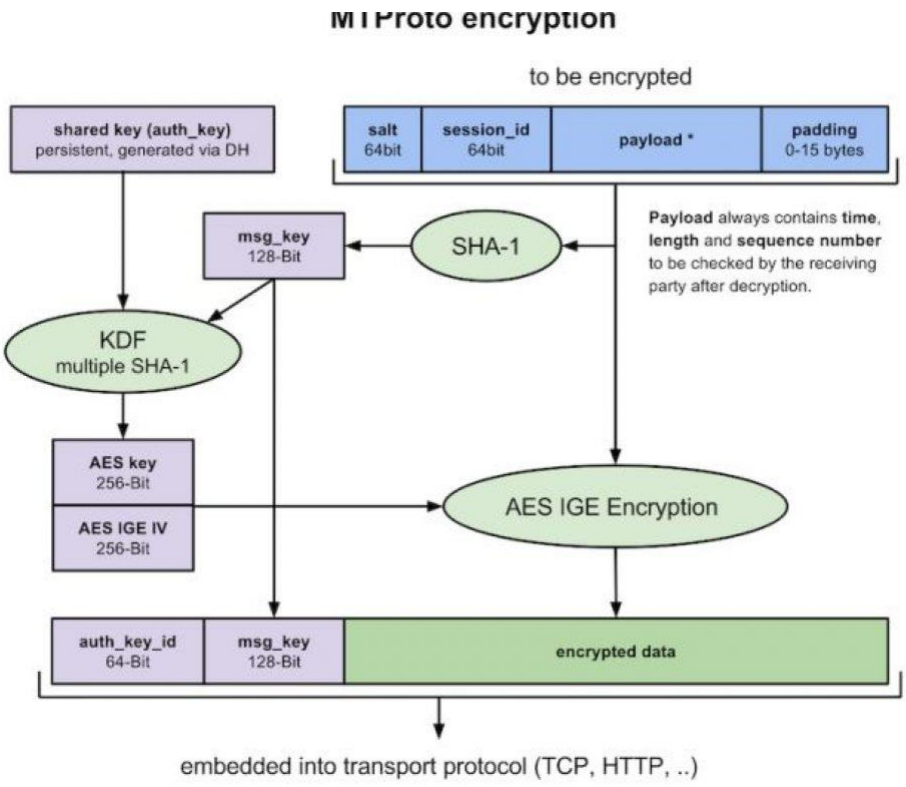
Содержание

Теоретическое описание протокола MTProto

Протокол предназначен для доступа к серверному API с приложений, запущенных на мобильных устройствах. Подчеркнем, что интернет-браузер не считается таким приложением.

Протокол разбит на три почти независимых части:

- Высокоуровневая часть (язык запросов к API) — определяет, каким образом запросы к API и ответы на эти запросы преобразуются в двоичные *сообщения*.
- Криптографическая (авторизационная) прослойка — определяет, каким образом сообщения шифруются перед передачей через транспортный протокол.
- Транспортная часть — определяет, каким образом передаются сообщения между клиентом и сервером поверх какого-либо другого существующего сетевого протокола (например, http, https, tcp, udp).



Примечание 1: Каждое текстовое сообщение, которое нужно зашифровать через MTProto, всегда содержит следующие данные, которые проверяются расшифровкой, чтобы сделать систему устойчивой против известных проблем с компонентами:

- server salt (соль сервера) (64-битная)
- session id (идентификатор сессии)
- message sequence number (порядковый номер сообщения)
- message length (длина сообщения)
- time (время)

Краткий обзор компонентов

Высокоуровневая часть (язык RPC-запросов/API)

С точки зрения высокоуровневой части, клиент и сервер обмениваются сообщениями в рамках некоторой сессии. Сессия привязана к клиентскому устройству (вернее, приложению), но не к конкретному http/https/tcp-соединению. Кроме того, каждая сессия привязана к идентификатору пользовательского ключа, по которому фактически производится авторизация.

Может быть открыто несколько соединений к серверу; сообщения в ту или иную сторону могут идти по любому из них (ответ на запрос не обязан прийти по тому же соединению, по которому был отправлен сам запрос, хотя чаще всего это так; однако ни в коем случае сообщение не может быть возвращено в соединении, принадлежащем другой сессии). При использовании UDP-протокола может случиться, что ответ на запрос приходит не с того IP, на который был отправлен запрос.

Сообщения бывают нескольких типов:

- RPC-вызовы (от клиента к серверу) — обращения к методам API
- RPC-результаты (от сервера к клиенту) — результаты RPC-вызовов
- Подтверждение приема сообщений (вернее, уведомление о состоянии набора сообщений)
- Запрос состояния сообщений
- *Составное сообщение или контейнер* (контейнер, содержащий несколько сообщений; нужен, например, чтобы по HTTP-соединению можно было отправить несколько RPC-вызовов сразу; кроме того, контейнер может поддерживать gzip).

С точки зрения протоколов более низкого уровня, сообщение — это поток двоичных данных, выровненный по границе 4 или 16 байтов. Первые несколько полей сообщения фиксированы и используются системой криптографии/авторизации.

Каждое сообщение, отдельное или внутри контейнера, состоит из *идентификатора сообщения* (64 бита; см. ниже), *порядкового номера сообщения в сессии* (32 бита), *длины* (тела в байтах; 32 бита) и *тела* (любой размер, кратный 4 байтам). Кроме того, при отправке контейнера или одиночного сообщения, в его начало дописывается *внутренний заголовок* (см. ниже), после чего все это шифруется, и в начало зашифрованного сообщения добавляется *внешний заголовок* (64-битный *идентификатор ключа* и 128-битный *ключ сообщения*).

Тело сообщения обычно состоит из 32-битного *типа сообщения*, за которым следуют *параметры*, зависящие от типа. В частности, каждой RPC-функции соответствует свой тип сообщения. Более подробно читайте в статье про двоичную сериализацию данных и служебные сообщения.

Все числа записываются как little-endian. Однако очень большие числа (2048-битные), используемые в RSA и DH, записываются как big-endian, потому что так делает библиотека OpenSSL.

Авторизация и криптография

Перед передачей сообщений (или составных сообщений) по сети посредством транспортного протокола они шифруются определенным образом; при этом перед сообщением приписывается внешний заголовок: 64-битный *идентификатор ключа* (однозначно определяющий *авторизационный ключ* для сервера, а также *пользователя*) и 128-битный *ключ* сообщения. Пользовательский ключ вместе с ключом сообщения определяют реальный 256-битный ключ, которым и зашифровано сообщение посредством шифра AES-256. Начало тела незашифрованного сообщения содержит некоторые данные (сессию, идентификатор сообщения, порядковый номер сообщения в сессии, серверную соль); *ключ сообщения* должен совпадать с младшими 128 битами SHA1 от тела сообщения (включая сессию, идентификатор сообщения и т.п.). Составные сообщения шифруются как единое целое.

Первым делом клиентское приложение должно произвести создание авторизационного ключа, который обычно создается при первом запуске и практически никогда не изменяется.

Основной недостаток протокола — в том, что злоумышленник, пассивно перехватывающий сообщения, а затем каким-либо образом заполучивший авторизационный ключ (например, украв устройство) получит возможность расшифровать все перехваченные сообщения *post factum*. Вероятно, это не слишком серьезно (украв устройство, можно получить и всю закешированную на нем информацию, ничего не расшифровывая), однако для преодоления этих проблем можно сделать следующее:

- *Сессионные ключи*, генерируемые по протоколу Диффи-Хелмана, и используемые совместно с авторизационным ключом и ключом сообщения для выбора параметров AES. Для их создания клиент должен первым действием после создания новой сессии отправить серверу специальный RPC-запрос («сгенерировать сессионный ключ»), сервер ответит на него, после чего все последующие сообщения сессии шифруются с учетом и сессионного ключа.
- Защищать ключ, хранимый на клиентском устройством, (текстовым) паролем; этот пароль никогда не хранится в памяти и вводится пользователем при запуске приложения или чаще (в зависимости от настроек приложения).
- Данные, хранимые (кэшируемые) на пользовательском устройстве, можно также защищать, шифруя с помощью авторизационного ключа, который, в свою очередь, надо защитить паролем. Тогда без ввода пароля невозможно будет получить доступ даже к этим данным.

Синхронизация времени

Если время на клиенте сильно отличается от времени на сервере, может так получиться, что сервер начнет игнорировать сообщения клиента, или наоборот, из-за некорректного значения идентификатора сообщения (которое тесно связано с временем создания). В таких ситуациях сервер шлет клиенту специальное сообщение с правильным временем, содержащие, помимо него, некую 128-битную соль (либо явно присланную клиентом в специальном RPC-запросе синхронизации, либо равную

ключу последнего сообщения, полученного от клиента в рамках данной сессии). Такое сообщение может быть первым в контейнере, содержащим и другие сообщения (если рассинхронизация существенна, но еще не приводит к игнорированию клиентских сообщений).

При получении такого сообщения (или содержащего его контейнера) клиент сначала выполняет синхронизацию времени (фактически всего лишь запоминает разницу своего и серверного времени, чтобы уметь впредь вычислять «правильное» время), а затем проверяет идентификаторы сообщений на корректность.

В запущенных случаях клиенту придется сгенерировать новую сессию, чтобы обеспечить монотонность идентификаторов сообщений.

Транспорт

Позволяет доставлять уже зашифрованные контейнеры вместе с внешним заголовком (в дальнейшем — полезную нагрузку) от клиента к серверу и наоборот. Есть три типа транспорта:

- HTTP
- TCP
- UDP

Рассмотрим первые два типа.

Передача по HTTP

Реализуется поверх HTTP/1.1 (с keepalive), запущенного поверх классического TCP-порта 80. HTTPS не используется; используется криптографическая схема, объясненная выше.

HTTP-соединение привязывается к сессии (вернее, сессии + идентификатору ключа), указанной в последнем пришедшем пользовательском запросе; обычно во всех запросах сессия одинакова, однако хитрые HTTP-прокси могут это испортить. Сервер может вернуть сообщение в HTTP-соединение только в том случае, если оно принадлежит той же сессии, и если сейчас очередь сервера (был получен HTTP-запрос от клиента, на который еще не было отправлено ответа).

Общая схема такова. Клиент открывает одно или несколько keepalive HTTP-соединений к серверу. При необходимости отправки одного или нескольких сообщений из них составляется полезная нагрузка, после чего делается POST-запрос на URL /api, в качестве данных которому и передается полезная нагрузка. Кроме того, допускаются HTTP-заголовки Content-Length, Keepalive, Host.

После получения запроса сервер может либо подождать немного (если запрос подразумевает ответ после небольшого ожидания), либо сразу вернуть фиктивный ответ (сообщающий всего лишь о том, что контейнер был получен). В любом случае в ответе может оказаться сколько угодно сообщений — сервер вправе заодно отправить любые накопившиеся у него сообщения для этой сессии.

Кроме того, есть специальный longpoll RPC-запрос (действительный только для http-соединений), в котором передается максимальное время ожидания T. Если у сервера есть сообщения для этой сессии, они возвращаются сразу же; в противном случае происходит ожидание до тех пор, пока у сервера не появится сообщение для клиента, либо не пройдет T секунд. Если за T секунд не произошло никаких событий, возвращается фиктивный ответ (специальное сообщение).

Если серверу надо отправить сообщение клиенту, он проверяет, нет ли HTTP-соединения, принадлежащего нужной сессии, и находящегося в состоянии «выполнения HTTP-запроса» (включая long poll), после чего сообщение добавляется в контейнер ответа этого соединения и отправляется пользователю. В типичном случае происходит небольшое дополнительное ожидание (50 миллисекунд), на тот случай, если у сервера вскоре появятся еще сообщения для этой сессии.

Если ни одного подходящего HTTP-соединения нет, сообщения ставятся в очередь отправки для данной сессии. Впрочем, они туда попадают в любом случае, пока явно или косвенно не подтверждено получение клиентом. Для http-протокола неявным подтверждением считается отправка следующего запроса по тому же HTTP-соединению (уже нет — и для HTTP-протокола необходимо слать явные подтверждения); в остальных случаях клиент должен прислать явное подтверждение за разумное время (его можно добавить в контейнер для следующего запроса).

Важно: если подтверждение вовремя не пришло, сообщение может быть перепослано (возможно, в составе другого контейнера). Стороны должны быть морально готовы

к этому и хранить идентификаторы последних полученных сообщений (и игнорировать такие дубли, а не повторять действие). Для того, чтобы не хранить идентификаторы вечно, есть специальные сообщения сборки мусора, эксплуатирующие монотонность идентификаторов сообщений.

Если очередь отправки переполняется, или сообщения ждут в ней больше 10 минут, то сервер их забывает (или отправляет в *своп* - дурное дело нехитрое). Это может случиться и быстрее, если у сервера заканчиваются буферы (например, из-за серьезных проблем в сети, приведших к разрыву большого количества соединений).

TCP-транспорт

Очень похож на HTTP-транспорт, может быть реализован тоже на порт 80 (чтобы проходить все фаерволы) и даже на те же ip-адреса серверов. В этом случае сервер понимает, нужно ли использовать HTTP или TCP-протокол для данного соединения по первым четырем пришедшим байтам (для HTTP это будет POST).

При создании TCP-соединения оно приписывается сессии (и авторизационному ключу), переданному в первом сообщении пользователя, и потом используется исключительно для данной сессии (схемы мультиплексирования не допускаются).

При необходимости отправки полезной нагрузки (пакета) от сервера к клиенту или от клиента к серверу она инкапсулируется следующим образом: спереди дописывается 4 байта длины (включая длину, порядковый номер и CRC32; всегда делится на четыре) и 4 байта с порядковым номером пакета внутри данного tcp-соединения (первый отправленный пакет помечается 0, следующий — 1 и т.д.), а в конце — 4 байта CRC32 (длины, порядкового номера и полезной нагрузки вместе).

Существует сокращённая версия этого протокола: если клиент отправляет первым байтом (важно: только перед самым первым пакетом данных) 0xEF, то после этого длина пакета кодируется одним байтом (0×01..0×7E = длина данных, делённая на 4; либо 0×7F, а затем 3 байта длины (little-endian), делённой на 4), а далее идут сами данные (порядковый номер или CRC32 не добавляются). Ответы сервера в этом случае имеют тот же вид (при этом сервер не отсылает первый байт 0xEF).

В случае, если требуется выравнивание 4-байтовых данных, может быть использована промежуточная версия оригинального протокола: если клиент отправляет 0xEEEEEEEE как первый инт (int) (четыре байта), то длина пакета зашифрована всегда четырьмя байтами как в оригинальной версии, но порядковый номер и CRC32 опускаются, таким образом уменьшая итоговый /общий размер пакета на 8 байт.

В полной и в сокращённой версии протокола есть поддержка быстрых подтверждений. В этом случае клиент устанавливает старший бит длины в пакете с запросом, а сервер отсылает в ответ специальные 4 байта, представляющие собой самостоятельный пакет. Они представляют собой старшие 32 бита SHA1 от зашифрованной части пакета, с установленным старшим битом, чтобы было понятно, что это не длина обычного пакета с ответом сервера; если используется сокращённая версия, то к этим четырём байтам применяется bswap.

Неявных подтверждений для TCP-транспорта не бывает: все сообщения должны быть явно подтверждены. Чаще всего подтверждения помещаются в контейнер вместе со следующим запросом или ответом, если он отправляется вскоре. Например, это почти всегда так для сообщений от клиента, содержащих RPC-запросы: подтверждение обычно приходит вместе с RPC-ответом.

В случае возникновения ошибки сервер может прислать пакет, полезная нагрузка которого состоит из 4 байтов — кода ошибки. Например, код ошибки −403 соответствует ситуациям, в которых через HTTP-протокол вернулась бы соответствующая HTTP-ошибка.

Описание программной реализации MTProto

Ссылка на продукт: https://github.com/KomogorovKirill/MTProto.git

Работа с потоками осуществляется при помощи класса std::thread
Работа с большими числами осуществляется при помощи библиотеки gmp
https://gmplib.org/

При помощи gmp реализован протокол Диффи-Хеллмана (модуль keyExchange.cpp) и функция getDigit (модуль digits.cpp).

Работа с шифрованием осуществляется при помощи библиотеки cryptopp

https://www.cryptopp.com/

При помощи cryptopp реализованы:

1. RAW-RSA (модуль rsa.cpp)
    1.1. https://www.cryptopp.com/wiki/Raw_RSA
    1.2. https://www.cryptopp.com/wiki/Keys_and_Formats
2. AES256 (модуль aes.cpp)
    2.1. https://cryptopp.com/wiki/Advanced_Encryption_Standard
3. Base64Encoder (модуль aes.cpp)
    3.1. https://www.cryptopp.com/wiki/Base64Encoder
4. Base64Decoder (модуль aes.cpp)
    4.1. https://www.cryptopp.com/wiki/Base64Decoder
5. SHA256 (модуль sha256.cpp)
    5.1. https://www.cryptopp.com/wiki/SHA2

Кодировка base64 используется только в модуле aes.cpp

Особенности

Для успешного запуска клиент - сервера необходимо сгенерировать ключи сервера командой *./server keygen* и публичный ключ скопировать в директорию со компилированной версией клиента.

модуль **server.cpp**

int **main**(int argc, char** argv)

- Функция реализует работу с сокетами, инициализацию базы данных, создание нового потока.
- Инициализация базы данных -создается база данных, содержащая таблицу:

SOCK_ID	CHAR(64)	NOT NULL
SESSION_ID	CHAR(64)	NOT NULL
AUTH_KEY	CHAR(2048)	NOT NULL

Где sock_id - постоянный id клиента (дескриптор файла для сокета используется как id клиента на текущую сессию),

session_id – id, необходимый для работы протокола mtproto,

auth_key - уникальный ключ, использующийся для работы протокола mtproto,

void **sendMsg**(int* recipient_socket, int sender_socket)

- Реализует протокол mtproto на стороне сервера. Принимает два параметра - массив sockets (содержащий sock_id всех клиентов) и sock_id отправителя. В рамках этой функции sockets выступает в качестве массива с sock_id получателей.

Клиент и сервер обмениваются структурой package, в которой объявлены поля:

- sender_session_id - сессионный идентификатор клиента-отправителя, генерируется в ходе раунда обмена ключей (модуль keyExhange)
- recipient_session_id - сессионный идентификатор клиента-получателя (не используется)
- msg_len - длина сообщения
- msg_key - уникальный идентификатор сообщения
- encrypted_data - зашифрованная информация AES256(salt+session_id+msg+padding)

db_decryption_aes_key – ключ для дешифрования данных из базы данных

db_decryption_aes_iv - инициализационный вектор для дешифрования данных из базы данных

1. **Прием пакета от клиента-отправителя**
    1.1. Принимается информация от клиента-отправителя. Затем сервер, зная sock_id клиента-отправителя, получает его auth_key из базы данных.
    1.2. Сервер расшифровывает auth_key из базы данных при помощи db_decryption_aes_key и db_decryption_aes_iv
    1.3. Сервер получает aes_key и aes_iv, используя auth_key и msg_key клиента-отправителя, расшифровывает encrypted_data и получает decrypted_data.
2. **Далее необходимо зашифровать decrypted_data для отправки клиентам-получателям**
    2.1. Сервер, зная sock_id клиента-получателя, получает его auth_key из базы данных.
    2.2. Сервер расшифровывает auth_key из базы данных при помощи db_decryption_aes_key и db_decryption_aes_iv
    2.3. Сервер получает aes_key и aes_iv, используя auth_key клиента-получателя и msg_key клиента-отправителя, зашифровывает decrypted_data и отправляет клиенту-получателю.
    2.4. Повтор шагов 2.1-2.3 для всех активных sock_id

модуль **client.cpp**

int main(int argc, char** argv)

- Реализует работу с сокетами.
- RSA ключи клиента генерируются заного при запуске программы. В раунде обмена ключами публичный ключ клиента отправляется на сервер.
- Инициализация базы данных -создается база данных, содержащая таблицу:

SESSION_ID	CHAR(64)	NOT NULL
AUTH_KEY	CHAR(2048)	NOT NULL

- При подключении нового клиента создает для него два потока обработки сообщений - приём и отправка

void **sendMsg**(int sockfd)

- Отправляет зашифрованное сообщение серверу

Структура package идентична структуре package на сервере.

Ход работы:

1. Получение auth_key и session_id из базы данных
2. Формирование блока to_be_encrypted (salt+msg+padding)
3. Формирование aes_key и aes_iv и шифрование блока to_be_encrypted AES256Encode(to_be_encrypted, aes_key, aes_iv)
4. Отправка на сервер

void **getMsg**(int sockfd)

- Принимает зашифрованное сообщение от сервера

Структура package идентична структуре package на сервере.

Ход работы:

1. Получение auth_key и session_id из базы данных
2. Формирование aes_key и aes_iv и дешифрование decrypted_data
3. Вывод сообщения в консоль

модуль **keyExchange.cpp**

- Генерация auth_key посредством протокола Диффи-Хеллмана

short **getNewSession_client**(int sockfd) - Обмен ключами, сторона сервера

short **getNewSession_server**(int sockfd) - Обмен ключами, сторона клиента

- Сервер обменивается с клиентом структурой package

Структура package клиента аналогична структуре package сервера:

- session_id - сессионный идентификатор нового клиента
- dh_aes_key – зашифрованный при помощи открытого ключа сервера/клиента AES ключ
- dh_aes_iv - зашифрованный при помощи открытого ключа сервера/клиента AES IV

- p– 2048 битное число P, генерируется на сервере (см. digits.cpp)
- g- 64 битное число g, генерируется на сервере (см. digits.cpp)
- A – g^amod p для сервера или g^bmod p для клиента

Сторона сервера:

1. Генерируется session_id и присваивается новому клиенту. Этот параметр нужен для составления блока tobeencrypted в дальнейшем
2. Сервер принимает открытый RSA-ключ клиента
3. Сервер генерирует числа p, g, a(64 бит) и высчитывает число A = g^amod p
4. Сервер шифрует число A при помощи ключа dh_aes_key и dh_aes_iv, которые шифруются при помоощи открытого ключа клиента. Затем отправляет package клиенту

Сторона клиента:

1. Клиент генерирует 64 битное число b и принимает package от сервера. Затем расшифровывает число A при помощи dh_aes_key, dh_aes_iv, которые в свою очередь расшифровываются при помощи закрытого ключа клиента
2. Клиент высчитывает число B = g^bmod p и шифрует его при помощи ключа dh_aes_key и dh_aes_iv, затем отправляет package на сервер. AES параметры шифруются при помощи открытого ключа сервера.
3. При помощи A высчитывает auth_key (A^bmod p).

Сторона сервера:

1. Сервер принимает package и расшифровывает dh_aes_key, dh_aes_iv при помощи закрытого ключа сервера.
2. При помощи dh_aes_key, dh_aes_iv расшифровывает число B и высчитывает auth_key (B^amod p).
3. Шифрует auth_key при помощи db_aes_key и db_aes_iv
4. Сохраняет в базу данных sock_id, session_id, auth_key

модуль **msg_encr_decr.cpp**

string **getEncryptedBlock**(string session_id, string msg)

- Принимает два параметра: session_id и msg
- Возвращает строку to_be_encrypted длиной 1024 байта

to_be_encrypted = salt + session_id + msg + padding, где padding нужен для дополнения до 1024 байт

string **get_msg_key**(string plaintext, string auth_key) - возвращает msg_key

string **get_aes_key**(string msg_key, string auth_key) - возвращает aes_key

string **get_aes_iv**(string msg_key, string auth_key) - возвращает aes_iv

msg_key, aes_key, aes_iv высчитываются по специальным формулам:

- msg_key_large = SHA256 (substr (auth_key, 88+x, 32) + plaintext + random_padding);
- msg_key = substr (msg_key_large, 8, 16);
- sha256_a = SHA256 (msg_key + substr (auth_key, x, 36));
- sha256_b = SHA256 (substr (auth_key, 40+x, 36) + msg_key);
- aes_key = substr (sha256_a, 0, 8) + substr (sha256_b, 8, 16) + substr (sha256_a, 24, 8);
- aes_iv = substr (sha256_b, 0, 8) + substr (sha256_a, 8, 16) + substr (sha256_b, 24, 8);

модуль **digits.cpp**

char* **getDigit**(char* digit, int16_t bits, int16_t mode, int16_t base)

Выработка чисел по заданным параметрам:

- digit - сюда записывается результат
- bits - количество битов
- mode - если 1, то вырабатывать простое, если 0, то вырабатывать составное
- base - если 10, вернуть число в DEC, если 16, вернуть число в HEX