**-Доп. теория-**

<https://ru.wikipedia.org/wiki/Прокси-сервер>

<https://proxys.io/ru/blog/proksi-info/otlichie-protokola-http-ot-socks5>

**SOCKS5**

SOCKS — сетевой протокол сеансового уровня модели OSI, который позволяет пересылать пакеты от клиента к серверу через прокси-сервер прозрачно (незаметно для них) и таким образом использовать сервисы за межсетевыми экранами (файрволами). Более поздняя версия SOCKS5 предполагает аутентификацию, так что только авторизованные пользователи получают доступ к серверу.

Сокс-сервер просто пробрасывает через себя TCP и UDP трафик. С его помощью можно решать самые разные задачи: организовывать защищенный доступ к службам, расположенным за межсетевым экраном (firewall), скрывать свой истинный IP-адрес во время работы с недружелюбными сетевыми ресурсами или реализовать универсальный прокси-сервер, поддерживающий любые протоколы прикладного уровня (HTTP, FTP, POP3/SMTP, ICQ и т.д.). Основная задача данного протокола – внедрить в «нормальный» процесс обмена данными некоего посредника, называемого SOCKS-сервером или SOCKS-прокси. Когда клиент (поддерживающее SOCKS приложение) желает отправить какую-либо информацию по сети, он устанавливает соединение не с реальным адресатом, а с SOCKS-сервером, который, в свою очередь, пересылает данные по назначению, но уже от своего имени. С точки зрения «настоящего» сервера (например, web-узла, который пользователь желает) SOCKS-прокси является самым обыкновенным клиентом. Таким образом, сущность (IP-адрес) истинного клиента оказывается скрытой от обслуживающего его сервера. Это весьма удобное обстоятельство таит в себе потенциальную опасность (можете спрятаться вы, но ведь могут и от вас), поэтому реально существующие SOCKS-сервера имеют развитые схемы контроля доступа (запрет входящих и исходящих соединений по заданному перечню адресов) и поддерживают авторизацию пользователей по паролю.

Отметим, что коль скоро SOCKS работает на более низком по сравнению с прикладным (а именно, транспортном) уровне модели OSI, его поддержка не потребует никаких изменений в логике работы клиента, а тем более – сервера. Действительно, все что нужно – это модифицировать реализацию функций, отвечающих за создание сетевого подключения и отправку данных: connect(), bind(), send() и т.п. На практике это обычно достигается перехватом системных вызовов с их последующей подменой поддерживающими SOCKS пользовательскими аналогами. Никаких правок в исходном коде клиентских приложений, а тем более самого доступа к исходным текстам, как правило, не требуется. Эта мощная процедура известна как «соксификация» и будет подробно рассмотрена ниже.

Теперь, когда мы получили общее представление о SOCKS, можно перейти к более детальному рассмотрению данного протокола.

Спецификация SOCKS5

Как уже отмечалось ранее, SOCKS5 является протоколом транспортного уровня. Его «соседи» – TCP и UDP непосредственно используются для передачи данных, поступающих с прикладного уровня (от пользовательских приложений), а значит, SOCKS-прокси должен уметь корректно работать с каждым из них. Отметим также, что протокол ICMP, используемый утилитами ping и traceroute, находится ниже транспортного уровня, а потому соксификации, к сожалению, не поддается.

Прежде чем отправлять какие-либо данные, клиент должен пройти процедуру авторизации на SOCKS-сервере. Для этого он открывает TCP-соединение с портом 1080 (значение по умолчанию) SOCKS-сервера и отправляет по нему сообщение, содержащее кодовые номера поддерживаемых им методов аутентификации. SOCKS-сервер выбирает один из методов по своему усмотрению и сообщает его номер клиенту. Аутентификация может отсутствовать (на практике это скорее всего означает, что SOCKS-сервер различает клиентов по их IP-адресам) или производиться на основании имени пользователя и пароля. В последнем случае возможно большое количество различных вариантов, от тривиального «Username/Password Authentication» предусматривающего передачу пароля в открытом виде до куда более безопасного CHAP (зашифрованный пароль, открытые данные) и GSSAPI, которое может использоваться для полной криптографической защиты трафика. После успешной авторизации клиент получает возможность посылать запросы (команды), устанавливать исходящие соединения и даже принимать входящие.

Установка исходящего TCP-соединения.

Для установки исходящего TCP-соединения, клиент отправляет SOCKS-серверу запрос «CONNECT», в котором указывает адрес и порт доставки. Для идентификации узла-получателя могут использоваться как IP-адреса (поддерживаются IPv4/IPv6), так и полноценные доменные имена. В последнем случае SOCKS-сервер берет на себя заботу по их разрешению, так что сеть, в которой работает клиент, в принципе может обходиться и без DNS-сервера. В ответном сообщении SOCKS-сервер сообщает код ошибки (как обычно, 0 обозначает, что операция прошла успешно), а также IP-адрес (BND.ADDR) и TCP-порт (BND.PORT), которые будут использоваться для фактической связи с запрошенным узлом.

Поскольку SOCKS-сервера, как правило, имеют более одного сетевого интерфейса, данный IP-адрес может отличаться от того, с которым было установлено управляющее соединение. После этого клиент открывает новый TCP-сеанс с BND.ADDR:BND.PORT и осуществляет отправку данных. Исходящее TCP-соединение разрывается одновременно с закрытием управляющей сессии. Заметим, что запрос «CONNECT» может быть отклонен SOCKS-прокси, если адреса источника (клиента) или получателя (сервера) запрещены.

Установка исходящего UDP-соединения

В отличие от потокового протокола TCP, подразумевающего установку сеанса, протокол UDP является датаграммным, а потому несколько более сложны в обращении. Его поддержка появилась лишь в SOCKS5.

Перед отправкой UDP-датаграмм клиент запрашивает у SOCKS-сервера UDP-ассоциацию, используя для этого команду «UDP ASSOCIATE». UDP-ассоциация – это своего рода виртуальный сеанс между клиентом и SOCKS-сервером. В исходящем запросе клиент указывает предполагаемые адрес и порт, которые будут выступать в качестве источника будущих UDP-датаграмм. Если на момент установки UDP-ассоциации эта информация еще не известна, клиент должен использовать комбинацию 0.0.0.0:0 (или, скажем, x.x.x.x:0, если неизвестен только номер порта). В ответном сообщении SOCKS-сервер указывает IP-адрес (BND.ADDR) и UDP-порт (BND.PORT), на которые следует направлять исходящие датаграммы. При этом адрес и порт их реального получателя указываются прямо в теле (можно сказать, что имеет место UDP-инкапсуляция). Эти параметры, наряду с адресом и портом отправителя используются для принятия решения о допустимости отправки датаграммы. Это создает дополнительную нагрузку на SOCKS-сервер: правила фильтрации необходимо применять к каждой UDP-датаграмме, тогда как в случае TCP-соединения его легитимность оценивается один раз, в момент исполнения SOCKS-сервером команды «CONNECT». Согласно требованиям стандарта, SOCKS-сервер должен следить за тем, чтобы IP-адрес отправителя датаграммы совпадал с адресом узла, создавшего UDP-ассоциацию.

UDP-ассоциация разрушается одновременно с закрытием управляющего TCP-сеанса, в рамках которого была послана команда «UDP ASSOCIATE». Многие из существующих SOCKS-серверов испытывают серьезные проблемы, если между ними и клиентом, запросившим UDP-ассоциацию, располагается межсетевой экран с функцией NAT. Причина этого кроется в изменении адреса и порта отправителя, которое происходит в тот момент, когда UDP-датаграмма пересекает межсетевой экран. Как следствие, сервер и ничего не подозревающее клиентское приложение начинают говорить на разных языках: предполагаемый адрес и порт источника, указанные в команде «UDP ASSOCIATE» перестают соответствовать реальным параметрам получаемых SOCKS-сервером датаграмм. В результате они оказываются отброшенными как не принадлежащие UDP-ассоциации. Проблему можно было бы решить, указав в качестве предполагаемого источника 0.0.0.0:0, что должно интерпретироваться SOCKS- сервером как «любая UDP-датаграмма, пришедшая с того же адреса, что и команда на создание ассоциации». К сожалению, большинство из реально существующих SOCKS-серверов трактуют стандарт более узко и не позволяют одновременно установить в ноль и предполагаемый адрес, и порт отправителя.

Прием входящих соединений

Эта достаточно оригинальная возможность может оказаться полезной в случаях, когда клиент и «настоящий» сервер в описанной выше схеме меняются местами, что может произойти, например, в протоколах типа FTP. В целях дальнейшего рассмотрения будем предполагать, что между “клиентом” (стороной, собирающейся принять входящее соединение) и “сервером” (стороной, инициирующей входящее соединение) уже установлен “прямой” канал связи при помощи команды “CONNECT”. Для открытия “обратного” канала “клиент” должен послать SOCKS-серверу команду “BIND”, указав в ее параметрах IP-адрес и порт, которые будут использоваться им для приема входящего соединения. В ответ на это SOCKS-сервер сообщает IP-адрес и порт, выделенные им для поддержания “обратного” канала. Предполагается, что «клиент» передаст эти параметры «серверу», используя средства, предоставляемые протоколами прикладного уровня (например, команду «PORT» протокола FTP). После того, как SOCKS-сервер примет (или отбросит) входящее соединение, он повторно уведомляет об этом «клиента», сообщая ему IP-адрес и порт, используемые “сервером”. Отметим, что прием входящих соединений может осуществлять лишь приложение, разработчики которого позаботились о поддержке SOCKS ещё на этапе проектирования. В противном случае (если приложение работает с SOCKS-сервером через программу-соксификатор), оно не сможет предоставить корректную информацию об адресе ожидающего “обратной связи” сокета (т.е. сформирует неверную команду “PORT” в рассмотренном выше примере с FTP).

**Цепочки proxy (как реализовать для socks5 цепочку)**

Архитектура протокола SOCKS5 позволяет легко объединять SOCKS-сервера в каскады, или, как их еще называют «цепочки» («chains»). Примечательно, что все необходимые для этого действия могут быть произведены на стороне клиента. К «звеньям» цепочки предъявляется единственное требование: они должны «доверять» друг другу (т.е. допускать установку входящих и исходящих соединений). Если образующие каскад SOCKS-сервера не являются анонимными (т.е. используют схемы аутентификации Username/Password, CHAP или подобные), необходимо также, чтобы пользователь мог успешно пройти процедуру авторизации на каждом из них.

Предположим, что у нас имеется набор из N SOCKS-серверов с именами socks1, socks2, ...,socksN, удовлетворяющих всем вышеперечисленным требованиям. Тогда для создания каскада клиент может поступить следующим образом:

В случае исходящего TCP-соединения: клиент подключается к socks1, проходит (при необходимости) процедуру авторизации и посылает команду «CONNECT», указав в качестве адреса доставки socks2. Выполняя этот запрос, socks1 создаст новое соединение с socks2 и будет исправно передавать всю идущую по нему информацию клиенту, при этом socks2 не будет даже догадываться, с кем он общается на самом деле. Далее процедура повторяется до тех пор, пока не будет установлено соединение между socks(N-1) и socksN. Последний сервер каскада подключается к непосредственно узлу, который интересует клиента. Передача данных происходит в обычном режиме: клиент отправляет пакет на сервер socks1, который, в свою очередь, передает его socks2, ... и так до тех пор, пока не будет достигнут конечный узел.

В случае исходящего UDP-соединения: клиент подключается к socks1, проходит процедуру авторизации и последовательно посылает две команды: “CONNECT” (адрес доставки – socks2) и “UDP ASSOCIATE”. Таким образом, создаются два новых соединения: виртуальный UDP-канал между клиентом и socks1, а также TCP-сессия между socks1 и socks2. Используя эту TCP-сессию, клиент (от имени socks1) посылает команду “UDP ASSOCIATE” на сервер socks2 (открывает UDP-канал между socks1 и socks2) и “CONNECT” на сервер socks3. Процедура продолжается до тех пор, пока между всеми SOCKS-серверами каскада не будут установлены виртуальные UDPканалы. Чтобы отослать какие-либо данные, клиент предварительно производит N-кратную инкапсуляцию UDP-датаграммы, указывая в качестве адреса доставки последовательно: socks1, socks2, socks3, socksN и адрес реального получателя, а затем отправляет ее на сервер socks1. Отметим, что на практике данный вариант каскадирования встречается крайне редко. Это связано с тем, что SOCKS-сервера, как и NAT Firewall'ы, могут изменить порт источника датаграммы, что приведет к проблемам, подробно описанным в разделе “Установка исходящего UDP-соединения”.

**HTTP Proxy**

HTTP прокси реализуется в соответствии c HTTP протоколом и позволяет клиентам получать через себя доступ к веб серверам, таким образом, клиент по HTTP протоколу отправляет прокси-серверу те же HTTP запросы, какие бы он посылал на требуемый сервер.

Задача прокси-сервера транслировать эти запросы на запрашиваемый сервер и осуществлять транспорт HTTP-трафика через себя клиенту, при этом прокси сервер может изменять параметры HTTP-запросов клиентов и ответа сервера и транслировать клиенту модифицированные объекты и заголовки ответа.

Задача кэширующего прокси сервера сохранять проходящие через него страницы и объекты, и если клиенты запросили объект, который уже был сохранен на сервере, то сразу отправлять этот объект клиенту не скачивая его с веб сервера в случае если этот объект на веб сервере не менялся, при этом если объект на веб сервере обновился, информацию об этом необходимо получать с помощью заголовков if-modified-since, if-none-match.

В языках программирования уже есть готовые компоненты для работы с HTTP сервером, которые позволяет переопределять HTTP обработчик запросов и получать доступ к передаваемым данным, полям запроса, анализировать их и делать новый запрос к серверу в качестве HTTP клиента.

**If-Modified-Since**

Заголовок HTTP запроса If-Modified-Since делает запрос условным: сервер отправит обратно запрошенный ресурс с статусом 200 (ОК), только если он был изменён после указанной даты. Если запрос не был изменён после указанной даты, ответ будет 304 (не изменён) без какого-либо тела; заголовок Last-Modified при этом будет содержать дату последней модификации. If-Modified-Since может использоваться только с GET или HEAD.

**Last-Modified**

Заголовок Last-Modified в ответе HTTP содержит дату и время, в которую, по мнению удалённого сервера, запрашиваемый ресурс был изменён. Он используется в качестве средства проверки для определения того, остался ли ресурс неизменным. Этот заголовок менее надёжный, чем ETag, и используется как резервный механизм. Условный запрос, содержащий заголовок If-Modified-Since или If-Unmodified-Since позволяет серверу использовать для сравнения эту дату.

**If-None-Match**

Заголовок HTTP-запроса If-None-Match делает запрос условным. Для методов GET и HEAD сервер вернёт запрошенный ресурс со статусом 200 (ОК), только если у него нет ETag, соответствующего заданным. Для других методов запрос будет обработан только в том случае, если ETag существующего в конечном итоге ресурса не соответствует ни одному из перечисленных значений.

Когда условие не выполняется для методов GET и HEAD, сервер должен вернуть код состояния HTTP 304 (не изменён). Для методов, которые применяют изменения на стороне сервера, используется код состояния 412 (необходимое условие не выполнено). Обратите внимание, что сервер, генерирующий ответ 304, ДОЛЖЕН сгенерировать любое из следующих полей заголовка, которые были бы отправлены в ответе 200 (OK) на тот же запрос: Cache-Control, Content-Location, Date, ETag, Expires и Vary.

Сравнение с сохранённым ETag использует алгоритм слабого сравнения, что означает, что два файла считаются идентичными, если содержимое эквивалентно — они не обязательно должны быть идентичными байт за байтом. Например, две страницы, которые отличаются датой создания в нижнем колонтитуле, всё равно будут считаться идентичными.

При использовании в сочетании с If-Modified-Since, If-None-Match имеет приоритет (если сервер его поддерживает).

**ETag**

Заголовок HTTP ответа ETag является идентификатором специфической версии ресурса. Он позволяет более эффективно использовать кеш и сохраняет пропускную способность, позволяя серверу отправлять не весь ответ, если содержимое не изменилось. С другой стороны, если контент все-таки поменялся, ETag помогает предотвратить одновременное обновление ресурса от перезаписи друг друга.

Если ресурс по заданному URL изменился, будет сгенерировано новое значение ETag. Поэтому ETag чем-то похож на отпечаток и может также быть использован для отслеживания предназначения некоторых серверов. Сравнение этих заголовков позволяет быстро определить являются ли два представления ресурса одними и теме же. Отслеживаемый сервер также может задать сохранять их постоянно.

**HTTP CONNECT Proxy**

HTTP-туннелирование используется для создания сетевого соединения между двумя компьютерами в условиях ограниченного сетевого подключения, включая брандмауэры, NAT и ACL, среди других ограничений. Туннель создается посредником, называемым прокси-сервером, который обычно находится в демилитаризованной зоне.

Туннелирование также может позволить осуществлять связь с использованием протокола, который обычно не поддерживается в сети с ограниченным доступом.

Наиболее распространенной формой HTTP-туннелирования является стандартизированный метод HTTP CONNECT. В этом механизме клиент запрашивает HTTP-прокси-сервер для переадресации TCP-соединения в желаемый пункт назначения. Затем сервер переходит к установлению соединения от имени клиента. Как только соединение установлено сервером, прокси-сервер продолжает передавать TCP-поток клиенту и обратно. Только начальный запрос на подключение является HTTP - после этого сервер просто проксирует установленное TCP-соединение.

Этот механизм позволяет клиенту, использующему HTTP-прокси, получать доступ к веб-сайтам с использованием SSL или TLS (т.е. HTTPS). Прокси-серверы также могут ограничивать соединения, разрешая подключения только к порту HTTPS по умолчанию 443, внося хосты в белый список или блокируя трафик, который, по-видимому, не является SSL.