Памятка пользователям ssh

Опубликовано: 04.08.2014 | Автор: Николай

В статье описаны продвинутые функций OpenSSH, которые позволяют сильно упростить жизнь системным администраторам и программистам, которые не боятся шелла. В отличие от большинства руководств, которые кроме ключей и -L/D/R опций ничего не описывают, я попытался собрать все интересные фичи и удобства, которые с собой несёт ssh.Предупреждение: пост *очень*объёмный, но для удобства использования я решил не резать его на части.Оглавление:

- управление ключами
- копирование файлов через ssh
- Проброс потоков ввода/вывода
- Монтирование удалённой FS через ssh
- Удалённое исполнение кода
- Алиасы и опции для подключений в .ssh/config
- Опции по-умолчанию
- Проброс X-сервера
- ssh в качестве socks-proxy
- Проброс портов — прямой и обратный
- Реверс-сокс-прокси
- туннелирование L2/L3 трафика
- Проброс агента авторизации
- Туннелирование ssh через ssh сквозь недоверенный сервер (**с большой вероятностью вы этого не знаете**)

Настройки `sshd` находятся в файле `/etc/ssh/sshd_config`. Открываем этот файл для редактирования и изменяем его содержимое.

Следующая группа параметров относится к аутентификации. Первый параметр означает, что соединение будет разорвано через указанное количество секунд, если пользователь не войдёт в систему. Уменьшим это время в два раза (по умолчанию 120 сек).

LoginGraceTime 60

Второй параметр разрешает или запрещает вход по SSH под суперпользователем. Запрещаем вход суперпользователю.

PermitRootLogin no

Третий параметр включает проверку сервисом ssh прав и владение домашним каталогом пользователя, который пытается получить удалённый доступ к компьютеру. Оставляем yes.

StrictModes yes

Добавляем параметр AllowUsers, которого нет в конфигурационном файле по умолчанию. Этот параметр разрешает доступ к серверу по протоколу SSH только для перечисленных пользователей (через пробел).

AllowUsers admin andrew

В нашем примере разрешение есть только у пользователей admin и andrew. Значениями этого параметра могут выступать имена пользователей, отделённые друг от друга пробелами. Нельзя использовать в качестве значений UID (User ID). Можно использовать запись пользователя в виде user@host, например andrew@sup.etr.ru. Это означает, что доступ разрешён пользователю andrew с компьютера sup.etr.ru. Причём пользователь и компьютер проверяются отдельно, это позволяет

разграничить доступ определенных пользователей с определенных компьютеров. Существует обратный параметр – DenyUsers, который запрещает доступ пользователям, перечисленным в значении этого параметра. Кроме параметров связанных с доступом отдельных пользователей существуют соответствующие параметры для групп: AllowGroups и DenyGroups.

Лучше добавим ещё один полезный параметр, который отсутствует в файле sshd_config – DebianBanner. Этот параметр добавляет в строку ответа sshd информацию об операционной системе, при обращению к серверу по протоколу TELNET или при сканировании nmap. Эта строка может выглядеть так: SSH-2.0-OpenSSH_5.5p1 Debian-6+squeeze1. Скроем информацию о нашей операционной системе.

DebianBanner no

Теперь строка ответа будет выглядеть так: SSH-2.0-OpenSSH_5.5p1

Управление ключами

Теория в нескольких словах: ssh может авторизоваться не по паролю, а по ключу. Ключ состоит из открытой и закрытой части. Открытая кладётся в домашний каталог пользователя, «которым» заходят на сервер, закрытая — в домашний каталог пользователя, который идёт на удалённый сервер. Половинки сравниваются (я утрирую) и если всё ок — пускают. Важно: авторизуется не только клиент на сервере, но и сервер по отношению к клиенту (то есть у сервера есть свой собственный ключ). Главной особенностью ключа по сравнению с паролем является то, что его нельзя «украсть», взломав сервер — ключ не передаётся с клиента на сервер, а во время авторизации клиент доказывает серверу, что владеет ключом (та самая криптографическая магия).

Генерация ключа

Свой ключ можно сгенерировать с помощью команды ssh-keygen. Если не задать параметры, то он сохранит всё так, как надо.

Ключ можно закрыть паролем. Этот пароль (в обычных графических интерфейсах) спрашивается один раз и сохраняется некоторое время. Если пароль указать пустым, он спрашиваться при использовании не будет. Восстановить забытый пароль невозможно.

Сменить пароль на ключ можно с помощью команды **ssh-keygen -p**.

Структура ключа

(если на вопрос про расположение ответили по-умолчанию).
~/.ssh/id_rsa.pub — открытый ключ. Его копируют на сервера, куда нужно получить доступ.
~/.ssh/id_rsa — закрытый ключ. Его нельзя никому показывать. Если вы в письмо/чат скопипастите его вместо pub, то нужно генерировать новый ключ. (Я не шучу, примерно 10% людей, которых просишь дать ssh-ключ постят id_rsa, причём из этих десяти процентов мужского пола 100%).

Копирование ключа на сервер

В каталоге пользователя, под которым вы хотите зайти, если создать файл **~/.ssh/authorized_keys** и положить туда открытый ключ, то можно будет заходить без пароля. Обратите внимание, права на файл не должны давать возможность писать в этот файл посторонним пользователям, иначе ssh его не примет. В ключе последнее поле — user@machine. Оно не имеет никакого отношения к авторизации и служит только для удобства определения где чей ключ. Заметим, это поле может быть поменяно (или даже удалено) без нарушения структуры ключа.

Если вы знаете пароль пользователя, то процесс можно упростить. Команда **ssh-copy-id user@server** позволяет скопировать ключ не редактируя файлы вручную.

Замечание: Старые руководства по ssh упоминают про authorized_keys2. Причина: была первая версия ssh, потом стала вторая (текущая), для неё сделали свой набор конфигов, всех это очень утомило, и вторая версия уже давным давно переключилась на версии без всяких «2». То есть всегда

authorized_keys и не думать о разных версиях.

Если у вас ssh на нестандартном порту, то ssh-copy-id требует особого ухищрения при работе: `ssh-copy-id '-p 443 user@server'` (внимание на кавычки).

Ключ сервера

Первый раз, когда вы заходите на сервер, ssh вас спрашивает, доверяете ли вы ключу. Если отвечаете нет, соединение закрывается. Если да — ключ сохраняется в файл **~/.ssh/known_hosts**. Узнать, где какой ключ нельзя (ибо несекьюрно).

Если ключ сервера поменялся (например, сервер переустановили), ssh вопит от подделке ключа. Обратите внимание, если сервер не трогали, а ssh вопит, значит вы не на тот сервер ломитесь (например, в сети появился ещё один компьютер с тем же IP, особо этим страдают всякие локальные сети с 192.168.1.1, которых в мире несколько миллионов). Сценарий «злобной man in the middle атаки» маловероятен, чаще просто ошибка с IP, хотя если «всё хорошо», а ключ поменялся — это повод поднять уровень паранойи на пару уровней (а если у вас авторизация по ключу, а сервер вдруг запросил пароль — то паранойю можно включать на 100% и пароль не вводить).

Удалить известный ключ сервера можно командой **ssh-keygen -R server**. При этом нужно удалить ещё и ключ IP (они хранятся раздельно): **ssh-keygen -R 127.0.0.1**.

Ключ сервера хранится в **/etc/ssh/ssh_host_rsa_key** и **/etc/ssh/ssh_host_rsa_key.pub**. Их можно:
а) скопировать со старого сервера на новый.
б) сгенерировать с помощью ssh-keygen. Пароля при этом задавать не надо (т.е. пустой). Ключ с паролем ssh-сервер использовать не сможет.

Заметим, если вы сервера клонируете (например, в виртуалках), то ssh-ключи сервера нужно обязательно перегенерировать.

Старые ключи из know_hosts при этом лучше убрать, иначе ssh будет ругаться на duplicate key.

Копирование файлов

Передача файлов на сервер иногда может утомлять. Помимо возни с sftp и прочими странными вещами, ssh предоставляет нам команду **scp**, которая осуществляет копирование файла через ssh-сессию.

```
scp path/myfile user@8.8.8.8:/full/path/to/new/location/
```

Обратно тоже можно:
```
scp user@8.8.8.8:/full/path/to/file /path/to/put/here
```

Fish warning: Не смотря на то, что mc умеет делать соединение по ssh, копировать большие файлы будет очень мучительно, т.к. fish (модуль mc для работы с ssh как с виртуальной fs) работает очень медленно. 100-200кб — предел, дальше начинается испытание терпения. (Я вспомнил свою очень раннюю молодость, когда не зная про scp, я копировал ~5Гб через fish в mc, заняло это чуть больше 12 часов на FastEthernet).

Возможность копировать здорово. Но хочется так, чтобы «сохранить как» — и сразу на сервер. И чтобы в графическом режиме копировать не из специальной программы, а из любой, привычной.

Так тоже можно:

sshfs

Теория: модуль fuse позволяет «экспортировать» запросы к файловой системе из ядра обратно в userspace к соответствующей программе. Это позволяет легко реализовывать «псевдофайловые системы». Например, мы можем предоставить доступ к удалённой файловой системе через ssh так, что все локальные приложения (за малым исключением) не будут ничего подозревать.

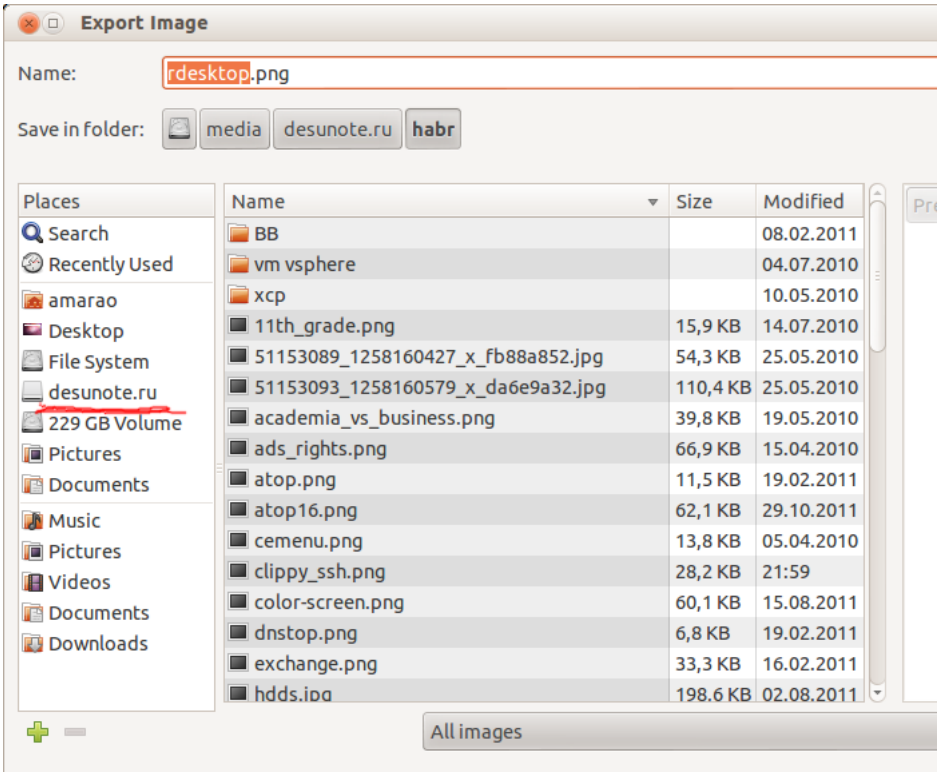
Собственно, исключение: O_DIRECT не поддерживается, увы (это проблема не sshfs, это проблема fuse вообще).

Использование: установить пакет sshfs (сам притащит за собой fuse).

Собственно, пример моего скрипта, который монтирует desunote.ru (размещающийся у меня на домашнем комьютере — с него в этой статье показываются картинки) на мой ноут:

```
#!/bin/bash
sshfs desunote.ru:/var/www/desunote.ru/ /media/desunote.ru -o reconnect
```

Делаем файл +x, вызываем, идём в любое приложение, говорим сохранить и видим:



Параметры sshfs, которые могут оказаться важными: -o reconnect (говорит пытаться пересоединиться вместо ошибок).

Если вы много работаете с данными от рута, то можно (нужно) сделать idmap:

-o idmap=user. Работает она следующим образом: если мы коннектимся как пользователь pupkin@server, а локально работаем как пользователь vasiliy, то мы говорим «считать, что файлы pupkin, это файлы vasiliy». ну или «root», если мы коннектимся как root.

В моём случае idmap не нужен, так как имена пользователей (локальное и удалённое) совпадают.

Заметим, комфортно работать получается только если у нас есть ssh-ключик (см. начало статьи), если нет — авторизация по паролю выбешивает на 2-3 подключение.

Отключить обратно можно командой **fusermount -u /path**, однако, если соединение залипло (например, нет сети), то можно/нужно делать это из-под рута: sudo umount -f /path.

Удалённое исполнение кода

ssh может выполнить команду на удалённом сервере и тут же закрыть соединение. Простейший пример:

```
ssh user@server ls /etc/
```

Выведет нам содержимое /etc/ на server, при этом у нас будет локальная командная строка.

Некоторые приложения хотят иметь управляющий терминал. Их следует запускать с опцией -t:
`ssh user@server -t remove_command`

Кстати, мы можем сделать что-то такого вида:
`ssh user@server cat /some/file|awk '{print $2}' |local_app`

Это нас приводит следующей фиче:

Проброс stdin/out

Допустим, мы хотим сделать запрос к программе удалённо, а потом её вывод поместить в локальный файл

ssh user@8.8.8.8 command >my_file

Допустим, мы хотим локальный вывод положить удалённо

mycommand |scp — user@8.8.8.8:/path/remote_file

Усложним пример — мы можем прокидывать файлы с сервера на сервер: Делаем цепочку, чтобы положить stdin на 10.1.1.2, который нам не доступен снаружи:

mycommand | ssh user@8.8.8.8 «scp — user@10.1.1.2:/path/to/file»

Есть и вот такой головоломный приём использования pipe'a (любезно подсказали в комментариях в жж):

`tar -c * | ssh user@server "cd && tar -x"`

Tar запаковывает файлы по маске локально, пишет их в stdout, откуда их читает ssh, передаёт в stdin на удалённом сервере, где их cd игнорирует (не читает stdin), а tar — читает и распаковывает. Так сказать, scp для бедных.

Алиасы

Скажу честно, до последнего времени не знал и не использовал. Оказались очень удобными.

В более-менее крупной компании часто оказывается, что имена серверов выглядят так: spb-MX-i3.extrt.int.company.net. И пользователь там не равен локальному. То есть логиниться надо так: ssh ivanov_i@spb-MX-i3.extrt.int.company.net. Каждый раз печатать — туннельных синдромов не напасёшься. В малых компаниях проблема обратная — никто не думает о DNS, и обращение на сервер выглядит так: ssh root@192.168.1.4. Короче, но всё равно напрягает. Ещё большая драма, если у нас есть нестандартный порт, и, например, первая версия ssh (привет цискам). Тогда всё выглядит так: ssh -1 -p 334 vv_pupkin@spb-MX-i4.extrt.int.company.net. Удавиться. Про драму с scp даже рассказывать не хочется.

Можно прописать общесистемные alias'ы на IP (/etc/hosts), но это кривоватый выход (и пользователя и опции всё равно печатать). Есть путь короче.

Файл **~/.ssh/config** позволяет задать параметры подключения, в том числе специальные для серверов, что самое важное, для каждого сервера своё. Вот пример конфига:

```
Host ric
        Hostname ooo-рога-и-копыта.рф
        User Администратор
        ForwardX11 yes
        Compression yes
Host home
        Hostname myhome.dyndns.org
        User vasya
        PasswordAuthentication no
```

Все доступные для использования опции можно увидеть в **man ssh_config** (не путать с sshd_config).

Опции по умолчанию

Вы можете указать настройки соединения по умолчанию с помощью конструкции Host *, т.е., например:

```
Host *
User root
Compression yes
```

То же самое можно сделать и в /etc/ssh/ssh_config (не путать с /etc/ssh/ssh**d**_config), но это требует прав рута и распространяется на всех пользователей.

Проброс X-сервера

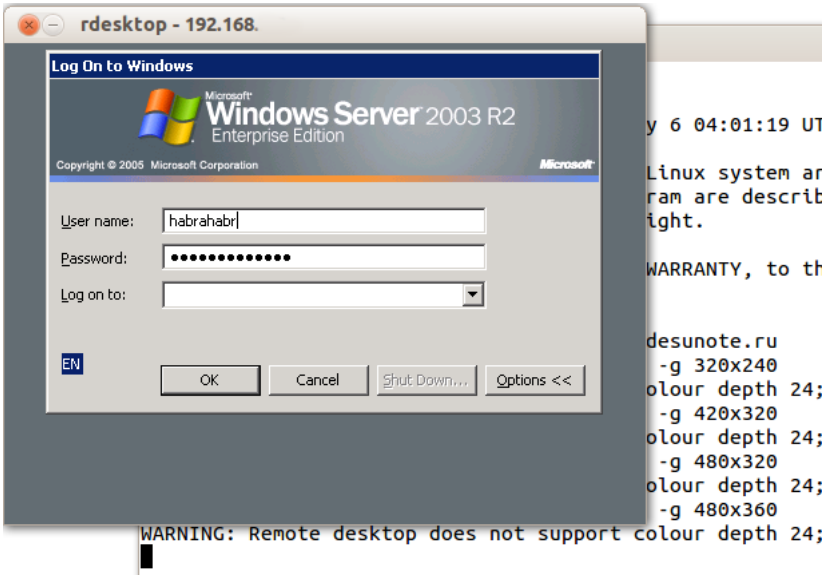
Собственно, немножко я проспойлерил эту часть в примере конфига выше. ForwardX11 — это как раз оно.

Теория: Графические приложения в юникс обычно используют X-сервер (wayland в пути, но всё ещё не готов). Это означает, что приложение запускается и подключается к X-серверу для рисования. Иными словами, если у вас есть голый сервер без гуя и есть локальный x-сервер (в котором вы работаете), то вы можете дать возможность приложениям с сервера рисовать у вас на рабочем столе. Обычно подключение к удалённом X-серверу — не самая безопасная и тривиальная вещь. SSH позволяет упростить этот процесс и сделать его совсем безопасным. А возможность жать трафик позволяет ещё и обойтись меньшим трафиком (т.е. уменьшить утилизацию канала, то есть уменьшить ping (точнее, latency), то есть уменьшить лаги).

Ключики: -X — проброс X-сервера. -Y проброс авторизации.

Достаточно просто запомнить комбинацию ssh -XYC user@SERVER.
В примере выше (названия компании вымышленные) я подключаюсь к серверу ooo-рога-и-копыта.рф не просто так, а с целью получить доступ к windows-серверу. Безопасность microsoft при работе в сети [urlspan]мы все хорошо знаем[/urlspan], так что выставлять наружу голый RDP неуютно. Вместо этого мы подключаемся к серверу по ssh, а дальше запускаем там команду rdesktop:
`ssh ric rdesktop -k en-us 192.168.1.1 -g 1900x1200`

и чудо, окошко логина в windows на нашем рабочем столе. Заметим, тщательно зашифрованное и неотличимое от обычного ssh-трафика.



Socks-proxy

Когда я оказываюсь в очередной гостинице (кафе, конференции), то местный wifi чаще всего оказывается ужасным — закрытые порты, неизвестно какой уровень безопасности. Да и доверия к чужим точкам доступа не особо много (это не паранойя, я вполне наблюдал как уводят пароли и куки с помощью банального ноутбука, раздающего 3G всем желающим с названием близлежащей кафешки (и пишущего интересное в процессе)).

Особые проблемы доставляют закрытые порты. То джаббер прикроют, то IMAP, то ещё что-нибудь.

Обычный VPN (pptp, l2tp, openvpn) в таких ситуациях не работает — его просто не пропускают. Экспериментально известно, что 443ий порт чаще всего оставляют, причём в режиме CONNECT, то есть пропускают «как есть» (обычный http могут ещё прозрачно на сквид завернуть).

Решением служит *socks-proxy* режим работы ssh. Его принцип: ssh-клиент подключается к серверу и слушает локально. Получив запрос, он отправляет его (через открытое соединение) на сервер, сервер устанавливает соединение согласно запросу и все данные передаёт обратно ssh-клиенту. А тот отвечает обратившемуся. Для работы нужно сказать приложениям «использовать socks-proxy». И указать IP-адрес прокси. В случае с ssh это чаще всего localhost (так вы не отдадите свой канал чужим людям).

Подключение в режиме sock-proxy выглядит так:

```
ssh -D 8080 user@server
```

В силу того, что чужие wifi чаще всего не только фиговые, но и лагливые, то бывает неплохо включить опцию -C (сжимать трафик). Получается почти что opera turbo (только картинки не жмёт). В реальном сёрфинге по http жмёт примерно в 2-3 раза (читай — если вам выпало несчастье в 64кбит, то вы будете мегабайтные страницы открывать не по две минуты, а секунд за 40. Фигово, но всё ж лучше). Но главное: никаких украденных кук и подслушанных сессий.

Я не зря сказал про закрытые порты. 22ой порт закрывают ровно так же, как «не нужный» порт джаббера. Решение — повесить сервер на 443-й порт. Снимать с 22 не стоит, иногда бывают системы с DPI (deep packet inspection), которые ваш «псевдо-ssl» не пустят.

Вот так выглядит мой конфиг:

/etc/ssh/sshd_config:
(фрагмент)
Port 22
Port 443

А вот кусок ~/.ssh/config с ноутбука, который описывает vpn

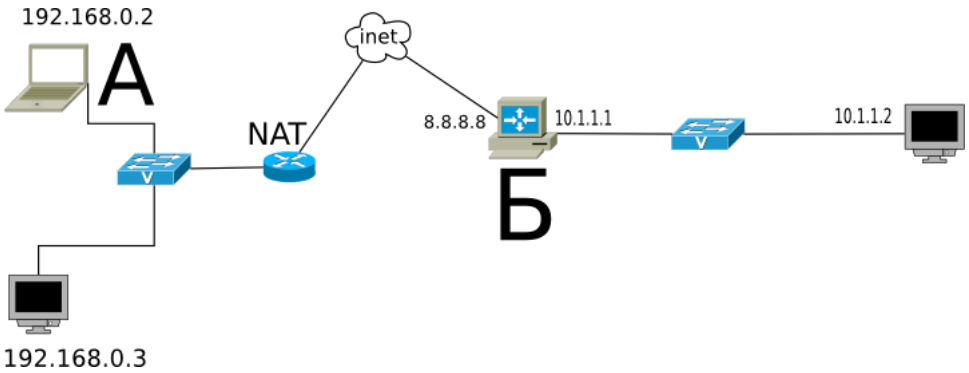
```
Host vpn
    Hostname desunote.ru
    User vasya
    Compression yes
    DynamicForward 127.1:8080
    Port 443
```

(обратите внимание на «ленивую» форму записи localhost — 127.1, это вполне себе законный метод написать 127.0.0.1)

Проброс портов

Мы переходим к крайне сложной для понимания части функционала SSH, позволяющей осуществлять головоломные операции по туннелированию TCP «из сервера» и «на сервер».

Для понимания ситуации все примеры ниже будут ссылаться на вот эту схему:



Комментарии: Две серые сети. Первая сеть напоминает типичную офисную сеть (NAT), вторая — «гейтвей», то есть сервер с белым интерфейсом и серым, смотрящим в свою собственную приватную сеть. В дальнейших рассуждениях мы полагаем, что «наш» ноутбук — А, а «сервер» — Б.

Задача: у нас локально запущено приложение, нам нужно дать возможность другому пользователю (за пределами нашей сети) посмотреть на него.

Решение: проброс локального порта (127.0.0.1:80) на публично доступный адрес. Допустим, наш «публично доступный» Б занял 80ый порт чем-то полезным, так что пробрасывать мы будем на нестандартный порт (8080).

Итоговая конфигурация: запросы на 8.8.8.8:8080 будут попадать на localhost ноутбука А.

```
ssh -R 127.1:80:8.8.8.8:8080 user@8.8.8.8
```

Опция **-R** позволяет перенаправлять с удалённого (**R**emote) сервера порт на свой (локальный).
Важно: если мы хотим использовать адрес 8.8.8.8, то нам нужно разрешить GatewayPorts в настройках сервера Б.
Задача. На сервере «Б» слушает некий демон (допустим, sql-сервер). Наше приложение не совместимо с сервером (другая битность, ОС, злой админ, запрещающий и накладывающий лимиты и т.д.). Мы хотим локально получить доступ к удалённому localhost'у.

Итоговая конфигурация: запросы на localhost:3333 на 'А' должны обслуживаться демоном на localhost:3128 'Б'.

```
ssh -L 127.1:3333:127.1:3128 user@8.8.8.8
```

Опция **-L** позволяет локальные обращения (**L**ocal) направлять на удалённый сервер.

Задача: На сервере «Б» на сером интерфейсе слушает некий сервис и мы хотим дать возможность коллеге (192.168.0.3) посмотреть на это приложение.

Итоговая конфигурация: запросы на наш серый IP-адрес (192.168.0.2) попадают на серый интерфейс сервера Б.

```
ssh -L 192.168.0.2:8080:10.1.1.1:80 user@8.8.8.8
```

Вложенные туннели

Разумеется, туннели можно перенаправлять.

Усложним задачу: теперь нам хочется показать коллеге приложение, запущенное на localhost на сервере с адресом 10.1.1.2 (на 80ом порту).

Решение сложно:

```
ssh -L 192.168.0.2:8080:127.1:9999 user@8.8.8.8 ssh -L 127.1:9999:127.1:80 user2@10.1.1.2
```

Что происходит? Мы говорим ssh перенаправлять локальные запросы с нашего адреса на localhost сервера Б и сразу после подключения запустить ssh (то есть клиента ssh) на сервере Б с опцией слушать на localhost и передавать запросы на сервер 10.1.1.2 (куда клиент и должен подключиться). Порт 9999 выбран произвольно, главное, чтобы совпадал в первом вызове и во втором.

Реверс-сокс-прокси

Если предыдущий пример вам показался простым и очевидным, то попробуйте догадаться, что сделает этот пример:

```
ssh -D 8080 -R 127.1:8080:127.1:8080 user@8.8.8.8 ssh -R 127.1:8080:127.1:8080 user@10.1.1.
```

Если вы офицер безопасности, задача которого запретить использование интернета на сервере 10.1.1.2, то можете начинать выдёргивать волосы на попе, ибо эта команда организует доступ в интернет для сервера 10.1.1.2 посредством сокс-прокси, запущенного на компьютере «А». Трафик полностью зашифрован и неотличим от любого другого трафика SSH. А исходящий трафик с компьютера с точки зрения сети «192.168.0/24» не отличим от обычного трафика компьютера А.

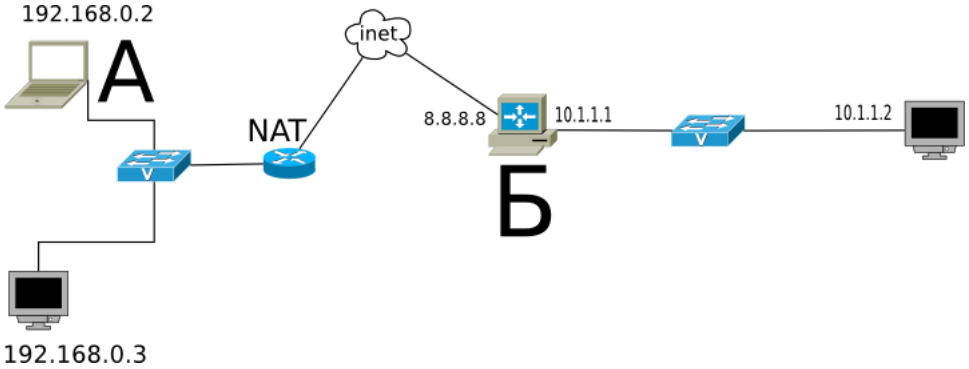
Проброс авторизации

Если вы думаете, что на этом всё, то…… впрочем, в отличие от автора, у которого «снизу» ещё не написано, читатель заранее видит, что там снизу много букв и интриги не получается.

OpenSSH позволяет использовать сервера в качестве плацдарма для подключения к другим серверам, даже если эти сервера недоверенные и могут злоупотреблять чем хотят.

Для начала о простом пробросе авторизации.

Повторю картинку:



Допустим, мы хотим подключиться к серверу 10.1.1.2, который готов принять наш ключ. Но копировать его на 8.8.8.8 мы не хотим, ибо там проходной двор и половина людей имеет sudo и может шариться по чужим каталогам. Компромиссным вариантом было бы иметь «другой» ssh-ключ, который бы авторизовывал user@8.8.8.8 на 10.1.1.2, но если мы не хотим пускать кого попало с 8.8.8.8 на 10.1.1.2, то это не вариант (тем паче, что ключ могут не только поюзать, но и скопировать себе «на чёрный день»).

ssh предлагает возможность форварда ssh-агента (это такой сервис, который запрашивает пароль к ключу). Опция **ssh -A**пробрасывает авторизацию на удалённый сервер.

Вызов выглядит так:

```
ssh -A user@8.8.8.8 ssh user2@10.1.1.2
```

Удалённый ssh-клиент (на 8.8.8.8) может доказать 10.1.1.2, что мы это мы только если мы к этому серверу подключены и дали ssh-клиенту доступ к своему агенту авторизации (но не ключу!).

В большинстве случаев это прокатывает.

Однако, если сервер совсем дурной, то root сервера может использовать сокет для имперсонализации, когда мы подключены.

Есть ещё более могучий метод — он превращает ssh в простой pipe (в смысле, «трубу») через которую насквозь мы осуществляем работу с удалённым сервером.

Главным достоинством этого метода является полная независимость от доверенности промежуточного сервера. Он может использовать поддельный ssh-сервер, логгировать все байты и все действия, перехватывать любые данные и подделывать их как хочет — взаимодействие идёт между «итоговым»

сервером и клиентом. Если данные оконечного сервера подделаны, то подпись не сойдётся. Если данные не подделаны, то сессия устанавливается в защищённом режиме, так что перехватывать нечего.

Настройка завязана на две возможности ssh: опцию -W (превращающую ssh в «трубу») и опцию конфига **ProxyCommand**(опции командной строки, вроде бы нет), которая говорит «запустить программу и присосаться к её stdin/out». Опции эти появились недавно, так что пользователи centos в пролёте.

Выглядит это так (циферки для картинки выше):

.ssh/config:

```
Host raep
     HostName 10.1.1.2
     User user2
     ProxyCommand ssh -W %h:%p user@8.8.8.8
```

Ну а подключение тривиально: `ssh raep`.

Повторю важную мысль: сервер 8.8.8.8 не может перехватить или подделать трафик, воспользоваться агентом авторизации пользователя или иным образом изменить трафик. Запретить — да, может. Но если разрешил — пропустит через себя без расшифровки или модификации. Для работы конфигурации нужно иметь свой открытый ключ в authorized_keys как для user@8.8.8.8, так и в user2@10.1.1.2

Разумеется, подключение можно оснащать всеми прочими фенечками — прокидыванием портов, копированием файлов, сокс-прокси, L2-туннелями, туннелированием X-сервера и т.д.

Сжатие ssh-трафика, инициируемое со стороны ssh-клиента.

```
В конец файла <strong>/etc/ssh/ssh_config</strong> добавляем 2 строки:
 <em> Compression yes
 CompressionLevel 9</em>
 (sudo gedit, sudo nano вам в помощь).
 Всё. Открываем новое соединение по ssh и радуемся.
```

Для обновления Ubuntu (Debian) через ssh socks proxy:

```
echo 'Acquire::socks::proxy "socks://localhost:3128/";' | sudo tee -a /etc/apt/apt.conf
ssh -CD localhost:3128 user@remote.host
http://habrahabr.ru/post/122445/
```