# **Настраиваем и защищаем SSH**

Защищаем администрирование и передачу данных через SSH на различных платформах

Привет.

Протокол [SSH](https://www.atraining.ru/ssh-secure-shell/) уже продолжительное время является основным используемым средством для удалённого администрирования сетевых устройств и unix-ОС.

Гайдов по настройке SSH много. Часть из них устарела. Часть состоит из сверхценных советов “перевесьте на другой порт, например 2222, и не работайте под рутом”. Часть состоит из копипасты чьей-то конфиги с подписью “это секурно”. Часть содержит откровенно вредные советы.

Попробуем избежать всего вышеперечисленного.

В качестве подопытных и настраиваемых систем будет выбран Centos Linux 7й версии и семейство ОС Cisco IOS / NX-OS, а также Windows Server 2016, имеющий с билда 1709 встроенную поддержку SSH.

Безусловно, можно [добавить поддержку SSH и в Windows-системы более старых версий](https://github.com/PowerShell/Win32-OpenSSH/releases/), но отдельно останавливаться на настройках этого варианта нет особого смысла, т.к. настройки эти будут урезанным функционалом того же OpenSSH. В общем, предлагаемые меры по улучшению безопасности SSH будут применимы для всех ОС.

Если используется другая \*nix-ОС, а не redhad-based, то расположение конфигурационных файлов может быть иным. Это принципиально ничего не меняет.

Также, Про настройку SSH в Cisco IOS есть [отдельный вебинар из трека Advanced CCNA](https://www.atraining.ru/advanced-ccna-terminal-ssh-telnet/), запись которого бесплатно доступна на нашем [YouTube](https://www.youtube.com/atrainingru" \o "Официальный канал учебного центра Advanced Training на YouTube" \t "_blank), а настройку SSH в плане разграничения доступа – типа *“хочу, чтобы на коммутаторы и маршрутизаторы могли заходить с админскими правами только участники группы ATRAINING\Cisco Admins из домена Active Directory”* – мы делаем на онлайн-курсе [Cisco IINS 3.0](https://www.atraining.ru/cisco-iins/" \o ).

* [Устанавливаем SSH](https://www.atraining.ru/ssh-armoring/#ssh-install)
* [Фиксируем версию SSHv2](https://www.atraining.ru/ssh-armoring/#ssh-version)
* [Ограничиваем перечень протоколов подтверждения подлинности сервера](https://www.atraining.ru/ssh-armoring/#ssh-publickeyalgo)
* [Настройка обмена ключами / создания ключевого материала](https://www.atraining.ru/ssh-armoring/#ssh-dh-exchange)
* [Перегенерация модулей SSHv2](https://www.atraining.ru/ssh-armoring/#ssh-moduli)
* [Настраиваем используемый алгоритм шифрования трафика SSHv2](https://www.atraining.ru/ssh-armoring/#ssh-secretkeyalgo)
* [Отключаем сжатие SSHv2-трафика](https://www.atraining.ru/ssh-armoring/#ssh-compression)
* [Выбираем Message Authentication Code у SSHv2](https://www.atraining.ru/ssh-armoring/#ssh-mac)
* [Сетевые настройки SSHv2](https://www.atraining.ru/ssh-armoring/#ssh-network)
* [Ограничения SSH на процедуру подключения к сеансу](https://www.atraining.ru/ssh-armoring/#ssh-logon)
* [Группы и пользователи в настройке SSH-сервера](https://www.atraining.ru/ssh-armoring/#ssh-accounts)

Начнём.

Устанавливаем SSH

Для сетевого оборудования этот шаг не нужен – поддержка SSH практически всегда интегрирована даже в самый минималистичный вариант ОС. Исключением можно считать разве что достаточно старые Cisco IOS варианта W/O CRYPTO.

Данный вариант IOS – с осознанным удалением всех даже малозначительных следов того, что относится к криптографии (вплоть до отсутствия **(config)#enable secret**) был нужен для экспорта в страны с жёстким законодательством в плане ввоза криптографии. Это, кстати, не только очевидные Куба + Северная Корея + Сирия + Иран, но и, например, Австралия.

Если у вас подобный IOS – это может быть, к примеру, на Catalyst 2950, которые хоть и древние, но вполне могут попадаться в production – обновите его. Без обновления нужные для SSH features, в частности:

* Secure Shell SSH Version 1 Integrated Client;
* Secure Shell SSH Version 1 Server Support;
* Secure Shell SSH Version 2 Server Support;

будут физически отсутствовать в составе IOS.

Добавление поддержки SSH в Windows Server

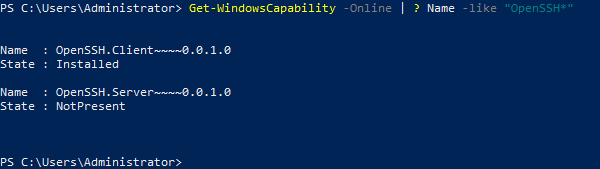
Начиная с Windows Server 2016 build 1709, поддержка SSH добавлена в платформу Windows.

Включить её несложно.

Первым делом – выясним, какая версия клиента и сервера OpenSSH есть в доступных для установке репозиториях прямо сейчас. Это нужно, чтобы когда версии пойдут увеличиваться (на момент написания версия одна, 1.0), то мы не получали бы проблем вида “что-то скрипт, ставящий конкретную версию, не срабатывает”. Сделаем это вот таким командлетом:

**Get-WindowsCapability -Online | ? Name -like "OpenSSH\*"**

У меня Windows Server 2019, поэтому вывод выглядит так:

[Выясняем поддерживаемые версии SSH на Windows Server  
(кликните для увеличения до 600 px на 169 px)](https://cdn.atraining.ru/i/ssh-secure-shell-armoring-4.png)

Видим, что SSH-клиент уже установлен изначально, а SSH-сервер – нет. У Windows Server 2016 вывод будет чуть другой, там ничего не установлено изначально. ОК, будем ставить SSH-сервер:

**Add-WindowsCapability -Online -Name OpenSSH.Server~~~~0.0.1.0**

Клиент, как понятно, если отсутствует, то ставится аналогично. Тильд – четыре штуки, не опечатайтесь.

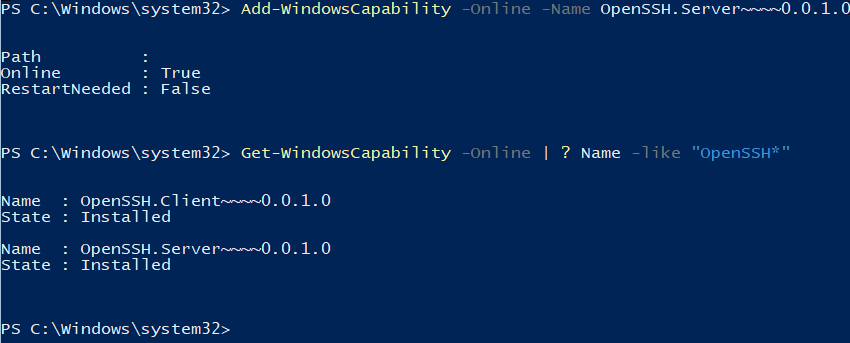
Если получили ошибку **0x800f0950**, не отчаивайтесь – это просто Feature-On-Demand не может найти репозиторий. Попробуйте через старый добрый DISM:

**dism /Add-Capability /CapabilityName:OpenSSH.Server~~~~0.0.1.0 /LimitAccess /Online**

Проверим, что всё установилось:

**Get-Service sshd**

ну либо просто запросим ещё раз **Get-WindowsCapability -Online | ? Name -like "OpenSSH\*"**:

[Проверяем установленный SSH Server на Windows Server  
(кликните для увеличения до 850 px на 343 px)](https://cdn.atraining.ru/i/ssh-secure-shell-armoring-5.png)

Если всё ОК, то сервис **sshd** у нас появился – правда, остановленный. Не включайте его сразу – вначале надо провести минимальные настройки, про которые будет в соответствующем разделе статьи.

Фиксируем версию SSHv2

Стартовый зоопарк версий SSH – SSH 1.3, потом SSH 1.5, потом “специальная версия от Cisco, показывающая, что сервер умеет и 2.0 и предыдущие”, которая 1.99 – сейчас совершенно не актуален, т.к. весь софт умеет SSHv2. Найти ПО, которое поддерживает только SSH 1.x – реально сложная задача. Поэтому, конечно, убедитесь, что такого ПО у вас нет, и обновите при необходимости – но рассматривать мы будем функциональность и безопасность только второй версии SSH.

Включаем SSH 2.х в Cisco IOS

Фиксируется она просто – для Cisco IOS это будет команда:

**atraining(config)#ip ssh version 2**

Если на данный момент вы ещё не создали ни одной ключевой пары, пригодной для работы SSHv2, то будет примерно такое сообщение:

**Please create RSA keys to enable SSH (and of atleast 768 bits for SSH v2).**

Это не страшно, разве что заметим, что у SSHv2 есть “нижние” требования по длине ключа в ключевой паре. Впрочем, мы не будем пытаться создать ключи, не подпадающие под это ограничение – времена, когда например 512ти битовые ключи RSA активно использовались, ушли.

Если ключи ещё не созданы, то делается это просто:

**atraining(config)#crypto key generate rsa modulus 2048 label SSH\_KEYS**

Параметры несложны – **rsa** задаст алгоритм (в новых IOS добавляется вариант **ec**), **modulus** – битовую длину (можете выставить её в 4096, так будет, безусловно, безопаснее), метка **label** – чтобы создать “именованную для конкретного назначения” ключевую пару.

В ряде версий Cisco IOS есть ограничение на хранимые ключевые пары – до 10 на устройство и до 2 на пользователя – поэтому может быть выведено предупреждение типа *“внимание, новая ключевая пара затрёт предыдущую”*. Отслеживайте это.

Теперь скажем SSH, чтобы он использовал именно эту пару:

**atraining(config)#ip ssh rsa keypair-name SSH\_KEYS**

Если всё ОК, нам выведется примерно такое:

**%SSH-5-DISABLED: SSH 2.0 has been disabled %SSH-5-ENABLED: SSH 2.0 has been enabled**

Это будет обозначать, что переключение с “какие попало ключи” на “явно указанная пара” произошло успешно.

Если нам надо также сообщить, что к данному устройству должны подключаться только используя SSH, а не telnet, например, то это делается следующим образом – вначале заходим в режим конфигурирования VTY:

**atraining(config)#line vty 0 5** (или **line vty 0 15** – зависит от модели)

и явно указываем, что входящие подключения должны быть исключительно по SSH:

**atraining(config-line)#transport input ssh**

Включаем SSH 2.х в Cisco NX-OS

С Cisco Nexus’ами будет просто – они поддерживают только SSH 2.x, поэтому дополнительных действий по ограничению возможности подключения старыми версиями SSH – не нужно.

Не забудьте разве что явно включить использование SSH:

**atraining-nx(config)#feature ssh**

Если ключей нет, то можно их в явном виде создать, сразу нужной длины и с нужным алгоритмом. Для этого выключим SSH, сгенерим ключи и включим SSH – тогда он сразу “подхватит” новые:

**atraining-nx(config)#no feature ssh**

**atraining-nx(config)#ssh key rsa 2048**

**atraining-nx(config)#feature ssh**

Параметры у **ssh key** тривиальны, разве что замечу, что автоматически перезаписываться старые ключи не будут, если надо, чтобы перезаписались – добавьте в конце команды **force**

Включаем SSH 2.х в sshd

В настройках **sshd** нам надо будет добавить (либо заменить) строку:

**Protocol 2**

Включаем SSH 2.х в Windows Server

В каталоге **%WINDIR%\System32\OpenSSH\** будет стандартный файл конфигурирования OpenSSH, **sshd\_config\_default** – и там в теории может быть настройка про “только вторую версию”, но по факту всегда используется только SSHv2. Поэтому специального действия для включения SSHv2 на Windows Server нет.

Теперь ограничим список протоколов, которые могут использоваться для подтверждения подлинности при подключении клиента.

Ограничиваем перечень протоколов подтверждения подлинности сервера

SSH поддерживает несколько вариантов подтверждения подлинности узла, к которому идёт подключение – с использованием алгоритмов DSA, ECDSA, RSA и распространённой эллиптической кривой 25519.

DSA нам сразу не нравится, т.к. он умеет только ключи по 1024 бита и есть мнение, высказанное тов.Сноуденом, что NSA не просто так активно любит DSA.

Поэтому сразу отсечём вариант использования DSA.

Фиксируем в Cisco IOS использование RSA для host identification

У Cisco это будет просто:

**atraining(config)#ip ssh server algorithm hostkey ssh-rsa**

Фиксируем алгоритмы host identification у sshd

В настройках **sshd** нам надо будет пойти в папку **/etc/sysconfig/sshd** и там поправить строку **AUTOCREATE\_SERVER\_KEYS**:

**AUTOCREATE\_SERVER\_KEYS="RSA ED25519"**

Как понятно, это настройка “в вариантах каких алгоритмов ключи хоста автогенерить”. Замечу, что если стоит задача высокой надёжности, то 4096ти битовый RSA – это правильный выбор, а если скорости – то EC 25519 будет предпочтительнее.

После этого зайдём в каталог настроек **sshd** – в нашем варианте это будет **/etc/ssh** – и удалим там файлы неиспользуемых алгоритмов с ключами хоста. То есть из списка вида:

* **ssh\_host\_dsa\_key**
* **ssh\_host\_dsa\_key.pub**
* **ssh\_host\_ecdsa\_key**
* **ssh\_host\_ecdsa\_key.pub**
* **ssh\_host\_rsa\_key**
* **ssh\_host\_rsa\_key.pub**
* **ssh\_host\_ed25519\_key**
* **ssh\_host\_ed25519\_key.pub**

оставим только нужные.

Если боитесь удалить лишнее – не бойтесь, при запуске сервис **sshd** прочитает конфигурационный файл и создаст отсутствующие, если надо.

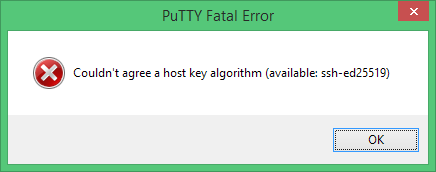
После этого в конфигурационном файле надо найти директивы об использовании этих файлов с ключами – выглядят они (директивы) обычно так:

* **HostKey /etc/ssh/ssh\_host\_rsa\_key**
* **HostKey /etc/ssh/ssh\_host\_dsa\_key**
* **HostKey /etc/ssh/ssh\_host\_ecdsa\_key**
* **HostKey /etc/ssh/ssh\_host\_ed25519\_key**

и оставить только нужные, закомментировав остальные путём добавления символа **#** (диез) перед строками.

Если директива про использование файла будет отсутствовать или файл будет недоступен, то соответствующий алгоритм не сможет использоваться для подтверждения подлинности узла, что может привести к проблемам с подключением.

Например, если оставить только модный Ed25519, а RSA выключить, то широко используемый PuTTY может говорить примерно такое:

[Ошибка PuTTY - couldnt agree a host key algorithm (available: ssh-ed25519)  
(кликните для увеличения до 436 px на 172 px)](https://cdn.atraining.ru/i/ssh-secure-shell-armoring-1.png)

Это, кстати, лечится обновлением PuTTY на последнюю версию, в которой есть поддержка новых алгоритмов. Так что сделайте это заранее.

Оставив только нужные типы ключей – обычно это ED25519 и RSA – нужно пересоздать их вручную, т.е. отдавать на откуп “пусть сервер сам это сделает на следующем рестарте сервиса” – не очень хорошая идея.

Делается это так:

Для Ed25519: **ssh-keygen -t ed25519 -f ssh\_host\_ed25519\_key -N ""**

Для RSA: **ssh-keygen -t rsa -b 4096 -f ssh\_host\_rsa\_key -N ""**

Делая этот шаг вручную, вы можете быть уверенны, что ключ RSA получится нужной, а не default’ной длины.

Ограничиваем протоколы для подтверждения подлинности SSH в Windows Server

Схема та же – идём в файл **sshd\_config\_default** и там оставляем только:

* **HostKey ssh\_host\_rsa\_key**
* **HostKey ssh\_host\_ed25519\_key**

, если надо и Ed25519 и RSA, или вообще только Ed25519. После – генерим ключи:

Для Ed25519: **.\ssh-keygen -t ed25519 -f ssh\_host\_ed25519\_key**

Для RSA: **.\ssh-keygen -t rsa -b 4096 -f ssh\_host\_rsa\_key**

Далее привязываем эти ключи командой **ssh-add**:

**.\ssh-add ssh\_host\_ed25519\_key**

В принципе всё, сервис sshd можно запускать.

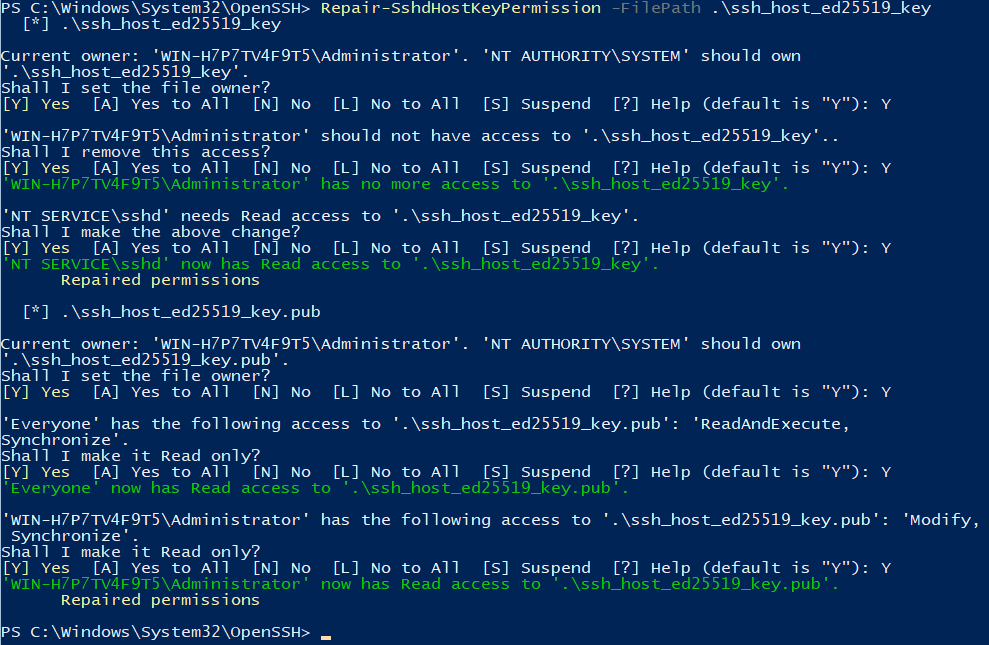
Фирменный костыль от Microsoft

Плод сумрачного разума разработчиков Microsoft – костыль для прописывания прав на файлы с ключами. Его рекомендуют запускать, но если вы психически здоровы и догадываетесь, что у сервиса должны быть права на файлы ключей, которые он используют, вы обойдётесь и без него. Впрочем, если что, ставится он так:

**Install-Module -Force OpenSSHUtils**

**Repair-SshdHostKeyPermission -FilePath**

Оно потребует NuGet и в общем … смотрите сами:

[Имбецильность - не порок, а преимущество DevOps  
(кликните для увеличения до 989 px на 465 px)](https://cdn.atraining.ru/i/ssh-secure-shell-armoring-6.png)

Прикинуть уровень целевой аудитории, которой для этих действий надо скачать пакет модулей и выполнить командлет – т.е. не включить мозг и раздать права, а зазубрить командочку – можно самостоятельно. Уверен, что Microsoft выдаст по этому модулю пару-тройку [MVP](https://www.atraining.ru/mvp/).

Теперь про обмен ключевой информацией.

Настройка обмена ключами / создания ключевого материала

Вариантов DH, или алгоритма Диффи-Хеллмана-Меркля – множество. Не углубляясь в данной статье в матчасть по самому алгоритму, посмотрим, как нам можно укрепить фронт с этой стороны.

Обмен ключами и совместная генерация ключевого материала для конкретной сессии – очень серьёзная тема в безопасности. Наша задача – избежать варианта, когда у нас будет возможен сценарий “аутентификация стойкая, а обмен ключами нет”.

Посмотрим, как сейчас у нас настроен DH в SSH на Cisco IOS:

**atraining#sh ip ssh | inc Diffie**

**Minimum expected Diffie Hellman key size : 1024 bits**

Плохо. Надо не ниже 2048 бит. Задаём это командой:

**atraining(config)#ip ssh dh min size 2048**

2048 бит – это 14я группа по [RFC 3526](https://tools.ietf.org/html/rfc3526). Можно поставить и 16ю группу, это 4096 бит – так ещё лучше, но надо дополнительно проработать вопрос совместимости со стороны клиентского ПО. Потому что поддержка 1й и 14й DH-групп есть везде, это стандарт по [RFC 4253](https://tools.ietf.org/html/rfc4253) – а вот остальные MODP-группы DH поддерживаются опционально. Главное, на самом деле, отсечь 1, 2 и 5 группы – наша настройка это и делает.

В варианте **sshd** нам надо будет добавить в конфигурацию строчку вида:

**KexAlgorithms diffie-hellman-group14-sha1**

Можно добавить туда и другие алгоритмы, если есть поддержка их со стороны клиента – например, **curve25519-sha256**. Главное – ограничить выбор, чтобы нельзя было согласовать “слабые” варианты. “Усиленный” вариант будет выглядеть, например, так:

**KexAlgorithms curve25519-sha256@libssh.org,diffie-hellman-group18-sha512,diffie-hellman-group16-sha512**

Здесь и длина ключа RSA серьёзная – 8192-4096 бит, и хэш-алгоритмы тоже используются “с запасом”. Если есть техническая возможность – имеет смысл использовать подобные настройки.

Зачастую предлагается заменить **diffie-hellman-group14-sha1** на **diffie-hellman-group-exchange-sha256** – мол, *“там же SHA-2, это круче, чем SHA-1”*. Это так, но **diffie-hellman-group-exchange-sha256** – это “любая DH-группа, а целостность – SHA-2”. Т.е. уходит критерий “минимальная битовая длина”, а он важен. SHA-2 тут не особо даст плюсы – потому что предположить существование атаки, которая “на лету” в момент DH-обмена успевает подделать SHA-1 пока трудно.

Весь комплект EC-вариантов от NIST – **ecdh-sha2-nistp256,ecdh-sha2-nistp384,ecdh-sha2-nistp521** – лучше не использовать вообще; помимо [известных вопросов к NIST](https://cdn.atraining.ru/docs/15-shumow.pdf) есть и новые предположения – впрочем, это тема для отдельной статьи, потому что тема [выбора эллиптических кривых](https://www.atraining.ru/ec-curves-key-exchange/) достаточно актуальна.

Алгоритмы выбрали – надо настроить вопрос обновления ключевого материала, он же rekeying.

Будет не очень хорошо, если всегда будет выполняться правило *“Одна сессия = один набор битового материала после стартового обмена ключами”*. Есть целый класс потенциальных атак, начинающихся с условия “надо набрать не менее N байт данных, зашифрованных этим методом/ключом”. Поэтому лучше, если сессия или продолжительная, или в ней передаётся много данных, периодически менять ключи.

За это в **sshd** будет отвечать настройка **RekeyLimit**. Она действует только для SSHv2, и по умолчанию имеет значение “отключено”:

**RekeyLimit default none**

Мы выставим её в разумное значение “менять ключи, когда передадим очередной гигабайт данных” – это будет выглядеть так:

**RekeyLimit 1G**

Очень интересным моментом является то, что изначально в стандарте SSH не было жёстко прописано, кто из пары клиент-сервер должен запрашивать rekeying. То есть по идее, могут оба. И настройка долгое время была “клиентской”, то есть в клиентском ПО можно было выставить “через сколько минут/мегабайт запросить”, а у сервера это как-то не предполагалось. В результате в части руководств прямо указывается, что сервер не может делать rekeying. Может.

В случае Cisco IOS (кстати, только в некоторых IOS) настройка параметров rekey делается командой:

**atraining(config)#ip ssh rekey**

с параметром **time** или **volume**. Заметьте, или-или.

В конфигурации могут быть параметры **KeyRegenerationInterval** и **ServerKeyBits** – они используются только для SSH 1.x, смело удаляйте; SSHv2 их не использует.

ОК, с этим разобрались – теперь от подтверждения подлинности хоста перейдём к генерации криптографических данных для каждой сессии.

Перегенерация модулей SSHv2

При подключении и стартовой совместной генерации ключевого материала по алгоритму Диффи-Хеллмана-Меркля системе нужно быстро откуда-то достать новое простое число, притом большого размера. Чтобы не делать эту трудоёмкую вычислительную задачу в момент подключения, замедляя оное, используется заранее созданный массив таких чисел и их параметров.

В момент установки ОС и установки/обновления **sshd** это действие не производится, поэтому оно обязательно должно быть выполнено администратором.

**ssh-keygen -G "${HOME}/moduli2048.bak" -M 127 -v -b 2048 -W 2**

Я попросил сгенерить новые “предварительно заготовленные числа” в файл **moduli2048.bak** в домашнем каталоге (файл, откуда **sshd** будет брать эти числа, обычно называется **/etc/ssh/moduli**), выделил под это побольше оперативной памяти (ключ **-M 127**) – 127 мегабайт, а не 4 стандартных, т.к. процесс генерации любит оперативную память, установил битовую длину в 2048 (для доступных из Интернета узлов лучше ставить 4096, но учтите – генериться это будет долго), выставил основание 2 (**-W 2** или **-W 5** – на безопасность не влияет, а на скорость при последующем использовании – может).

Когда вы сделаете такое же действие, система подумает какое-то время и выдаст подобное: **Increased memory: 127 MB; need 4190208 bytes Sieve next 2130706432 plus 2047-bit Sieved with 203277289 small primes in 360 seconds Found 1468150 candidates**

Как видим, процесс генерации занял 6 минут ровно. В результате получился файл с пачкой “заготовок” для 2048-битового обмена/совместной генерации ключевого материала. Кратко отвлечёмся на то, как устроен данный файл.

Формат файла moduli

Файл состоит из краткого заголовка-комментария, описывающего содержимое колонок таблицы (выглядит он обычно так – **# Time Type Tests Tries Size Generator Modulus**) и строк, в которых содержатся интересующие нас параметры.

* **Time** – это время генерации, формат YYYYMMDDHHMMSS.
* **Type** – это, как понятно, тип. По результатам работы генератора у нас будут сплошь type 4 – это так называемые простые [числа Софи Жермен](https://oeis.org/A005384) – для них выполняется правило, что у числа Софи Жермен p число 2\*p + 1 также будет простым. После того, как мы проверим список чисел-кандидатов на простоту, тип поменяется на двойку – это будут уже “безопасные простые числа”.
* **Tests** – битовая маска, говорящая о том, какие тесты прошло данное число, и иногда сообщающая о прискорбных результатах. Если это значение нуль, то число вообще не проверялось на простоту. Если включен первый бит (**0x01**) – проверялось и оказалось составным (такие использовать совсем не надо). Бит **0x02** говорит о том, что число выжило после решета Эратосфена (у свежесозданных “заготовок” будет как раз такое значение). Бит **0x04** скажет о том, что число выжило после теста Миллера-Рабина – поэтому после второго шага по проверке “заготовок” у всех хороших чисел в колонке Tests будет стоять число 6 – что говорит о том, что они прошли оба теста.
* **Trials** – количество тестов, проведённых над числом (будет впечатляющим, но тут важнее вопрос, какие именно это тесты – поэтому сотни миллионов итераций не равны гарантии надёжности).
* **Size** – длина числа в битах. Чаще всего вы увидите “странные” значения – не 1024/2048/4096/8192, а на бит меньше. Это нормально, т.к. у простого числа есть как минимум два известных бита – первый и последний. Первый будет единицей, т.к. это число имеет фиксированную длину – если бы число могло начинаться с нуля или пачки нулей, оно было бы короче заданного размера, а это недопустимо – мы заказывали генерацию чисел именно нужно длины. Последний будет единицей, т.к. если у числа последний бит нуль, то это число чётное, а чётные числа редко (только в случае двойки) бывают простыми. Поэтому в принципе можно и сэкономить на хранении простых чисел, предположив два вышеупомянутых тезиса.
* **Generator** – основание – т.е. 2 или 5 (или нуль – тогда тоже 2).
* **Modulus** – само число.

Продолжим.

Эти заготовки теперь надо прогнать через процедуру тестирования, чтобы отсеять модули, которые “слишком легко” будут разлагаться на множители, или вообще не будут простыми числами. Т.е. процедура генерации создаёт именно “кандидатов, предположительно годных для ношения гордого звания простого числа”, а тестирование отсеивает из них тех, кто явно не подходит.

**ssh-keygen -T /etc/ssh/moduli -f "${HOME}/moduli2048.bak" -a 500**

Параметры тут очевидны – где брать тестируемые и куда складывать готовые – разве что хочу отметить **-a 500**. Этот параметр – это “глубина” проверки, по умолчанию она равняется 100, я выставляю 500 для более тщательной проверки и отсева. Так как мы делаем разовую процедуру, то увеличение времени работы тут не критично. А оно будет, притом неслабое. Система начнёт отчитываться на консоль раз в 5 минут о том, как двигается процесс – в моём случае это выглядит так: **[testuser@atraining ~]# ssh-keygen -T /etc/ssh/moduli -f "${HOME}/moduli2048.bak" -a 500 processed 31311 of 1468725 (2%) in 0:05, ETA 3:49 processed 61015 of 1468725 (4%) in 0:10, ETA 3:51 processed 93610 of 1468725 (6%) in 0:15, ETA 3:40 processed 114521 of 1468725 (7%) in 0:20, ETA 3:57 processed 145914 of 1468725 (9%) in 0:25, ETA 3:47 processed 180592 of 1468725 (12%) in 0:30, ETA 3:34 processed 204619 of 1468725 (13%) in 0:35, ETA 3:37 processed 233644 of 1468725 (15%) in 0:40, ETA 3:32 processed 263243 of 1468725 (17%) in 0:45, ETA 3:27 processed 290385 of 1468725 (19%) in 0:50, ETA 3:24 processed 316302 of 1468725 (21%) in 0:55, ETA 3:21 processed 350252 of 1468725 (23%) in 1:00, ETA 3:12 processed 381090 of 1468725 (25%) in 1:05, ETA 3:06 processed 407552 of 1468725 (27%) in 1:10, ETA 3:03 processed 432599 of 1468725 (29%) in 1:15, ETA 3:00 processed 463306 of 1468725 (31%) in 1:20, ETA 2:54 processed 494272 of 1468725 (33%) in 1:25, ETA 2:48 processed 521319 of 1468725 (35%) in 1:30, ETA 2:44 processed 547391 of 1468725 (37%) in 1:35, ETA 2:40 processed 577760 of 1468725 (39%) in 1:40, ETA 2:34 processed 607799 of 1468725 (41%) in 1:45, ETA 2:29 processed 634402 of 1468725 (43%) in 1:50, ETA 2:25 processed 664471 of 1468725 (45%) in 1:55, ETA 2:19 processed 690995 of 1468725 (47%) in 2:00, ETA 2:15 processed 717235 of 1468725 (48%) in 2:05, ETA 2:11 processed 745487 of 1468725 (50%) in 2:10, ETA 2:06 processed 774837 of 1468725 (52%) in 2:15, ETA 2:01 processed 811298 of 1468725 (55%) in 2:20, ETA 1:54 processed 843016 of 1468725 (57%) in 2:25, ETA 1:48 processed 867423 of 1468725 (59%) in 2:30, ETA 1:44 processed 895713 of 1468725 (60%) in 2:35, ETA 1:39 processed 922506 of 1468725 (62%) in 2:40, ETA 1:35 processed 953693 of 1468725 (64%) in 2:45, ETA 1:29 processed 984896 of 1468725 (67%) in 2:50, ETA 1:23 processed 1012815 of 1468725 (68%) in 2:55, ETA 1:19 processed 1035874 of 1468725 (70%) in 3:00, ETA 1:15 processed 1066079 of 1468725 (72%) in 3:05, ETA 1:10 processed 1097883 of 1468725 (74%) in 3:10, ETA 1:04 processed 1124370 of 1468725 (76%) in 3:15, ETA 0:59 processed 1150179 of 1468725 (78%) in 3:20, ETA 0:55 processed 1178185 of 1468725 (80%) in 3:25, ETA 0:50 processed 1208345 of 1468725 (82%) in 3:31, ETA 0:45 processed 1231626 of 1468725 (83%) in 3:36, ETA 0:41 processed 1256634 of 1468725 (85%) in 3:41, ETA 0:37 processed 1285065 of 1468725 (87%) in 3:46, ETA 0:32 processed 1314094 of 1468725 (89%) in 3:51, ETA 0:27 processed 1344108 of 1468725 (91%) in 3:56, ETA 0:21 processed 1375328 of 1468725 (93%) in 4:01, ETA 0:16 processed 1410594 of 1468725 (96%) in 4:06, ETA 0:10 processed 1439388 of 1468725 (98%) in 4:11, ETA 0:05 processed 1465957 of 1468725 (99%) in 4:16, ETA 0:00 Found 917 safe primes of 1223961 candidates in 15426 seconds [testuser@atraining ~]#**

Я специально показываю полный вывод команды, чтобы можно было оценить и масштабы времени, и КПД отсеивания “плохих” модулей. Из 1.2 с лишним миллиона потенциальных primes прошли тесты лишь около 900! За четыре с лишним часа.

Кстати, процесс не распараллеливается – т.е. добавлением ядер вопрос не ускорить. У той машины, на которой это делалось, их 12 – это никак не помогло ускорить процесс, одно ядро лежало под 100%, другие простаивали.

Чтобы можно было оценить влияние количества тестов на общее время выполнения – вот так выглядит вариант с **-a 10**: **processed 47249 of 1468725 (3%) in 0:05, ETA 2:30**

(только первая строка, для сравнения этого достаточно).

Заметно, что оценка времени ощутимо изменилась, но не в 50 раз, как количество тестов – т.е. “минимальный” комплект проверок всё ж завязан на битовость ключа и общее число candidates.

После всего этого временный файл с расширением **bak** надо удалить, он больше не нужен, и наличие его вне каталога, доступного для **sshd** – опасно.

Добавлю ещё один момент – при запуске **ssh-keygen -T** в файл **/etc/ssh/moduli** новые строки будут добавляться, а старые при этом – остаются. То есть если вы не стёрли этот файл перед генерацией, в нём будут и предыдущие и новые модули. Если так, то надо как-то почистить от старых – например, можно отфильтровать по критерию длины:

**awk '$5 > 2000' /etc/ssh/moduli > "${HOME}/moduli.2048"**

Этой командой мы скопировали в **moduli.2048** только те строки, в которых в 5й колонке, где длина, значение более 2000.

Подходы могут быть разными, в общем просто учитывайте, что к файлу новое дописывается, а не замещается.

В случае с Cisco модули генерятся “на лету”, но не особо стойким методом, поэтому там всё немножко грустно по части потенциальных возможностей для нарушителей.

Теперь про главный протокол шифрования – которым шифруются сами данные.

Настраиваем используемый алгоритм шифрования трафика SSHv2

SSH может использовать различные алгоритмы для шифрования трафика, поэтому наша задача – упростить ему выбор и свести его к минимальному перечню надёжных алгоритмов.

Старые варианты, в частности 3DES, нас не интересуют – и в силу вопросов по стойкости, и в силу отсутствия аппаратного ускорения. Нам проще использовать AES.

Но какой? Помимо выбора длины в битах, нам надо не ошибиться с выбором режима работы AES. Нам не подойдёт простой AES-CBC, потому что в 2008 году была продемонстрирована потенциальная уязвимость в SSH с AES-CBC, из-за которой можно получить до 32х бит исходных данных. Наш выбор – AES256-CTR.

В Cisco IOS это будет задаваться так:

**atraining(config)#ip ssh server algorithm encryption aes256-ctr**

После **aes256-ctr** можно указать и другие алгоритмы, в этом случае будет задаваться список согласовываемых с клиентом вариантов; однако найти клиентское ПО, которое не поддерживает AES – сложно.

В настройках **sshd** нам надо будет добавить (либо заменить) строку:

**Ciphers aes256-ctr**

в конфигурационном файле. Ранее была настройка **Cipher**, в единственном числе, но она используется только в SSH 1.x, поэтому если она есть в конфигурационном файле – спокойно удаляйте строку с ней.

Узнать, какие алгоритмы поддерживаются вашей версий **sshd** можно, выполнив команду:

**ssh -Q cipher**

Думаю, **aes256-ctr** там точно будет.

Отключаем сжатие SSHv2-трафика

Это простое действие, нужное для защиты от класса атак на или некорректную/неточную обработку распаковки каких-то специфичных данных со стороны клиента, или на различные переполнения.

Сжатие – штука полезная и трафик экономит, кто же спорит. Просто в реальности в SSH вы или осуществляете администрирование – тогда объём отправляемых/принимаемых данных смешной и КПД сжатия с точки зрения использования будет совершенно не заметным, либо передаёте большие файлы при помощи SFTP – тогда, соответственно, эти файлы “досжать” при помощи встроенного в SSH сжатия обычно не получится, т.к. это или архивы, или мультимедийное содержимое.

Отключаем просто – добавляя в конфигурационный файл строку:

**Compression no**

В Cisco IOS сжатие в SSH не поддерживается официально, поэтому и отключать нечего.

Выбираем Message Authentication Code у SSHv2

Алгоритмы шифрования (за исключением варианта AEAD) не умеют проверять целостность данных, которые шифруют. Поэтому в нашем случае, когда используется AES256-CTR, надо озаботиться выбором MAC.

Вообще, по сути, это не сильно влияющая на безопасность настройка, т.к. компрометация способом “как-то на лету подменить хэш” чрезвычайно затруднена ограничениями по времени действия. Выцепить одиночный пакет, переделать содержимое, перешифровать и пересчитать целостность, а потом молнией вбросить обратно – это чрезвычайно сложная задача на уровне самой концепции этого действия.

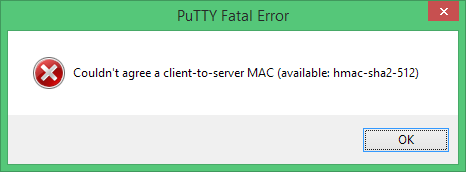
Поэтому, честно говоря, базового варианта **hmac-sha1-96** вполне достаточно.

Вы можете добавить поддержку более стойких вариантов – например, **hmac-sha2-512**, предварительно командой **ssh -Q mac** выяснив, что умеет ваш сервер, но будьте аккуратны – далеко не все клиенты поддерживают продвинутые варианты (опять же по причине не особо заметного влияния на безопасности). Упомянутый SHA-2/512, например, несмотря на то, что он [быстрый](https://www.atraining.ru/sha-2-fastest-sha-512/), поддерживается только последними версиями PuTTY.

Команда по части MAC, если всё же хотите вариант “безопаснее и не используя SHA-1” будет, например, такой:

**MACs hmac-sha2-512**

Вариант **hmac-sha2-256** поддерживается большинством существующих SSH-клиентов даже не самой последней версии. А вот **hmac-sha2-512** может не поддерживаться даже последней доступной (на момент публикации – 0.70) версией популярного клиента PuTTY:

[Ошибка PuTTY - невозможно согласовать MAC hmac-sha2-512  
(кликните для увеличения до 466 px на 172 px)](https://cdn.atraining.ru/i/ssh-secure-shell-armoring-2.png)

Поэтому выбирайте сами – SHA-512, при поддержке его на всём используемом оборудовании, будет лучшим выбором.

Но есть одна тонкость – режим использования хэша и шифрования.

По умолчанию SSH использует вариант, называемый Encrypt-and-MAC – к зашифрованным данным добавляется MAC, который считался от незашифрованных. В этом случае для проверки MAC надо вначале расшифровать принятый блок информации, чтобы получить plaintext, а после уже посчитать и сравнить хэши. Такой вариант делает возможными атаки на алгоритмы шифрования и получение “промежуточных” данных в случае компрометации целевой системы.

Лучшим вариантом с точки зрения безопасности будет Encrypt-Then-MAC. Почему? В случае, когда используется схема “хэш от уже зашифрованного”, первым делом проверяется целостность, и если что-то не так – данные сразу отбрасываются; никакой пробной расшифровки не происходит. Такие варианты MAC будут иметь суффикс **-etm**. С учётом этих моментов наша конфигурация будет выглядеть так:

**MACs hmac-sha2-512-etm@openssh.com,hmac-sha2-256-etm@openssh.com**

В Cisco IOS тип MAC будет задаваться так:

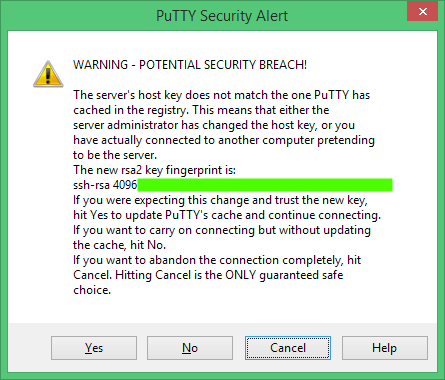
**atraining(config)#ip ssh server algorithm mac hmac-sha1**

У Cisco IOS выбор небогатый – **hmac-sha1** и **hmac-sha1-96**. Второй вариант с урезанием выхода SHA-1 со 160 бит до 96 (как в ipsec) нам не подойдёт, потому что считается он с той же скоростью (считать-то все равно надо обычный SHA-1), а экономия трафика – ну, кхм, вообще никакая.

MAC для подсчёта fingerprint’а ключа

В OpenSSH мы также можем задать алгоритм, которым будет высчитываться “отпечаток” – fingerprint ключа. По умолчанию используется MD5 – однако будет лучше явно указать, что для отображения и сравнения хэшей ключей лучше использовать SHA-2/256:

**FingerprintHash sha256**

[Окно с подтверждением обновления закэшированного ключа хоста  
(кликните для увеличения до 445 px на 380 px)](https://cdn.atraining.ru/i/ssh-secure-shell-armoring-3.png)

Теперь перейдём от криптографической части к сетевой.

Сетевые настройки SSHv2

Сетевых настроек будет много, но большинство из них тривиальны – “что использовать” и “как фильтровать трафик”, поэтому по разделу на каждую заводить смысла нет.

Блок будет выглядеть примерно так:

**Port 22 AddressFamily inet IgnoreRhosts yes UseDNS no ListenAddress x.x.x.x TCPKeepAlive yes #VerifyHostKeyDNS no #UseRoaming no**

Часть настроек понятна – например **Port 22** привязывает сервис SSH к указанному номеру порта, который можно при необходимости изменить – как минимум, чтобы боты-подбиратели-паролей не стучались, а **ListenAddress** явно указывает, на каких L3-адресах принимать запросы на подключение (ограничение актуально в случае нескольких адресов и/или сетевых интерфейсов, либо в сценарии “у хоста могут появляться на ходу новые сетевые интерфейсы, и это не значит, что на них надо ждать SSH-подключений”). Другие же настройки менее очевидны.

**AddressFamily inet** говорит о том, что мы будем ждать подключений только по IPv4. Если у вас используется IPv6 и подключения к SSH-серверу возможны по нему – добавьте **AddressFamily inet,inet6**.

**TCPKeepAlive yes** будет нужен, чтобы на сетевом уровне определять отключившихся пользователей и прекращать их сеансы. Выключение этого механизма, которое иногда рекомендуется “для экономии трафика” (слёзы, а не экономия) приведёт к ситуации, когда ssh не сможет в ряде случаев понять, что пользователь не просто неактивен, а уже никогда не сможет продолжить работу в данной сессии. Поэтому включаем.

**IgnoreRhosts yes** отключает древний механизм создания списка узлов (обычно в файле**.rhosts**), с которых возможен вход без аутентификации.

**UseDNS no** будет нужна, чтобы отключить проверку PTR-записи у подключающегося абонента – помимо того, что во внутренней сети, да и при подключении из внешних тоже наличие PTR совершенно необязательно, данная мера лишь замедляет подключение, а уровень безопасности не поднимает – максимум, что делает – пишет в журнал warning про “подключающийся не сказал своё настоящее FQDN-имя”. Хотя возможна ситуация, когда проверка эта включена, а DNS на узле не работает (например, указывает на неправильный IP-адрес DNS-сервера) – в этом случае возможен сценарий, что подключиться к узлу не получится. Но это совсем не “защитная мера” а, скорее, ещё одна причина отключать эту проверку, т.к. из-за неё, получается, растёт количество задействованных подсистем, а следовательно падает общая надёжность работы системы.

**VerifyHostKeyDNS no** отключает механизм [SSHFP](https://www.atraining.ru/sshfp-ssh-fingerprint-dns-resource-record/), который практически не используется в сетях, а вот запросы на тему “а есть ли такая экзотическая запись в вашем DNS” – отправляются. Конечно, если у вас в сети [SSHFP](https://www.atraining.ru/sshfp-ssh-fingerprint-dns-resource-record/) используется, то отключать его не надо. Это по своей логике клиентская настройка, но почему-то иногда встречаются мысли “включить её на сервере”. Я её добавил в этот список и закомментировал, чтобы подчеркнуть данный момент – не нужно в настройках сервера эту опцию указывать.

**UseRoaming no** выключает поддержку роуминга – экспериментального расширения OpenSSH, которое должно было обрабатывать сценарии вида “начал админить из одного места, потом перешёл в другое и продолжил оттуда”. По факту функционал этот оказался никому не нужен и никаких прорывных нововведений не добавил, а вот проблемы безопасности – вплоть до уязвимостей с PoC – принёс. Поэтому в явном виде отключаем. Как и предыдущий параметр – клиентский, т.е. здесь приведён потому что опять же в ряде гайдов пишется “выключите везде, и на клиенте, и на сервере”. Это не так, только на клиенте.

Ограничения SSH на процедуру подключения к сеансу

Процесс подключения к сеансу также надо проработать, т.к. что использование “необычных” способов подключения, что предоставление подключающемуся клиенту лишней информации, что расходование ресурсов сервера на поддержание множества “параллельных” запросов – не нужны.

Блок наших настроек про всё это будет выглядеть так:

**RhostsRSAAuthentication no PubkeyAuthentication no HostbaseAuthentication no ChallengeResponseAuthentication no KerberosAuthentication no PasswordAuthentication yes  
LoginGraceTime 15  
ClientAliveInterval 1800 ClientAliveCountMax 0  
MaxAuthTries 3 MaxSessions 1 PermitTunnel no  
MaxStartups 10:50:20 ShowPatchLevel no**

Разберёмся, что и как.

Блок из **RhostsRSAAuthentication no**, **PubkeyAuthentication no**, **HostbaseAuthentication no**, **ChallengeResponseAuthentication no**, **KerberosAuthentication no** отключает неиспользуемые методы аутентификации. Безусловно, если вы используете для входа на SSH-сервер, допустим, Kerberos, то отключать Kerberos не нужно. Но в типовом сценарии, когда вход осуществляется более распространёнными способами, лишнее надо в явном виде отключать.

Параметр **LoginGraceTime** говорит о том, сколько времени пользователю можно потратить на процедуру входа. По умолчанию – 2 минуты. Это очень много. Очень медленный пользователь должен быть, чтобы столько времени ему требовалось на вход. Поэтому этот параметр выставляется в 15 – за 15 секунд войти можно. Если надо дольше – можно увеличить, но разумно. Важнее то, что сервер будет быстрее “сбрасывать” сессии, которые начались, но ещё не завершили аутентификацию, и экономить ресурсы.

Аналог **LoginGraceTime** в Cisco IOS – это команда **atraining(config)#ip ssh time-out**, задающая максимальное время для процедуры входа. По умолчанию также 2 минуты и это также многовато и надо уменьшать. В случае с Cisco NX-OS это будет **atraining-nx(config)#ssh login-gracetime**.

Пара настроек **ClientAliveCountMax** и **ClientAliveInterval** будет нужна, чтобы определять, когда надо отключать неактивного клиента. **ClientAliveInterval** – время неактивности в секундах, через которое клиента отключат, а **ClientAliveCountMax** – количество попыток “разбудить” клиента. В нашем случае клиента отключат через полчаса неактивности.

Параметр **MaxAuthTries** говорит, через сколько попыток неудачной аутентификации (например, ввода неверного пароля) сессия будет отключена сервером. По умолчанию это 6, многовато. Трёх раз хватит.

Аналог **MaxAuthTries** в Cisco IOS – команда **atraining(config)#ip ssh authentication-retries**, с умолчанием в 3 попытки.

Параметр **MaxSessions** показывает, сколько сессий **внутри одного SSH-подключения** можно инициализировать. Это не ограничение на “параллельные сессии с одного хоста”! Это именно “поставил SSH-трубу до сервера и внутри неё мультиплексируешь много сессий с форвардингом данных”. Если вам такой сценарий не нужен – когда подключаетесь до сервера X и через него форвардите трафик вглубь сети, то хватит и единицы – а надо именно подключаться к конкретному серверу и его администрировать, то **MaxSessions 1**, да. Параметр **PermitTunnel no** будет довершать конфигурирование режима “ssh только для администрирования сервера, к которому подключаемся”.

**MaxStartups 10:50:20** – это WRED-подобный механизм, про семейство которых обсуждается на [курсах по QoS](https://www.atraining.ru/cisco-qos/) и логика его конфигурирования будет следующей – первое и последние значения – это стартовое количество подключений (речь только про подключения, которые не прошли аутентификацию), начиная с которых механизм начнёт работать и максимальное количество подключений, возможных вообще (в нашем случае механизм включится, когда к серверу будет 10 подключений, а 21е подключение будет технически невозможно), а средний параметр – это вероятность в процентах. У нас она 50, т.е. мы будем отказывать в половине случаев, когда количество “висящих на фазе аутентификации” сессий будет от 10 до 20.

Можно сделать и проще – например, **MaxStartups 10**, т.е. задать MaxStartups с одним параметром. Это просто укажет максимальное число параллельно аутентифицирующихся сессий.

Аналог **MaxStartups N** с одним параметром в Cisco IOS – это команда **atraining(config)#ip ssh maxstartups N**, где N – то же самое максимальное число параллельно аутентифицирующихся сессий.

Командой **ShowPatchLevel no** мы выключим публикацию детальной информации о версии SSH подключающемуся клиенту.

Теперь перейдём к блоку настроек, связанных с учётными записями.

Группы и пользователи в настройке SSH-сервера

Вполне очевидно, что нам надо как-то явно указать, кто к нам может подключаться и какие требования к этому кому-то будут выдвигаться.

Делаем:

**AllowUsers admin admin2 AllowGroups nixadmins DenyUsers nginx DenyGroups nginx PermitEmptyPasswords no PermitRootLogin no UsePrivilegeSeparation sandbox**

Тут тоже особо ничего удивительного нет, все параметры названы весьма точно – разве что остановлюсь на явном запрете входа через ssh для служебных учётных записей и групп (в моём примере **nginx**). Не ленитесь явно прописать это, такие учётные записи меняются исключительно редко, а подстраховка не помешает. Да, и не работайте под рутом, как бы тривиально это не звучало.

В варианте с Cisco IOS такие настройки локально на устройстве делать смысла не имеет, т.к. логичнее использовать AAA и перенаправлять запросы про аутентификацию и авторизацию через RADIUS/TACACS на какой-либо централизованный сервер, либо (в новых IOS) обращаться напрямую в LDAP-хранилище с запросами “есть ли такой пользователь”. Плодить локальные базы на каждом устройстве неудобно и небезопасно. Весь этот механизм не привязан к SSH, а является более общим способом перенаправления запросов на аутентификацию/авторизацию, поэтому здесь не описывается, чтобы статья осталась про SSH, а не “про всё, что может иметь отношение к SSH”.

Впрочем, по части паролей настройку сделать всё ж не помешает – команда

**atraining(config)#security password min-length N**

установит минимальную длину паролей на данном устройстве.

Настройка **UsePrivilegeSeparation sandbox** будет в явном виде говорить ssh, что для каждого входящего подключения будет создаваться отдельный процесс sshd с минимальными привилегиями – а после удачной авторизации уже будет создаваться новый, который и будет обрабатывать сеанс конкретного подключившегося пользователя. Это снижает потери при появлении новой уязвимости в механизме подключения к sshd – тот, кто будет эксплуатировать эту уязвимость, даже при захвате процесса ssh получит минимум прав в системе.

Краткий итог

Надеюсь, этот небольшой “чеклист” будет вам полезен. SSH крайне широко используется, поэтому предсказуемо настроенный и безопасный доступ через него – фундамент для надёжной сетевой инфраструктуры.

Удач!