Управление ключами в OpenSSH, Часть 1

Автор: Инга Захарова, arlantine@lnx.ru
Опубликовано: 8.6.2002

Перевод статьи Дэниэла Робинса (Daniel Robbins) OpenSSH key management, Part 1

Из этой серии статей вы узнаете, как действует RSA и DSA-аутентификация и как правильно настроить беспарольную аутентификацию. В первой статье цикла Дэниел Робинс уделяет внимание представлению протоколов аутентификации RSA и DSA и демонстрирует, как применять их для работы в сети.

Многие из нас используют замечательную программу OpenSSH (смотрите раздел Источники далее в этой статье) в качестве защищенной шифрованной замены архаичных telnet и rsh. Одной из самых захватывающих особенностей OpenSSH является ее способность аутентифицировать пользователей посредством протоколов аутентификации RSA и DSA, в основе которых лежит пара комплементарных числовых «ключей». Одной из самых притягательных сторон RSA- и DSA-аутентификации является их потенциальная способность устанавливать соединения с удаленными системами *без указания пароля*. Поскольку это очень заманчиво, новые пользователи OpenSSH зачастую настраивают RSA/DSA второпях, в результате получая беспарольные входы, но при этом открывая огромную дыру.

Что такое RSA/DSA-аутентификация?

SSH, и в особенности OpenSSH (свободно распространяемая версия SSH) – просто невероятная утилита. Так же, как и в*telnet* или *rsh*, *ssh*-клиент может быть использован для входа на удаленную машину. Все, что требуется от этой удаленной машины – использовать *sshd*, серверный процесс *ssh*. Но в отличие от *telnet*, протокол *ssh* является безопасным. Он использует специальные алгоритмы для шифрования потоков данных, обеспечивая неприкосновенность и целостность потока данных, и выполняет аутентификацию надежным и безопасным способом.

Хотя *ssh* действительно замечательная программа, среди функций *ssh* существует определенная компонента, часто обделяемая вниманием и катастрофически неиспользуемая, или просто недопонятая. Эта компонента – система ключей аутентификации RSA/DSA для OpenSSH, являющаяся альтернативой стандартной системе аутентификации безопасных паролей, которая в OpenSSH используется по умолчанию.

Протоколы RSA и DSA-аутентификации для OpenSSH имеют в основе два специально созданных криптографических ключа, носящие названия *«личный ключ»* и *«публичный ключ»*. Преимущество использования этой основанной на ключах системы аутентификации в том, что в большинстве случаев они позволяют устанавливать безопасные соединения без необходимости набирать пароль вручную.

Хотя основанные на ключах протоколы аутентификации относительно безопасны, в случае, если пользователь воспринимает быстрый вход как всего лишь удобное приспособление и не полностью осознает ее прямую связь с безопасностью, то могут возникнуть проблемы. В данной статье мы внимательно рассмотрим как правильно использовать протоколы RSA- и DSA-аутентификации и при этом не подвергать собственную безопасность ненужному риску. В своей следующей статье я продемонстрирую вам, как использовать *ssh-agent* для кэширования дешифрованных личных ключей, и представлю вашему вниманию *keychain* – программу-надстройку над *ssh-agent* который обеспечивает целый ряд полезных преимуществ, не жертвуя при этом безопасностью. Если вы всю жизнь мечтали докопаться до сути самых крутых примочек, тогда продолжайте читать.

Как работают ключи RSA/DSA.

Вот общий обзор принципов работы ключей RSA/DSA. Начнем с гипотетического сценария, в котором мы хотим использовать RSA-аутентификацию, дабы позволить локальной рабочей станции под Linux (назовем ее *localbox*) открыть shell на удаленной машине *remotebox* нашего поставщика интернет услуг (ISP). И теперь, при попытке установить соединение с remotebox используя клиент *ssh*, мы получаем следующее уведомление:

```
% ssh drobbins@remotebox
drobbins@remotebox's password:
```

Вот пример того, как *ssh* по умолчанию управляет аутентификацией. А именно, она запрашивает пароль учетной записи*drobbins* на *remotebox*. Если мы наберем пароль для *remotebox*, *ssh* задействует собственный безопасный протокол аутентификации паролей, который передаст наш пароль дальше на *remotebox* для подтверждения. Однако в отличие от того, как это происходит в *telnet*, здесь наш пароль зашифрован, таким образом тому, кто прослушивает наше соединение, не удастся его перехватить. Далее *remotebox* произведет аутентификацию поступившего от нас пароля с собственной базой данных паролей, и в случае успешного отождествления нам будет позволено войти и *shell* машины *remotebox* выдаст сообщение с приветствием. И хотя метод аутентификации осуществляемый *ssh* по умолчанию вполне надежен, все же RSA/DSA-аутентификация раскрывает кое-какие дополнительные возможности.

Однако, в отличие от осуществляемой *ssh* надежной «парольной» аутентификации, RSA-аутентификация требует некоторой первоначальной настройки. Нам нужно только один раз выполнить пункты этой первоначальной настройки. После этого RSA-аутентификация между *localbox* и *remotebox* станет абсолютно беспроблемной процедурой. Чтобы настроить RSA-аутентификацию, нам прежде всего потребуется генерировать пару ключей. Один личный и один публичный. Эти два ключа обладают весьма интересными особенностями. Публичный ключ можно использовать для шифрования сообщений и только владелец личного ключа сможет это сообщение дешифровать. Публичный ключ используется только для *шифрования*, а личный ключ используется только для *расшифровки* сообщений, закодированных при помощи соответствующего публичного ключа. Протоколы RSA- (и DSA-) аутентификации используют специфические особенности парных ключей для осуществления безопасной аутентификации, где нет необходимости пересылать конфиденциальную информацию через сеть.

Для того чтобы RSA- или DSA-аутентификация заработала, выполняем один единственный пункт настройки. Копируем наш *публичный ключ* через сеть на *remotebox*. «Публичным» этот ключ назван не без оснований. Поскольку он может быть использован только для *шифрования* наших сообщений, не стоит особенно беспокоиться, если он попадет в чужие руки. Теперь, когда публичный ключ скопирован через сеть на *remotebox* и помещен в специальный файл (~/.ssh/authorized_keys), *sshd remotebox*'а сможет его обнаружить, и мы готовы использовать RSA-аутентификацию для входа на *remotebox*. Чтобы это сделать, мы просто как всегда набираем *ssh drobbins@remotebox* на консоли *localbox*. Однако теперь *ssh* сообщает *sshd remotebox*'а, что она хочет использовать протокол аутентификации RSA. А дальше происходит совсем интересное. *sshd remotebox*'а генерирует случайное число и зашифровывает его при помощи публичного ключа, скопированного нами ранее. Затем он отправляет это случайное число в зашифрованном виде обратно к *ssh* на *localbox*'е. В свою очередь *ssh* нашей машины использует свой личный ключ для расшифровки этого случайного числа, которое затем отправляет обратно на *remotebox* в качестве утверждения: «Видишь, у меня на *самом деле* есть соответствующий личный ключ. Я смогла успешно расшифровать твое сообщение!» В итоге *sshd* делает вывод, что нам можно позволить войти, поскольку у нас есть соответствующий личный ключ. Таким образом наличие у нас соответствующего личного ключа обеспечивает нам доступ к *remotebox*.

Два нюанса.

Существует два немаловажных нюанса, касающихся RSA/DSA-аутентификации. Первое – то, что нам в действительности понадобится генерировать всего лишь пару ключей. Далее мы можем скопировать публичный ключ на удаленные машины, к которым хотим получить доступ, и все они будут успешно аутентифицировать нас по нашему личному ключу. Другими словами, нам не понадобится своя пара ключей для *каждой* системы, к которой мы хотим получить доступ. Будет достаточно одной единственной пары.

Второй нюанс заключается в том, что ваш *личный ключ не должен попасть в чужие руки*. Личный ключ – единственная вещь, открывающая доступ к вашим удаленным системам, и любому использующему ваш личный ключ обеспечиваются точно такие же как у вас права. Мы не собираемся давать незнакомцам ключи от нашего дома, и точно так же нужно защитить свой личный ключ от несанкционированного использования. В мире битов и байтов это означает, что никто не должен иметь возможности прочесть или скопировать ваш личный ключ.

Конечно, разработчики *ssh* осознают важность личного ключа, и в *ssh* и генераторе ключей *ssh-keygen* предпринят ряд мер безопасности для того чтобы не нарушался режим использования личного ключа. Во-первых, *ssh* настроен таким образом, чтобы выводить большое предупредительное сообщение, если кто-либо кроме вас имеет доступ «чтение» к файлу личного ключа. Во-вторых, когда вы при помощи *ssh-keygen* создаете вашу пару ключей личный/публичный, *ssh-keygen* попросит вас ввести ключевую фразу. Если вы это делаете, то ваш личный ключ будет зашифрован с использованием этой ключевой фразы, таким образом даже если ключ будет похищен, он будет

абсолютно бесполезен в руках того, кто этой фразы не знает. Вооруженные этим знанием, давайте посмотрим как настроить *ssh* на использование протоколов RSA- и DSA-аутентификации.

ssh-keygen при «ближайшем рассмотрении».

Первый шаг настройки RSA-аутентификации начинается с генерирования пары ключей публичный/личный. RSA-аутентификация – оригинальная версия ключевой аутентификации *ssh*, поэтому RSA будет работать с любой версией OpenSSH, кроме того я рекомендую вам установить самую свежую доступную версию (на момент, когда была написана эта статья, это openssh-2.9_p2). Генерируйте пару ключей RSA как показано ниже:

```
% ssh-keygen
Generating public/private rsa1 key pair
Enter file in which to save the key (/home/drobbins/.ssh/identity): (нажмите ввод)
Enter passphrase (empty for no passphrase): (введите ключевую фразу)
Enter same passphrase again: (введите фразу еще раз)
Your identification has been saved in /home/drobbins/.ssh/identity.
Your public key has been saved in /home/drobbins/.ssh/identity.pub.
The key fingerprint is:
a4:e7:f2:39:a7:eb:fd:f8:39:f1:f1:7b:fe:48:a1:09 drobbins@localbox
```

Когда *ssh-keygen* запрашивает местонахождение ключа по умолчанию, нажимаем ввод в знак принятия указанного по умолчанию /home/drobbins/.ssh/identity. ssh-keygen будет хранить личный ключ под приведенным выше паролем, а публичный ключ будет храниться рядом с ним в файле с именем identity.pub.

Обратите внимание, что *ssh-keygen* советовал вам ввести ключевую фразу. Следуя совету вы ввели хорошую фразу (состоящую из семи или более труднопредсказуемых символов). После этого *ssh-keygen* зашифровал ваш личный ключ (~/.ssh/identity) используя эту фразу, таким образом ваш личный ключ будет бесполезен для того, кто этой фразы не знает.

Быстрый компромисс.

Если вы задаете ключевую фразу, это позволяет *ssh-keygen* защитить ваш личный ключ от злоумышленного использования, но также создает и некоторое неудобство. Теперь, каждый раз, когда вы пытаетесь, используя *ssh*установить соединение с учетной записью *drobbins@remotebox*, *ssh* просит вас ввести ключевую фразу, чтобы расшифровать ваш личный ключ и использовать его для RSA-аутентификации. Еще раз напоминаю, что мы не будем вводить пароль для учетной записи *drobbins* на *remotebox*, мы наберем ключевую фразу, необходимую для расшифровки личного ключа на локальной машине. Поскольку наш личный ключ зашифрован, *ssh-клиент* позаботится обо всем остальном. Хотя механизмы использования пароля для удаленного доступа и ключевой фразы в RSA абсолютно различны, на практике нас по-прежнему приглашают набрать в *ssh* «условную фразу».

```
# ssh drobbins@remotebox
Enter passphrase for key '/home/drobbins/.ssh/identity': (введите ключевую фразу)
Last login: Thu Jun 28 20:28:47 2001 from localbox.gentoo.org
Welcome to remotebox!
%
```

И вот именно здесь люди вступают на опасный путь «быстрого компромисса». В большинстве случаев люди создают незашифрованные личные ключи, поэтому им не приходится вводить пароль. Таким образом они просто вводят команду *ssh*, сразу же аутентифицируются посредством RSA (или DSA) и входят в систему.

```
# ssh drobbins@remotebox
Last login: Thu Jun 28 20:28:47 2001 from localbox.gentoo.org
Welcome to remotebox!
%
```

Несмотря на то, что так удобнее, вы не должны использовать подобный подход, не отдавая себе отчета какой удар по безопасности он наносит. При использовании незашифрованного личного ключа, если кто-нибудь когда-либо взломает ваш *localbox*, он автоматически получит доступ к *remotebox* и всем другим системам, которые были настроены с помощью публичного ключа.

Знаю, о чем вы подумали. Беспарольная аутентификация, несмотря на несколько повышенную степень риска, все же кажется воистину привлекательной. Я полностью с этим согласен. Но существует лучший выход! Оставайтесь со мной, и я покажу вам, как получить все преимущества беспарольной аутентификации, не подвергая риску безопасность вашего личного ключа. В своей следующей статье я покажу вам, как мастерски использовать *ssh-agent* (ту самую штуку, которая в первую очередь делает возможной безопасную беспарольную аутентификацию). А сейчас давайте приготовимся использовать *ssh-agent* для настройки RSA- и DSA-аутентификации. Вот вам пошаговые указания на этот счет.

Генерирование пары RSA-ключей.

Для того, чтобы настроить RSA-аутентификацию понадобится один раз выполнить действие по созданию пары ключей личный/публичный. Делаем это, набирая:

```
% ssh-keygen
```

Подтвердите месторасположение ключа по умолчанию (как правило, для публичного ключа это ~/.ssh/identity and ~/.ssh/identity.pub) в строке предложения и снабдите *ssh-keygen* надежной ключевой фразой. Когда *ssh-keygen* завершит процесс, у вас в наличии будет публичный ключ, а также зашифрованных с помощью ключевой фразы личный ключ.

Установка публичного ключа RSA.

Далее нам необходимо настроить удаленные системы, в которых работает *sshd*, чтобы использовать для аутентификации наш публичный RSA-ключ. Как правило, это осуществляется копированием публичного ключа в удаленную систему следующим образом:

```
% scp ~/.ssh/identity.pub drobbins@remotebox:
```

До тех пор, пока RSA-аутентификация не будет полностью настроена, строка приглашения будет требовать от нас ввести пароль для доступа к *remotebox*. Сделайте это. Теперь войдите в систему *remotebox* и добавьте публичный ключ в файл ~/.ssh/authorized_keys вот таким образом:

```
% ssh drobbins@remotebox
drobbins@remotebox's password: (введите пароль)
Last login: Thu Jun 28 20:28:47 2001 from localbox.gentoo.org

Welcome to remotebox!
% cat identity.pub >> ~/.ssh/authorized_keys
% exit
```

Теперь, когда RSA-аутентификация настроена, при попытке установить соединение с *remotebox* посредством *ssh*, в строке приглашения вас попросят ввести ключевую фразу RSA (вместо пароля).

```
% ssh drobbins@remotebox
Enter passphrase for key '/home/drobbins/.ssh/identity':
```

Ура, настройка RSA-аутентификации завершена! Если вы не получили приглашения ввести ключевую фразу, вам следует кое-что проверить. Прежде всего попытайтесь войти в систему, набирая *ssh -1 drobbins@remotebox*. Таким образом вы укажете *ssh* использовать только версию 1 ssh-протокола, вам

может это потребоваться, если по какой-либо причине удаленная система по умолчанию настроена использовать DSA-аутентификацию. Если это не работает, убедитесь, что у вас нет строки, которая считывает *запрет RSA-аутентификации* из вашего файла /etc/ssh/ssh_config. Если такая есть, отметьте ее, поставив в начале «#». В противном случае попытайтесь связаться с администратором системы *remotebox* и удостоверьтесь, что RSA-аутентификация разрешена к использованию на выходе и соответствующие настройки имеются в файле /etc/ssh/sshd_config.

Генерирование DSA-ключей.

Если RSA-ключи используются протоколом *ssh* версии 1, то DSA-ключи используются для 2-го уровня и обновленной версии протокола. Любая из современных версий OpenSSH должна поддерживать работу как RSA так и DSA ключей. Генерирование DSA-ключей посредством *ssh-keygen* для OpenSSH можно осуществить аналогично той же процедуре в RSA следующим образом:

```
% ssh-keygen -t dsa
```

И опять мы получим приглашение ввести ключевую фразу. Введите какую-нибудь понадежнее. Мы также получим приглашение указать место, куда будут сохранены наши DSA-ключи. Обычно предлагаемый по умолчанию файл ~/.ssh/id_dsa and ~/.ssh/id_dsa.pub вполне подойдет. Генерирование ключей DSA завершено, и теперь самое время установить публичный DSA-ключ в удаленной системе.

Установка публичного DSA-ключа.

И снова мы сталкиваемся с тем, что установка публичного DSA-ключа практически идентична такой же процедуре в RSA. Для DSA мы скопируем наш файл ~/.ssh/id_dsa.pub в удаленную систему *remotebox* и добавим его в файл ~/.ssh/authorized_keys2 в этой системе. Обратите внимание, что этот файл имеет иное имя, чем файл authorized_keys в RSA. По окончании настройки мы должны получить возможность входить в систему *remotebox*, набирая ключевую фразу для личного DSA-ключа, вместо того чтобы вводить текущий пароль для входа в *remotebox*.

В следующий раз.

Теперь у вас имеется работающая RSA- или DSA-аутентификация, но вам все еще приходится вводить ключевую фразу для каждого нового соединения. В моей следующей статье вы увидите, как использовать *ssh-agent* – действительно классную систему, позволяющую устанавливать соединения без предоставления пароля, но при этом также позволяющую хранить наши личные ключи зашифрованными на диске. Я также представлю вашему вниманию *keychain*– весьма полезного клиента *ssh-agent*'а, который делает ssh-agent еще более надежным, удобным и прикольным. А пока ознакомьтесь с полезными источниками, приведенными ниже, дабы войти в курс дела.

Источники

- Непременно посетите домашнюю страницу разработчиков OpenSSH.
- Не обойдите вниманием OpenSSH source tarballs and RPMs.
- Просмотрите OpenSSH FAQ.
- PuTTY– превосходный клиент *ssh* для машин под Windows.
- Возможно вы найдете полезной книгу *O'Reilly's SSH, The Secure Shell: The Definitive Guide*. Сайт этого авторасодержит информацию о книге, FAQ, новости и обновления.
- Прочтите Addressing security issues in Linux на *developerWorks* для получения общего представления о шифровании данных и по многим другим вопросам, связанным с безопасностью.
- Просмотрите дополнительные материалы по Linux на *developerWorks*.
- Просмотрите дополнительные материалы Open source на *developerWorks*.

Управление ключами в OpenSSH, Часть 2

Автор: Инга Захарова, arlantine@lnx.ru
Опубликовано: 11.6.2002

Перевод статьи Дэниэла Робинса (Daniel Robbins) OpenSSH key management, Part 2

Многие разработчики используют замечательную программу OpenSSH в качестве защищенной шифрованной замены архаичных команд *telnet* и *rsh*. Одной из самых захватывающих особенностей OpenSSH является ее способность аутентифицировать пользователей посредством протоколов аутентификации RSA и DSA, в основе которых лежит пара комплементарных числовых «ключей». Одной из самых притягательных сторон RSA- и DSA-аутентификации является их потенциальная способность устанавливать соединения с удаленными системами *без указания пароля*. В этой второй статье Дэниел представляет вашему вниманию *ssh-agent* (кэш личного ключа) и *keychain* – специальный скрипт bash, разработанный для того, чтобы сделать основанную на ключах аутентификацию исключительно удобной и маневренной.

Знакомство с ssh-agent.

Включенный в дистрибутив OpenSSH *ssh-agent* – специальная программа, разработанная для того, чтобы сделать работу с ключами RSA и DSA приятной и безопасной (смотрите Первую часть этого выпуска для ознакомления с RSA- и DSA-аутентификацией). В отличие от *ssh*, *ssh-agent* – это постоянно работающий демон, созданный исключительно для кэширования ваших дешифрованных личных ключей.

В *ssh* включены встроенные средства поддержки, позволяющие ему общаться с *ssh-agent*'ом, а *ssh-agent*'у – получать ваши дешифрованные личные ключи не приглашая вас вводить пароль при установлении каждого нового соединения. При работе с *ssh-agent*'ом вы просто используете *ssh-add*, чтобы добавить ваш личный ключ в кэш *ssh-agent*'а. Это делается один раз; после использования *ssh-add*, *ssh* будет извлекать ваш личный ключ из *ssh-agent*'а, вместо того чтобы выдавать вам приглашение ввести ключевую фразу.

Использование *ssh-agent*.

Давайте рассмотрим, как работает вся система кэширования *ssh-agent*'а. Когда запускается *ssh-agent*, то перед тем как отделиться от shell и продолжить фоновую работу, он выдает на гора несколько важных переменных среды. Вот пример того, что может вывести *ssh-agent* при запуске:

```
% ssh-agent
SSH_AUTH_SOCK=/tmp/ssh-XX4LkMJS/agent.26916; export SSH_AUTH_SOCK;
SSH_AGENT_PID=26917; export SSH_AGENT_PID;
echo Agent pid 26917;
```

Как вы видите, выводимая *ssh-agent*'ом информация в действительности представляет собой серию команд bash; если эти команды будут выполнены, то они установят две переменные среды: SSH_AUTH_SOCK и SSH_AGENT_PID. Благодаря включенным в перечень командам экспорта доступ к этим переменным среды могут получить любые дополнительные команды, запущенные позднее. В общем, все будет происходить именно так в том случае, *если* shell действительно вычислит данные в строках, но в данном случае они просто выведены в stdout. Чтобы исправить ситуацию мы можем вызвать *ssh-agent* нижеприведенным способом:

```
eval `ssh-agent`
```

Эта команда указывает bash запустить *ssh-agent* и вычислить выведенные им данные. Вызванные таким образом (с back-quotes, а не обычными едиными квотами) переменные SSH_AGENT_PID и

SSH_AUTH_SOCK будут установлены и экспортированы вашей shell, что сделает эти переменные доступными для любых новых процессов, которые вы, возможно, запустите в процессе работы.

Лучший способ запустить *ssh-agent* – добавить строку в самый верх вашего файла ~/.bash_profile. В этом случае все программы, запускаемые в вашем login shell, будут видеть переменные среды, смогут определить местонахождение *ssh-agent*'а и, если понадобится, запросить у него ключи. Исключительно важной переменной среды является SSH_AUTH_SOCK. SSH_AUTH_SOCK содержит маршрут к UNIX domain socket, который *ssh* и *scp* могут использовать для установления диалога с *ssh-agent*'ом.

Использование ssh-add.

Но, конечно же, при запуске *ssh-agent*'а его кэш не содержит дешифрованных личных ключей. Прежде, чем мы сможем реально использовать *ssh-agent*, нам необходимо добавить наш личный ключ(и) в кэш *ssh-agent*'а при помощи команды *ssh-add*. В нижеприведенном примере я использую команду *ssh-add* чтобы добавить мой личный RSA-ключ ~/.ssh/identity в кэш *ssh-agent*.

```
# ssh-add ~/.ssh/identity
Need passphrase for /home/drobbins/.ssh/identity
Enter passphrase for /home/drobbins/.ssh/identity
(введите ключевую фразу)
```

Как вы видите, *ssh-add* запросила у меня ключевую фразу для того чтобы расшифровать личный ключ, который после этого будет готов к использованию и будет храниться в кэше *ssh-agent*. Теперь, когда вы использовали команду *ssh-add*чтобы добавить ваш личный ключ (или ключи) в кэш *ssh-agent*, а в вашей текущей shell определена переменная SSH_AUTH_SOCK (а она должна быть определена, если вы запускали *ssh-agent* из ~/.bash_profile), вы можете использовать *scp* и *ssh* для установления соединений с удаленными системами без указания вашей ключевой фразы.

Ограничения ssh-agent.

ssh-agent – действительно классная штука, но его настройки по умолчанию по-прежнему имеют некоторые небольшие неудобства. Давайте их рассмотрим.

В одном случае, связанном с *eval* `*ssh-agent*` в ~/.bash_profile, для каждой сессии запускается новая копия *ssh-agent*, и это означает не только создание дополнительных тэгов, но и то, что вам придется использовать *ssh-add* для добавления личного ключа в каждой новой копии *ssh-agent*. Не велика проблема, если вы открываете единичный терминал или консоль в вашей системе, но большинство из нас открывает сразу несколько терминалов и вынуждены вводить ключевую фразу каждый раз при открытии новой консоли. Формально нет причины, по которой мы непременно должны так поступать, в то время как однократного подобного действия в *ssh-agent* должно быть вполне достаточно.

Другая проблема, связанная с настройками *ssh-agent* по умолчанию, состоит в том, что он не совместим с cron jobs. После того как cron jobs запускаются процессом cron, они не наследуют из своей среды переменную SSH_AUTH_SOCK, и поэтому не знают ни того, что процесс *ssh-agent* запущен, ни как с ним связаться. Как оказалось, эта проблема тоже решаема.

Ввод keychain.

Для решения этих проблем я написал удобный основанный на bash клиент для *ssh-agent*, называемый *keychain*. Особенным *keychain* делает тот факт, что он позволяет вам использовать отдельный процесс *ssh-agent* для каждой системы, а не для каждой сессии. Это означает, что вам нужно будет один раз использовать *ssh-add* в отношении каждого личного ключа, и все. Как мы в последствии увидим, *keychain* способствует оптимизации процесса команды *ssh-add* уже одними даже своими попытками добавить в работающий кэш *ssh-agent* личные ключи, которых там еще нет.

Вот краткий обзор принципов работы *keychain*. Когда он запускается из файла ~/.bash_profile, он прежде всего проверит, запустился или нет *ssh-agent*. Если нет, то он запустит *ssh-agent* и запишет немаловажные переменные SSH_AUTH_SOCK и SSH_AGENT_PID в файл ~/.ssh-agent для сохранности и возможности дальнейшего использования. Здесь приведен лучший способ

запуска *keychain*; как при использовании доброго старого *ssh-agent* выполним необходимые настройки в ~/.bash_profile:

```
#!/bin/bash
#example ~/.bash_profile file
/usr/bin/keychain ~/.ssh/id_rsa
#redirect ~/.ssh-agent output to /dev/null to zap the annoying
#"Agent PID" message
source ~/.ssh-agent > /dev/null
```

Как вы могли заметить, для *keychain* мы используем в качестве исходного файл source the ~/.ssh-agent, вместо того чтобы вычислять выведенные данные, как при непосредственном использовании *ssh-agent*. Хотя результат такой же: чрезвычайно важная переменная SSH_AUTH_SOCK определена, *ssh-agent* запущен и готов к работе. А поскольку SSH_AUTH_SOCK записана в файл ~/.ssh-agent, то скрипты нашего собственного shell и cron jobs могут без труда связаться с *ssh-agent* просто используя информацию из файла ~/.ssh-agent. Сам keychain также использует преимущества этого файла. Вспомните, что при запуске keychain проверяет, запущен ли *ssh-agent*. Если запущен, то тогда keychain воспользуется файлом ~/.ssh-agent чтобы запросить точные установочные параметры переменной SSH_AUTH_SOCK, это позволит ей использовать уже действующую agent, а не открывать новую. *keychain* запустит новый процесс ssh-agent только в том случае, если файл ~/.ssh-agent является устаревшим (ссылается на уже не существующий *ssh-agent*) или если сам этот файл не существует.

Установка keychain.

Установить *keychain* просто. Сначала отправьтесь на keychain project page и скачайте из архива самую свежую версию*keychain*. Затем установите его нижеследующим образом:

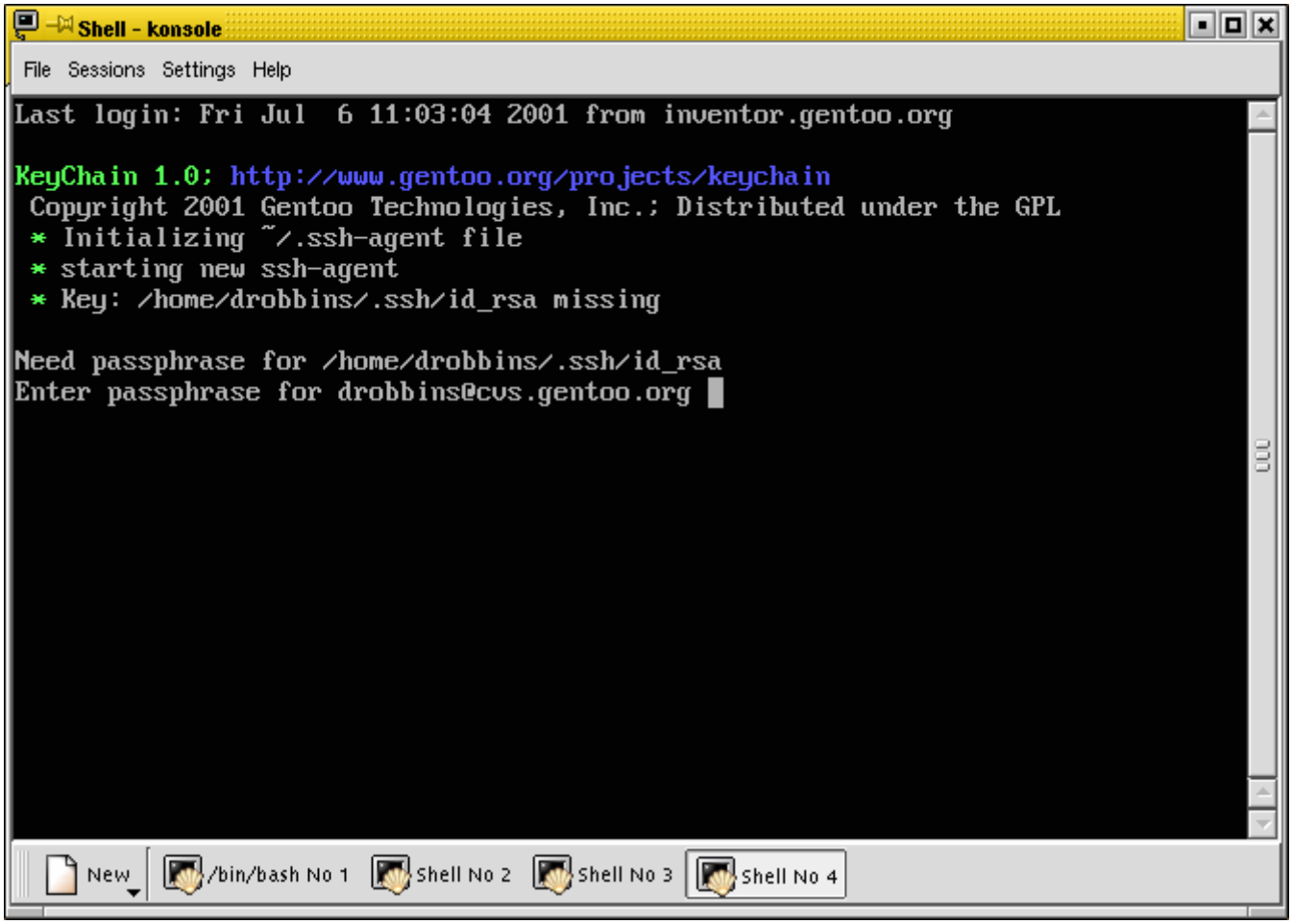
```
# tar xzvf keychain-1.0.tar.gz
# cd keychain-1.0
# install -m0755 keychain /usr/bin
```

Теперь когда *keychain* находится в вашей директории /usr/bin/, добавьте его в ваш файл ~/.bash_profile, указывая в качестве аргументов путь к вашим личным ключам. Вот неплохой образец ~/.bash_profile с действующим *keychain*(keychain-enabled ~/.bash_profile):
Пример ~/.bash_profile с действующим keychain

```
#!/bin/bash
# в этой следующей стороке мы запускаем keychain
# и указываем ему те личные ключи, которые он должен будет кэшировать
/usr/bin/keychain ~/.ssh/id_rsa ~/.ssh/id_dsa
source ~/.ssh-agent > /dev/null
#sourcing ~/.bashrc is a good thing
source ~/.bashrc
```

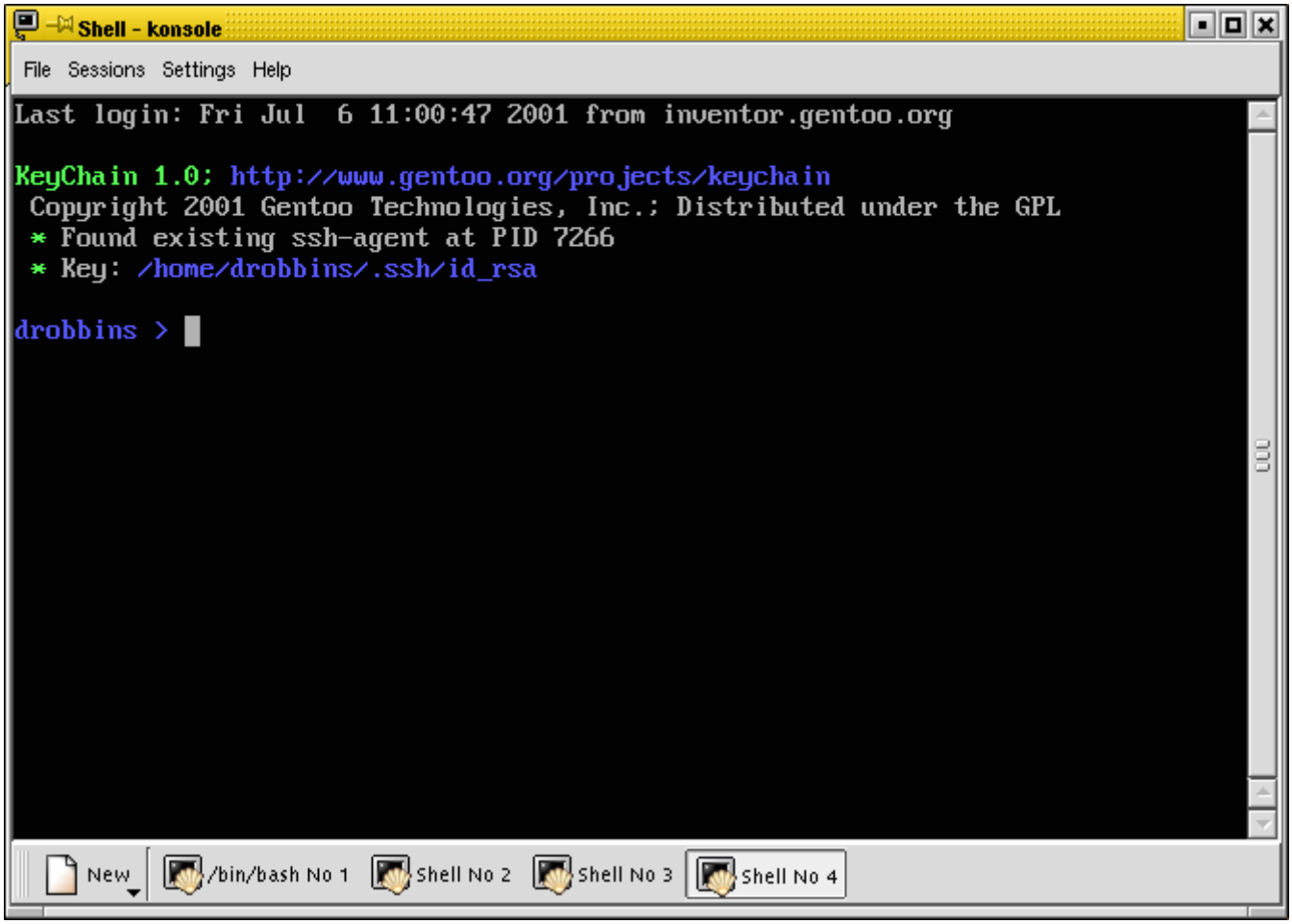
Keychain в действии.

Теперь вы настроили ~/.bash_profile вызывать *keychain* при каждом входе в систему, выходе из нее и возвращении. Когда это будет происходить, *keychain* будет запускать *ssh-agent* и записывать установочные характеристики переменной среды *agent*'а в файл ~/.ssh-agent, а затем пригласит вас ввести ключевые фразы для каждого личного ключа, указанного в командной строке keychain в файле ~/.bash_profile:



Keychain запускается в первый раз.

После того как вы введете ваши ключевые фразы, ваши личные ключи будут кэшированы и *keychain* сам свернется. Затем будет задействован ~/.ssh-agent, который инициализирует вашу сессию с *ssh-agent*. Теперь, если вы выйдете из системы и войдете вновь, то увидите, что *keychain* обнаружит уже действующий процесс *ssh-agent* (он не завершается при вашем выходе из системы). В дополнение ко всему *keychain* подтвердит наличие в кэш *ssh-agent*'а уже указанных вами личных ключей. В противном случае вы получите приглашение вести соответствующие ключевые фразы, хотя если все будет в порядке, то действующий *ssh-agent* должен по-прежнему содержать добавленные ранее личные ключи (это означает, что вас не попросят ввести пароль):



Keychain обнаруживает действующий ssh-agent

Мои поздравления: вы только что вошли в систему и теперь сможете использовать *ssh* и *scp* для удаленных систем. Вам не придется сразу же после входа применять *ssh-add* и ни *ssh* ни *scp* не попросят вас ввести ключевую фразу. Фактически, пока работает ваш исходный процесс *ssh-agent*'а, вы сможете входить в систему и устанавливать соединения *ssh* без введения пароля. И вполне вероятно, что процесс *ssh-agent*'а будет продолжать работать до перезагрузки машины. Учитывая тот факт, что вы вероятнее всего установили эти программы в системе Linux, то, возможно, пароль вам не понадобится в течение нескольких месяцев! Добро пожаловать в мир защищенных беспарольных соединений, использующих RSA- и DSA-аутентификацию.

Продолжайте в том же духе и создайте несколько новых сессий и вы заметите, что *keychain* каждый раз будет делать*«привязку»* к одному и тому же процессу *ssh-agent*. Не забудьте, что вы можете так же *«привязать»* к уже запущенному процессу *ssh-agent* свои скрипты и cron jobs. Чтобы пользоваться командами *ssh* и *scp* из скриптов shell и cron jobs, для начала просто убедитесь, что они используют в качестве источника информации в первую очередь ваш файл ~/.ssh-agent:

```
source ~/.ssh-agent
```

Тогда любые последующие команды *ssh* или *scp* смогут обнаружить уже работающий *ssh-agent* и установить защищенное беспарольное соединение так же, как вы делаете это из shell.

Опции keychain.

После того как вы настроили и запустили *keychain* потрудитесь вызвать *keychain --help* дабы ознакомиться со всеми опциями командной строки *keychain*. Мы сейчас рассмотрим одну особенную: опцию *--clear*.

Если вы помните, в Части 1 я объяснял, что использование незашифрованных личных ключей чревато, поскольку дает возможность кому-нибудь украсть ваш личный ключ и использовать его для доступа к вашим удаленным учетным записям из любой системы безо всякого пароля. Ну,

хотя *keychain* и не является уязвимой с этой точки зрения (поскольку вы используете зашифрованные личные ключи), все-таки в ее работе есть слабое место, непосредственно связанное с тем, что *keychain* позволяет так легко «делать привязку#187; к постоянно работающему процессу *ssh-agent*. Я думал, что произойдет, если какой-нибудь взломщик сможет каким-то образом вычислить мой пароль или ключевую фразу и войдет в мою локальную систему? Если он сможет каким-либо образом войти под моим именем, *keychain* тут же откроет ему доступ к моим дешифрованным личным ключам, что позволит взломщику без труда получить доступ ко всем остальным моим учетным записям.

Но перед тем как я продолжу, давайте рассмотрим эту угрозу безопасности в перспективе. Если какой-нибудь злоумышленник сможет каким-то образом войти в систему под моим именем, *keychain* конечно же позволит ему получить доступ к моим удаленным учетным записям. Хотя даже в этом случае взломщику будет очень сложно украсть мои дешифрованные личные ключи, если они будут по-прежнему зашифрованы на диске. Также для получения доступа к моим личным ключам пользователю потребуется действительно *войти в систему* под моим именем, а не просто получить возможность читать файлы в моей директории. Таким образом провести *ssh-agent* будет куда сложнее, чем просто украсть незашифрованные личные ключи, для чего взломщику, независимо от того, под чьим именем он вошел, потребуется всего лишь получить доступ к моим файлам в директории ~/.ssh. Тем не менее, если взломщику удастся войти в систему под моим именем, то используя мои дешифрованные личные ключи он сможет причинить немало вреда. Так что если вы используете *keychain* на сервере, в систему которого вы входите не слишком часто или который не слишком активно проверяете на наличие дыр в системе безопасности, то подумайте об использовании опции *--clear*дабы обеспечить дополнительный уровень защиты.

Опция *--clear* дает вам возможность указать для *keychain*, что она должна рассматривать каждое новое подключение к вашей учетной записи как потенциальную дыру в защите, до тех пор пока обратное не будет доказано. При запуске *keychain* с опцией *-clear*, то как только вы входите в систему, прежде чем приступить к выполнению своих обычных обязанностей, *keychain* немедленно сбрасывает все ваши личные ключи из кэша *ssh-agent*. Таким образом, если вы взломщик, то *keychain* попросит вас ввести ключевые фразы, прежде чем предоставить доступ к вашим существующим установкам кэшируемых ключей. Хотя это и позволяет повысить степень защиты, но делает работу чуть более неудобной и очень похожей на работу непосредственно *ssh-agent*'ом без *keychain*. И в этом случае, как это чаще всего бывает, кто-то может сделать выбор в пользу большей безопасности, а кто-то в пользу удобства, но никак не вместе.

Несмотря на это использование *keychain* с опцией *--clear* все же имеет преимущества перед использованием просто *ssh-agent*. Помните, что когда вы используете *keychain --clear*, ваши скрипты и cron jobs могут устанавливать беспарольные соединения, поскольку ваши личные ключи сбрасываются при *входе* в систему, а не при *выходе* из нее. А поскольку выход из системы не создает потенциальных брешей в безопасности, то у keychain нет причин реагировать на него сбросом ключей из *ssh-agent*. Таким образом опция *--clear* является идеальным выбором для нечасто посещаемых серверов на которых от случая к случаю необходимо выполнить защищенное копирование (такие как серверы для резервных копий, firewall'ы и router'ы).

Мы закончили!

Теперь когда выпуск по управлению ключами в OpenSSH завершен, вы должны быть хорошо знакомы с ключами RSA и DSA и знать, как их использовать удобным и в то же время безопасным способом. Также не забудьте просмотреть нижеприведенные материалы.

Источники

- Прочтите Часть 1 из выпуска Дэниела по управлению ключами в OpenSSH.
- Посетите страницу home of OpenSSH development.
- Скачайте самую свежую версию *keychain*.
- найдите новейшие tarball'ы and RPM'ы от OpenSSH source.
- Просмотрите FAQ по OpenSSH.
- PuTTY – превосходная программа-клиент *ssh* для машин под Windows.

- Возможно, вы найдете полезной книгу O'Reilly's SSH, The Secure Shell: The Definitive Guide. Сайт этого авторасодержит информацию о книге, FAQ, новости и обновления.

- Просмотрите дополнительные материалы по Linux на *developerWorks*.

- Просмотрите дополнительные материалы по Open source на *developerWorks*.

Управление ключами в OpenSSH, Часть 3

Автор: Инга Захарова, arlantine@lnx.ru
Опубликовано: 15.6.2002

Перевод статьи Дэниэла Робинса (Daniel Robbins) OpenSSH key management, Part 3

В третьей статье данного выпуска Дэниел Робинс демонстрирует, как в целях повышения безопасности использовать преимущества переадресации соединения, установленного посредством OpenSSH agent. Он также поделится новейшими доработками, внесенными в скрипт keychain shell.

Многие из нас используют замечательную программу OpenSSH в качестве защищенной шифрованной замены архаичных команд *telnet* и *rsh*. Одной из самых захватывающих особенностей OpenSSH является ее способность аутентифицировать пользователей посредством протоколов аутентификации RSA и DSA, в основе которых лежит пара комплементарных числовых «ключей». Одной из самых притягательных сторон RSA- и DSA-аутентификации является их потенциальная способность устанавливать соединения с удаленными системами без указания пароля. Для получения дополнительной базовой информации просмотрите предыдущие статьи из этой серии по Управлению ключами в OpenSSH, на темы **RSA/DSA-аутентификации** (Часть 1) и **ssh-agent и keychain** (Часть 2) соответственно.

Поскольку Часть 2 была опубликована на *developerWorks* в сентябре 2001, а затем была упомянута на *Slashdot* и *Freshmeat* (ищите ссылки на эти сайты далее в статье в разделе Источники), многие стали использовать *keychain*, вследствие чего программа претерпела множество изменений. Я получил около 20 патчей высокого качества от разработчиков со всего мира. И многие из этих патчей я включил в состав *keychain*, и теперь это версия 1.8 (смотрите Источники). Я выражаю искреннюю благодарность всем, кто предоставил свои патчи, отчеты об ошибках, требования к функциям и собственную оценку.

Повышение безопасности ssh

В своей предыдущей статье я уделил некоторое время обсуждению преимуществ и выгод, которые сулит использование*ssh-agent*. Несколько дней спустя после того, как вторая статья появилась на *developerWorks*, я получил e-mail от Чарльза Карнея (Charles Karney) из Sarnoff Corporation, в котором он любезно сообщил мне о новых возможностях переадресации программы аутентификации agent в OpenSSH, с которыми мы сами в скором времени ознакомимся. В дополнение Чарльз подчеркнул тот факт, что использование *ssh-agent* на *ненадежных* машинах довольно опасно: если кому-то удастся получить права доступа root'а системы, то ваши дешифрованные ключи *могут быть извлечены* из *ssh-agent*. Даже если извлечение ключей трудновыполнимо, тем не менее, это входит в число профессиональных навыков взломщиков. А сам *факт возможности* кражи личных ключей означает, что мы в первую очередь должны принять меры для защиты от этой опасности.

Для разработки стратегии защиты наших личных ключей нам прежде придется отнести машины, к которым мы имеем доступ, к одной из двух категорий. Если какой-то конкретный хост хорошо защищен или изолирован, вряд ли удастся успешно воспользоваться доступом root для реализации собственных планов на этой машине – тогда такая машина будет считаться *надежным хостом*. Однако, если с машиной работают также многие другие люди, или у вас есть какие-либо сомнения в защищенности ее системы, то эта машина будет считаться *ненадежным хостом*. Чтобы защитить ваши личные ключи от возможности быть извлеченными, *ssh-agent* (а, следовательно, и *keychain*) никогда не должны запускаться с ненадежного хоста. Таким образом, даже если безопасность системы нарушена, то там, куда взломщик попадет в первую очередь, просто не будет по близости *ssh-agent* для извлечения ключей.

Однако, это создает проблему. Если вы не сможете запустить *ssh-agent* на ненадежных хостах, то как вы сможете устанавливать безопасные беспарольные соединения через *ssh* из этих систем? Это возможно только если вы будете использовать *ssh-agent* и *keychain* на *надежных* хостах, и при этом воспользуетесь новыми возможностями OpenSSH по *переадресации аутентификации* для того чтобы протянуть беспарольную аутентификацию до любого ненадежного хоста. в двух словах переадресация аутентификации действует таким образом, что позволяет удаленным сессиям *ssh*устанавливать соединения с *ssh-agent*'ом, работающим в надежной системе.

Переадресация agent'а аутентификации

Чтобы получить представление о том, как работает переадресация аутентификации, давайте прежде рассмотрим гипотетическую ситуацию, в которой у пользователя *drobbins* в наличии надежный лаптоп с именем *lappy*, надежный сервер, называемый *trustbox*, и еще две ненадежных системы, к которым он должен иметь доступ, названных *notrust1* и*notrust2*. В настоящий момент он использует *ssh-agent* в паре с *keychain* на всех четырех машинах, как показано ниже:

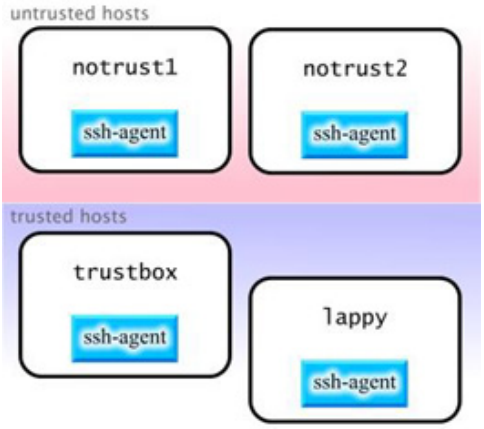


Иллюстрация 1. запуск ssh-agent на надежных и ненадежных машинах

Проблема данного подхода в том, что если кто-либо получает доступ root к системам *notrust1* или *notrust2*, то для этого человека становится безусловно возможным извлечь ключи из крайне уязвимого сейчас процесса *ssh-agent*. Чтобы исправить эту ситуацию *drobbins* прекращает работу *ssh-agent* и *keychain* на ненадежных хостах *notrust1* и *notrust2*. На самом деле из осторожности *drobbins* решает использовать *ssh-agent* и *keychain* исключительно на *lappy*. Это ограничивает «засвечивание» его личных ключей, тем самым защищая их от возможности быть украденными:

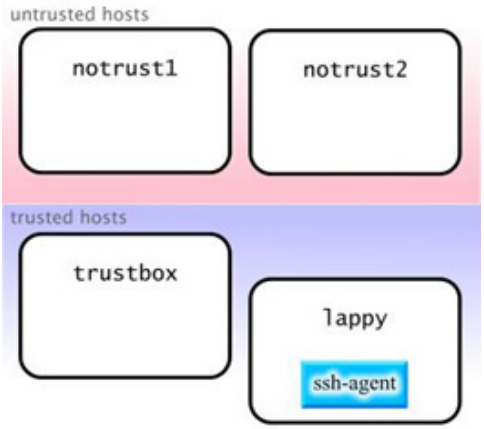


Иллюстрация 2. ssh-agent запускается только на lappy; более безопасный вариант настройки

Конечно, проблема данного подхода в том, что сейчас *drobbins* может установить беспарольное соединение только с*lappy*. Давайте посмотрим, как можно сделать возможной переадресацию аутентификации и обойти эту проблему.

Предположив, что все машины используют последние версии OpenSSH, мы можем обойти эту проблему, воспользовавшись переадресацией аутентификации. Переадресация аутентификации позволяет удаленному процессу*ssh* установить связь с *ssh-agent*, запускаемым на вашем локальном надежном компьютере – даже если версия используемого *ssh-agent* требует, что он должен быть запущен непременно с той машины, с которой устанавливается соединение посредством *ssh*. Как правило это позволяет вам запускать *agent* (и *keychain*) на отдельно взятой машине и означает, что соединение посредством *ssh*, работающего на данной машине (не важно, напрямую или нет) будет использовать ваш локальный *ssh-agent*.

Для того, чтобы сделать возможной переадресацию аутентификации, добавляем в файл */etc/ssh/ssh_config* систем *lappy*и *trustbox* нижеприведенную строку. Обратите внимание, что это файл настройки для *ssh* (*ssh_config*), а не для демона *ssh* – *sshd* (*sshd_config*):
Листинг 1. Добавьте эту строку в ваш файл /etc/ssh/ssh_config

```
ForwardAgent Yes
```

Теперь, чтобы воспользоваться бонусами переадресации аутентификации, *drobbins* может установить связь от *lappy* к *trustbox*, а затем от *trustbox* к *notrust1* без указания ключевой фразы ни для одного из соединений. Оба процесса *ssh*«подключаются» к *ssh-agent*, запущенному на lappy:
Листинг 2. Подсоединение lappy

```
$ ssh drobbins@trustbox
Last login: Wed Sep 26 13:42:08 2001 from lappy

Welcome to trustbox!
$ ssh drobbins@notrust1
Last login: Tue Sep 25 12:03:40 2001 from trustbox
Welcome to notrust1!
$
```

Если вы использовали подобную настройку, но обнаружили, что переадресация *agent*'а не работает, попробуйте использовать *ssh -A* вместо старой *ssh*, чтобы наверняка разрешить переадресацию аутентификации. Здесь приведена диаграмма процесса, происходящего «на заднем плане» в то время, как мы входим в систему *trustbox* and *notrust1*посредством описанной выше переадресации аутентификации:

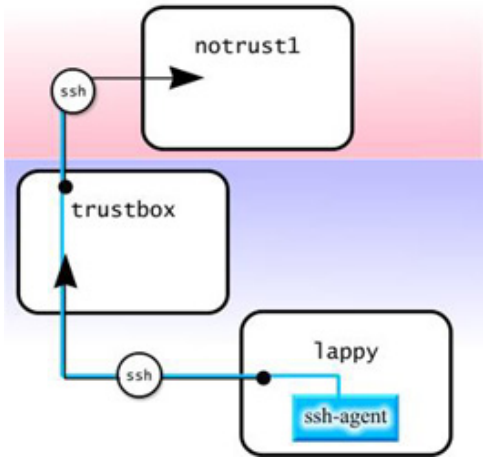


Иллюстрация 3. Переадресация agent'а в действии

Как вы видите, в то время, когда *ssh* установила соединение с *trustbox*, она продолжает сохранять связь с *ssh-agent*'ом, работающим на *lappy*. Когда было установлено соединение от *trustbox* к *notrust1*, то этот новый процесс *ssh* по-прежнему поддерживает связь посредством аутентификации с предыдущей *ssh*, увеличивая тем самым протяженность цепи. Получится ли продолжить эту цепь аутентификации дальше от *notrust1* к другим хостам – зависит от того, как настроен файл /etc/ssh/ssh_config системы *notrust1*. В том случае, если переадресация *agent*'а разрешена, все звенья цепи могут производить аутентификацию используя *ssh-agent*, работающий на надежном *lappy*.

Преимущества переадресации соединений agent'а

Переадресация *agent*'а предоставляет ряд преимуществ в сфере безопасности, не затронутых здесь. Дабы убедить меня, насколько значима переадресация соединений *agent*'а, Чарльз Карней (Charles Karney) поделился со мной, какие это три преимущества:

1. Личный ключ хранится только на надежной машине. Это предотвращает возможность того, что злоумышленник похитит ваш дешифрованный ключ с диска и попытается взломать шифр.

2. *ssh-agent* запускается только на надежной машине. Это не позволит взломщику сделать «снимок» памяти с удаленного процесса *ssh-agent*'а и извлечь ваши дешифрованные личные ключи из этого «снимка».

3. Так как вам приходится набирать ключевую фразу только на надежной машине, вы пресекаете поползновения со стороны любых регистраторов последовательности нажатия клавиш «стянуть» вашу ключевую фразу после того, как она уже введена.

Единственным недостатком переадресации соединений agent'а аутентификации является то, что он не решает такой проблемы как предоставление cron jobs возможности использовать блага RSA/DSA-аутентификации. Единственное решение данной проблемы – настроить все cron jobs, для которых необходимо использовать

RSA/DSA-аутентификацию таким образом, чтобы они работали с надежной машины в вашей локальной сети. В случае необходимости эти cron jobs смогут использовать *ssh* для установления соединений с удаленными системами для автоматического создания резервных копий, синхронизации файлов и тому подобных операций.

Теперь, когда мы рассмотрели переадресацию соединений agent'a аутентификации, давайте вернемся к доработкам, внесенным в скрипт *keychain* в последнее время.

Усовершенствование функций keychain.

Спасибо пользователям за предоставленные патчи, в первоначальный вариант *keychain* было внесено много замечательных усовершенствований. Например, вы переименовали созданный *keychain* файл ~/.ssh-agent, теперь он называется ~/.ssh-agent-[имя хоста], поэтому *keychain* работает с домашними директориями home, смонтированными в сетевой файловой системе (NFS), доступ к которым можно получить с некоторых физических хостов. В качестве дополнения к файлу ~/.ssh-agent-[имя хоста] теперь есть файл ~/.ssh-agent-csh-[имя хоста], который может служить источником информации для *csh*-совместимых командных интерпретаторов. И в довершение, добавлена новая опция *-nocolor*, таким образом, если вы используете терминал, несовместимый с vt100, то вы можете отменить функции цветности.

Исправления по поводу совместимости с командными интерпретаторами.

Хотя усовершенствования функций являются достаточно важными, подавляющее большинство исправлений касалось вопросов *совместимости с командными интерпретаторами*. Вы видите, что, тогда как для работы с keychain 1.0 требовался *bash*, более поздние версии были изменены так, чтобы работать с любым *sh*-совместимым командным интерпретатором. Эти изменения позволяют *keychain* прекрасно работать практически в любой системе UNIX, включая Linux, BSD, Solaris, IRIX, и AIX , равно как и прочие платформы UNIX. Хотя переход к *sh*- и UNIX-совместимости вообще происходил неравномерно (скачками), в то же время он является великолепным способом обучения и получения практических навыков. Создание одного единственного скрипта, который работает со всеми этими платформами, вообще непростая задача, по большому счету вследствие того, что я попросту не имею доступа к большинству этих операционных систем! К счастью, пользователи *keychain* со всего земного шара имеют, и многие из них оказали огромное содействие в выявлении проблем совместимости и предоставили патчи для их решения.

В действительности было два вида проблем совместимости, подлежавших исправлению. Прежде всего я должен был убедиться, что *keychain* использует только те встроенные функции (built-ins), выражения и операторы, которые полностью поддерживаются всеми интерпретаторами *sh*, в том числе всеми популярными свободными и коммерческими командными интерпретаторами *sh* для UNIX, *zsh* (в *sh*-совместимом режиме) и *bash* версий 1 и 2. Вот некоторые из предоставленных пользователями исправлений касательно совместимости с командными интерпретаторами, которые были внесены в первоначальную версию *keychain*:

Поскольку более старые командные интерпретаторы *sh* не поддерживают использование символа '~' в качестве ссылки на директорию home пользователя, строки, в которых использовался символ '~' были заменены на *$HOME*:
Листинг 3. Изменение на $HOME

```
hostname=`uname -n`pidf=${HOME}/.ssh-agent-${hostname}
cshpidf=${HOME}/.ssh-agent-csh-${hostname}
```

Далее, все ссылки на *source* были заменены на '.' Для того чтобы обеспечить совместимость для приверженцев NetBSD*/bin/sh*, которая вообще не поддерживает использование команды *source*:
Листинг 4. Подгонка под NetBSD

```
if [ -f $pidf ]
then
 . $pidf
else
SSH_AGENT_PID="NULL"
fi
```

По ходу дела я также внес несколько исправлений для улучшения рабочих характеристик. Один сообразительный создатель скриптов для командного интерпретатора сообщил мне, что вместо того, чтобы для создания файла набирать *touch foo*, вы можете просто поступить так:
Листинг 5. Создание файлов

```
> foo
```

Полагаясь скорее на встроенный синтаксис командного интерпретатора, нежели на использование привлеченных бинарных кодов, функция *fork()* не используется, и эффективность скрипта немного повышается. Предположительно команда > *foo* должна работать с любым *sh*-совместимым командным интерпретатором, однако, она, кажется, не будет поддерживаться *ash*. Для большинства людей это не будет являться проблемой, поскольку *ash* скорее является командным интерпретатором диска для аварийного восстановления системы, чем той программой, которую люди используют для обычной работы изо дня в день.

Особенности запуска на разных платформах

Для получения скрипта, работающего сразу под многими операционными системами UNIX, требуется больше, чем простое сращивание синтаксисов *sh*. Помните, что в большинстве своем скрипты еще и вызывают внешние команды, такие как *grep*, *awk*, *ps*, и другие, а все эти команды должны вызываться способом, максимально приближенным к стандартному. Например, *echo*, входящий в большинство версий UNIX, распознает опцию *-e*, а Solaris – нет, если применить *-e* он просто отпечатывает ее в stdout. Таким образом,для Solaris *keychain* теперь автоматически распознает, работает ли опция *echo -e*:
Листинг 6. Распознавание Solaris'a

```
if [ -z "`echo -e`" ]
then
 E="-e"
fi
```

Вверху – значение *E* установлено равным *-e* если поддерживаются Escape-последовательности. Теперь команда *echo*может быть вызвана следующим образом:
Листинг 7. Улучшенная echo

```
echo $E Usage: ${CYAN}${0}${OFF} [ ${GREEN}options${OFF} ] ${CYAN}sshkey${OFF} ...
```

При использовании *echo $E* вместо *echo -e*, опция *-e* в случае необходимости может быть оперативно разрешена или отменена.

pidof, ps

Наверное, для всех наиболее значительных поправок, касающихся совместимости, потребовалось внести изменения в способы, которыми *keychain* выявляет процессы *ssh-agent*, работающие в текущий момент. Раньше для этих целей я пользовался командой *pidof*, но ее пришлось убрать, поскольку в некоторых из систем нет *pidof*. На самом деле, *pidof* в любом случае не являлась наилучшим решением, поскольку она выводит список всех запущенных в системе процессов *ssh-agent*'а, невзирая на пользователя, в то время как нас в действительности интересуют процессы *ssh-agent*'а, принадлежащие текущему действующему UID.

Таким образом, вместо того, чтобы зависеть от команды *pidof*, мы переходим к конвейеризации данных выводимых *ps* в*grep* и *awk* для того чтобы извлечь необходимые id процессов. Эта поправка была предложена пользователем:
Листинг 8. Конвейеризация лучше pidof

```
mypids=`ps uxw | grep ssh-agent | grep -v grep | awk '{print $2}'`
```

Приведенная выше конвейерная обработка установит для переменной *mypids* значение, равное числу всех процессов*ssh-agent*, принадлежащих текущему пользователю. Команда *grep -v grep* является частью конвейерной обработки, обеспечивающей, чтобы процесс *grep ssh-agent* не стал частью списка PID.

В то время как в теории этот метод хорош, использование *ps* обнаружило целый ряд скрытых проблем, поскольку опции*ps* не имеют единых стандартов для различных систем BSD и производных от System V UNIX. Вот вам пример: в то время как *ps uxw* работает под Linux, она не работает под IRIX. А *ps -u username -f*, которая работает под Linux, IRIX, и Solaris, не работает в системе BSD, которая понимает только типичные для BSD опции *ps*. Чтобы разобраться с этой проблемой, прежде чем выполнить конвейерную обработку *ps*, *keychain* автоматически выявляет, с синтаксисом каких систем работает данная *ps*: BSD или System V:
Листинг 9. Определение BSD это, или System V

```
psopts="FAIL"
ps uxw >/dev/null 2>&1
if [ $? -eq 0 ]
then
psopts="uxw"
```

```
else
ps -u `whoami` -f >/dev/null 2>&1
if [ $? -eq 0 ]
then
psopts="-u `whoami` -f"
fi
fi
if [ "$psopts" = "FAIL" ]
then
echo $0: unable to use \"ps\" to scan for ssh-agent processes.
Report KeyChain version and echo system configuration to drobbins@gentoo.org.
exit 1
fi

mypids=`ps $psopts 2>/dev/null | grep \
 "[s]sh-agent" | awk '{print $2}'` > /dev/null 2>&1
```

Чтобы обеспечить возможность работы с *ps*-командами System V равно как с *ps*-командами BSD-типа, скрипт осуществляет «пробный прогон» *ps uxw*, отбрасывая все выводимые данные. Если код ошибки в этой команде равен нулю, тогда мы знаем, что *ps uxw* работает, и мы устанавливаем соответственное значение *psopts*. А если *ps uxw* выдало ненулевой код ошибки (указывая, что нам придется использовать опции BSD-типа), что мы осуществляем пробный прогон *ps -u `whoami` -f*, еще раз отбрасывая все выводимые данные. В данном случае, будем надеяться, мы выяснили, какой вариант ps мы сможем использовать: для BSD или для System V. Если по-прежнему не выяснили, то распечатываем код ошибки и выходим. Но весьма вероятно, что одна из двух команд *ps* все же будет работать, и в этом случае мы выполняем заключительную строку из приведенного выше лекала кода, конвейерной обработки нашей *ps*. Используя расширение переменной *$psopts* сразу после ps, мы сможем сообщить правильные опции команде *ps*.

Конвейерная обработка *ps* также содержит истинный трюк с *grep*, любезно присланный мне Гансом Петером Верне (Hans Peter Verne). Обратите внимание, что *grep -v grep* больше не является частью конвейерной обработки – он удален, а *grep "ssh-agent"* заменен на *grep "[s]sh-agent"*. Эта единственная команда *grep* делает то же самое, что и *grep ssh-agent | grep -v grep*; можете догадаться, почему?

Листинг 10. Ловкий фокус с grep

```
mypids=`ps $psopts 2>/dev/null | grep "[s]sh-agent" | awk '{print $2}'` > /dev/null 2>&1
```

Озадачены? Если вы решили, что *grep "ssh-agent"* и *grep "[s]sh-agent"* должны соответствовать одним и тем же строкам текста, то вы правы. Тогда почему они генерируют различающиеся результаты, когда им переводятся данные вывода *ps*? Вот как это работает: когда вы используете *grep "[s]sh-agent"*, вы вносите изменения в то, как именно команда *grep*появляется в списке процессов. Поступая таким образом вы не допускаете, чтобы *grep* соответствовала самой себе, в то время как строка *[s]sh-agent*'а не соответствует стандартному выражению *[s]sh-agent*. Ну не блестяще ли? Если вы все еще не въехали, поупражняйтесь с *grep* подольше и довольно скоро вы все поймете.

Вывод

В этой колонке приведены мои материалы по OpenSSH. Надеюсь, вы узнали достаточно, для того чтобы начать использовать OpenSSH для защиты своих систем. В следующем месяце в колонке *Common threads* выйдет продолжение выпуска «Advanced filesystem implementor's guide».

Источники

- Common threads: OpenSSH key management, Part 1 (*developerWorks*, Июль 2001) освещает RSA/DSA-аутентификацию (перевод).
- Common threads: OpenSSH key management, Part 2 (*developerWorks*, Сентябрь 2001) знакомит с ssh-agent и keychain (перевод).
- новейшая версия *keychain* находится на странице Gentoo Linux Keychain.
- Непременно посетите home of OpenSSH development, и найдите OpenSSH FAQ.
- Вы сможете скачать новейшие версии tarball'ов и RPM'ов OpenSSH с Openbsd.org.
- PuTTY - превосходный клиент ssh для Windows.

- В качестве вспомогательной можете использовать книгу «SSH, The Secure Shell: The Definitive Guide» (O'Reilly & Associates, 2001). Сайт автора содержит информацию о книге, FAQ, новости и обновления.
- Посетите Slashdot, где вы найдете "новости для зануд и тому подобный мусор".
- Найдите Freshmeat, там содержится список новых появляющихся версий пакетов от open source.
- Просмотрите дополнительные материалы по Linux на *developerWorks*.
- Просмотрите дополнительные материалы Open source на *developerWorks*.

Об авторе

Проживающий в Альбукерке, Нью-Мексико (Albuquerque, New Mexico) Дэниел Робинс является Президентом/Главным Администратором компании Gentoo Technologies, Inc., создателем Gentoo Linux – продвинутой версии Linux для PC и системы **Portage** - системы портов нового поколения для Linux. Он так же в качестве автора выпустил книги *Caldera OpenLinux Unleashed*, *SuSE Linux Unleashed*, and *Samba Unleashed* в сотрудничестве с издательством Macmillan books. Достаточно регулярно Дэниел стал общаться с компьютером на втором курсе, когда впервые занялся исследованием языка программирования Logo на предмет определения потенциально опасной дозы Pac Man. Возможно, это объясняет тот факт, что с тех самых пор он является ведущим художником-графиком в компании **SONY Electronic Publishing/Psygnosis**. Свободное время Дэниел проводит со своей женой Мэри (Mary) и маленькой дочерью Надассой (Hadassah). Вы можете связаться с ним по e-mail drobbins(at)gentoo.org.