実践で学ぶ

一歩進んだサーバー構築・運用術

written by 仙石 浩明

第9回 ssh（前編）

ssh（セキュア・シェル）は，単にセキュアなrsh(リモート・シェル)というだけでなく，さまざまな応用が効く大変便利なコマンドです。リモート・ホスト上で任意のコマンドを実行できます。今月号はsshの基本的な使い方を解説し，来月号でその応用方法を解説します。

私は道楽でgcd.orgというサイトを運営しています。運営費（OCNエコノミーの回線代）の足しにするべく，任意団体GCDを作って会員にサービスを提供しています。最初の会員*1は1996年1月に入会しているので，かれこれ5年近くサービスを提供し続けていることになります。

この5年間でさまざまな障害が発生しましたが，1日以上にわたるサービス停止に陥ったことは一度もありません。道楽*2としてはなかなかの安定度ではないかと思っています。

今春UPS(無停電電源装置)を購入し，これで停電*3が起きても安心と思っていた矢先に今回のトラブルは起きました。深夜2時30分ごろ，寝る前にメールをチェックしようとしてゲートウエイの電源が落ちていることに気付いたのです*4。即座にバックアップ用のサーバーをゲートウエイとして立ち上げ，原因を調べました。

電源ファンすら回っていなかったので，ヒューズが切れたのかと思って電源ユニットの蓋を開けてみたら，ユニット内に電解コンデンサのケースが転がっていて，基板上に電解液が吹き飛んだ後のコンデンサの残骸が残っていた，というわけです。

バックアップ用のサーバーの電源ユニットと入れ替えるだけでOKと思ったのが間違いでした。電源ユニットからマザーボードへの線の長さが足りず，結局マザーボードの入れ替えまでする羽目になったのです*5。午前3時20分ごろようやく入れ替えが終わったのですが，20Gバイトもあるディスクのfsckに延々30分以上かかるので，復旧は午前4時06分になってしまいました。

予備の電源ユニットか，バックアップ用のサーバーがもう1台必要，というのが今回のトラブルの教訓でした。あるいはケースは安物でも構わないけど，電源ユニットは良いものを使え，ということかも知れません。

ssh（セキュア・シェル）

sshとは，「Secure SHell（セキュア・シェル）」の略で，rsh（リモート・シェル）のセキュア版です。rshと同様，リモート・ホスト上で任意のコマンドを実行することができます。rshの関連コマンドに，リモート・ホストへログインするためのrlogin（リモート・ログイン）コマンド，ホスト間でファイルをコピーするrcp（リモート・コピー）コマンドがありますが，

＊1　現在も会員です。退会した会員はあまり多くはありません。
＊2　道楽と言いつつ，トラブル発生時に急きょ年休をとるなど，仕事より優先順位が高い道楽だったかも知れません。
＊3　停電後ブートし損なって，急きょ会社を早退して立ち上げ直したことがあります。
＊4　後でログを確認したら，午前0時58分ごろ落ちたようです。
＊5　電源ユニットの交換でなく，ディスクを交換してバックアップ用サーバーをゲートウエイとして立ち上げれば，復旧はもっと早かったことでしょう。

sshにもそれぞれ対応するコマンドとして，slogin(セキュア・ログイン)コマンド，scp(セキュア・コピー)コマンドがあります。

rsh/rlogin/rcpコマンドには，本連載の7回目「ファイアウォール(前編)」で解説したように，送信元アドレスだけの認証なので，IPアドレス・スプーフィングによって比較的容易に侵入される恐れがあります。

ssh/slogin/scpコマンドは，RSA暗号に基づいた認証方法を用いているので，通常の運用においては，秘密かぎが奪われない限り安全と見なすことができます*6。また，送受信するデータはすべて暗号化されるので盗聴される危険もありません。

ssh/slogin/scpコマンドは，rsh/rlogin/rcpコマンドと互換であり，完全に置き換えてしまうことができます。つまりsshを/usr/bin/rshとしてインストールできます。サーバー側の置き換えも簡単で,まず/etc/inetd.confにおいて，**図1**の行をコメントアウトします。次にinetd*7にHUPシグナルを送ってinetd.confの再読み込みを行わせて，in.rshdとin.rlogindが動かないようにした後，sshサーバーであるsshdを実行するだけです。/etc/rc.d/rc.localなどに，**図2**のように書いておいて，起動時にsshdが自動的に立ち上がるようにしておくと良いでしょう。

```
shell  stream  tcp     nowait  root    /usr/sbin/tcpd  in.rshd
login  stream  tcp     nowait  root    /usr/sbin/tcpd  in.rlogind
```

図1　rshの代わりにsshを使用する場合は/etc/inetd.confのこの行をコメントアウトする

```
# ssh
if [ -x /usr/sbin/sshd ]
then    /usr/sbin/sshd
fi
```

図2　/etc/rc.d/rc.localにこのように記述すると起動時にsshdが立ち上がる

写真1　ログイン先ホストやログイン方法を選択できる

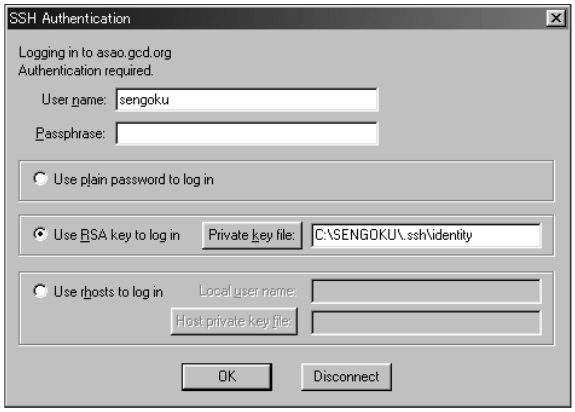

写真2　「Passphrase:」の部分にパスフレーズを入力するとログインできる

sshの実装には，OpenBSDプロジェクトによるOpenSSH*8や，SSH Communications Security*9によるSSH Secure Shellなどがあります。WindowsやMacintosh，さらにはPalm，BeOS，Java上の実装もあるようです。ここではOpenSSHを中心に解説しますが，他の実装でも基本は同じです。もちろんプロトコル・レベルでは，どの実装も互換ですから，異なる実装間での通信は問題なく，可能です。

sshはrshに比べれば暗号化/復号化処理が必要な分遅いのですが，最近のマシンは十分すぎるほど速いので，LAN上で高速転送を何度も行う*10必要があるのでも無い限り，遅さが全く気にならないと言っても過言ではないでしょう。

盗聴の多くがLAN上で行われる*11ことを考えれば，LAN内に閉じた通信であっても，rsh，rloginやtelnetよりはsshを使うべきだと思います。

幸い，Windows上にも使いやすいsshクライアントがあります。私が愛用しているのはターミナル・エミュレータ，

TeraTerm Pro＊12にsshのための拡張モジュールTTSSHを追加したバージョンです。

実行すると**写真1**のようにログイン先ホストやログイン方法（sshのほか，telnetすることもできる）を選べます。「OK」ボタンを押すと，**写真2**の画面になり，ここで「Passphrase:」の部分にパスフレーズを入力するとログインできます。

つまり普通のtelnetクライアントと同程度の手間でログインできるわけです。盗聴される危険があるtelnetクライアントをわざわざ使う必要性は全くありません。

秘密かぎと公開かぎ

sshでは，クライアント側とサーバー側のそれぞれが自身の秘密かぎと公開かぎを持ちます。サーバーのかぎはサーバーのインストール時に作られます。OpenSSHのデフォルトでは，秘密かぎと公開かぎをまとめたファイルが，/usr/local/etc/ssh_host_keyに，公開かぎをテキストで表現したファイルが，/usr/local/etc/ssh_host_key.pubにインストールされます。

一方，クライアントのかぎはユーザーそれぞれが自分専用のかぎを作らなければなりません。デフォルトでは，秘密かぎと公開かぎをまとめたファイルが~/.ssh/identityに，公開かぎをテキストで表現したファイルが，~/.ssh/identity.pubに作られます。

sshプロトコル

クライアント側のidentity.pubファイルの内容が，サーバー側の~/.ssh/authorized_keysに登録されていれば，sshクライアントを使ってログインできます。sshプロトコルについてより詳しく知りたい方は，インターネット・ドラフトdraft-ietf-secsh-architecture-05＊13などを参照してください。

ここではsshプロトコルの概要を簡単に説明します。

共通かぎの送信

sshクライアントがsshサーバーへ接続要求を出すと，互いのバージョン番号などを交換した後，クライアントが共通かぎをランダムに生成し，サーバーに対して送信します。この共通かぎ＊14は，クライアントとサーバーとの間の通信を暗号化するために使われるかぎです。共通かぎをサーバーに送信するときに盗聴されてしまっては元も子もありませんから，暗号化して送ることになりますが，この時のプロトコルを**図3**に示します。

まず，サーバーは自身の公開かぎ(1)をクライアントに対して送信します。クライアントは接続相手サーバーごとの公開かぎを保存しています。もしサーバーの公開かぎ(1)が，以前同じサーバーから送られてきた公開かぎと一致しなかった場合は，**図4**のような警告をクライアントがユーザーに対して表示します（OpenSSHの場合。以下，同様）。すなわち，だれかがセッションのハイジャッ

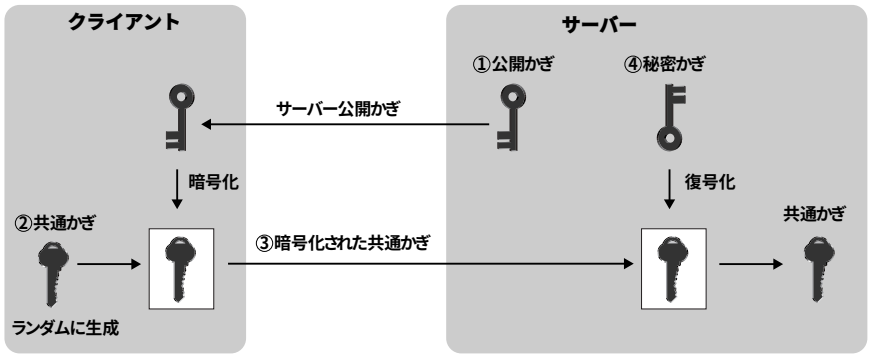

図3　共通かぎの送信
この後のクライアントとサーバー間の通信は共通かぎによって暗号化される。

＊6　もちろんsshサーバーにセキュリティ・ホールが発見された場合は，その限りではありません。sshのバージョンアップ情報には常に目を通す必要があります。
＊7　inetdが面倒を見るサービスのほとんどは今となっては使われないものばかりで，なぜ多くのディストリビューションがいまだにinetdをデフォルトで走らせているのか理解に苦しみます。inetdは前世紀の遺物ですからさっさと捨てるべきです。
＊8　http://www.openssh.com/を参照。
＊9　http://www.ssh.fi/を参照。
＊10　私は2台のマシン間でディスクを丸ごとコピーする（Gバイト単位の転送になります）ときでさえ，sshを使っています。盗聴を防ぐためというよりは，セキュアでない通信方法を一掃してしまったので，今さら暗号化しないコピーを行うのが面倒なためです。
＊11　サイト間通信の場合，大抵は一次プロバイダ（つまりそれなりに信頼できる企業）の設備のみを経由して通信が行われますから，盗聴の危険はインターネット上でなく，むしろ両サイトのLAN上の方が高いと言えます。この意味で，インターネット区間のみを暗号化するVPN方式はナンセンスです。
＊12　http://hp.vector.co.jp/authors/VA002416/を参照。
＊13　インターネット・ドラフトとは，IETFで標準化作業中の草稿です。標準化が行われればRFCになりますが，インターネット・ドラフトの段階では常に更新や置き換えが行われる可能性があります。インターネット・ドラフトは作られてから6カ月たつと無効になります。draft-ietf-secsh-architecture-05は次のURLで参照できます。http://search.ietf.org/internet-drafts/draft-ietf-secsh-architecture-05.txt。
＊14　暗号化に使われるかぎと，復号化に使われるかぎが同一である，対称暗号化方式におけるかぎ。

ク*15をしている可能性がある，という警告です。

もちろん，ハイジャックされたのではなくて，本当にサーバーのかぎが変更されたのかも知れません。サーバーの管理者に問い合わせる必要があるでしょう。図4の警告表示にあるように，クライアントは接続相手サーバーごとの公開かぎを~/.ssh/known_hostsファイルに保存しています。本当にサーバーのかぎが変更されたのであれば，known_hostsファイル内の，このサーバーに対応する部分を削除して，接続し直してください。

known_hostsファイルに，サーバーの公開かぎ(1)が登録されていない場合，クライアントには，その公開かぎが本物かどうか確認する手段がありません。そこで，ユーザーに対して**図5**のように表示*16し，ユーザーの確認を求めます。本当に接続するのか？(「Are you sure you want to continue connecting (yes/no)?」という問いに対し，「yes」と答えれば，サーバーの公開かぎ(1)が，known_hostsファイルに登録されます。

さて，これでサーバーから正しい公開かぎが送られてきたことが確認できました。次にクライアントはサーバーの公開かぎ(1)(図5を参照)を使って，共通かぎ(2)を暗号化します。そして暗号化された共通かぎ(3)をサーバーへ送信します。サーバーは，自身の秘密かぎ(4)を使って，暗号化された共通かぎ(3)を復号化することによって，共通かぎを得ます。

以上で，クライアントとサーバーは同じ共通かぎを持つことができました。この後，クライアントとサーバーとの間の通信は，この共通かぎによって暗号化されます。

この暗号通信が行える，ということはサーバーがクライアントと同じ共通かぎを持っていることの証明ですから，サーバーが正しい秘密かぎ(4)を持っていることの証明にもなります。つまりクライアントがサーバーの認証を行ったことになります。

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@    WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!     @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!
Someone could be eavesdropping on you right now (man-in-the-middle attack)!
It is also possible that the RSA host key has just been changed.
Please contact your system administrator.
Add correct host key in /home/sengoku/.ssh/known_hosts to get rid of this message.
RSA host key for azabu.klab.org has changed and you have requested strict checking.
```

図4　サーバーかぎが異なるという警告

```
The authenticity of host 'azabu.klab.org' can't be established.
RSA key fingerprint is 9e:f3:1d:35:67:ad:74:54:35:8a:7b:9a:46:dc:c4:c7.
Are you sure you want to continue connecting (yes/no)? yes↵
Warning: Permanently added 'azabu.klab.org' (RSA) to the list of known hosts.
```

図5　初めて接続するサーバーである場合の確認　　↵このマークで改行

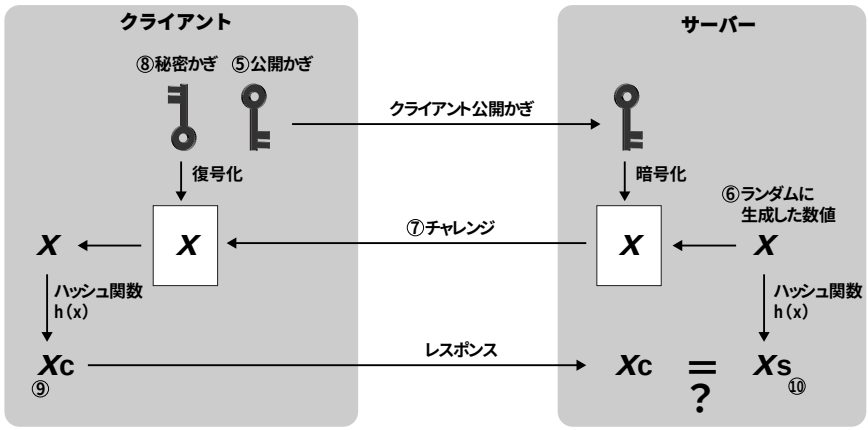

図6　クライアント認証

クライアント認証

次はサーバーがクライアントの認証を行う番です。

接続に先立って，サーバー上の各ユーザーのホーム・ディレクトリにある~/.ssh/authorized_keysファイルに，ログインを許可するクライアントの公開かぎを登録しておきます。クライアント認証とは，登録された公開かぎに対応する秘密かぎをクライアントが持っているか確認することです。もちろん秘密かぎ自体をサーバーに送ってしまうと，「秘密」でなくなってしまいますから，秘密かぎを送らずに秘密かぎを持っていることをサーバーに対して証明しなければなりません。クライアント認証のプロトコルを**図6**に示します。

まず，クライアントは，サーバーにログインするときのユーザーIDと，自身の公開かぎ(5)をサーバーに対して送信します。サーバーは，ログインを要求されたユーザーのホーム・ディレクトリのauthorized_keysファイルを確認して，ログインが許可されているか調べます。

authorized_keysファイルにクライアントの公開かぎ(5)が登録されていれば，サーバーはランダムな数値X(6)を生成し，公開かぎ(5)で暗号化します。そして暗号化されたXをクライアントに送信します。これをチャレンジ(7)と呼びます。つまりクライアントが本当に公開かぎ(5)に対応する秘密かぎを持っているのか試すための「チャレンジ」です。

クライアントは，チャレンジ(7)を秘密かぎ(8)で復号化して元のX(6)を得ます。Xをハッシュ関数MD5で変換した値X_C(9)をレスポンス(「チャレンジ」に対する受け答え)としてサーバーへ送信します。サーバーは，クライアントと同様にX(6)をハッシュ関数MD5で変換した値X_S(10)を計算し，クライアントから送られてきたレスポンスと比較します。

もし一致していたらならば，クライアントがチャレンジを正しく復号化できたことの証明になり，クライアントが正しい秘密かぎを持っていることの証明にもなります。つまりサーバーがクライアントの認証を行ったことになります。

ssh-keygen

クライアント側のかぎを作るにはクライアント側でssh-keygenコマンドを使います。ssh-keygenコマンドを使って，クライアントのかぎを作る実行例を**図7**に示します。

最初に，「Enter file in which to save the key」と，かぎを保存するファイル名を聞いてきます。デフォルトは ~/.ssh/identityです。この場合，秘密かぎと公開かぎをまとめたファイルが ~/.ssh/identityに，公開かぎをテキストで表現したファイルが~/.ssh/identity.pubに作られます。

~/.ssh/identity.pubファイルの内容を，サーバー側のホーム・ディレクトリの~/.ssh/authorized_keysファイルに追加(無い場合は新規に作成する)すれば，sshでログインできるようになります。

ここで注意すべきなのは秘密かぎ(identityファイル)の管理です。秘密かぎを持っている人はだれであれ正規ユーザーであるとサーバーに見なされるわけですから，絶対に第三者に盗まれないようにしなければなりません。当然，秘密かぎファイルはデフォルトでユーザー本人しか読めないパーミッションになっていますが，パーミッションが与えられて無い人でも何かのはずみに読むことができるかも知れません。

例えば，テープなど＊17にバックアップすれば，テープを読める人ならファイルのパーミッションに関係なく読めますし，マシンに物理的にアクセスできる人にとってはファイルのパーミッションなど関係ありません。フロッピあるいはCD-ROMなどからブートさせればだれでもrootになれるのですから。

また，Windows95/98などのOSでは，ファイルにユーザーごとのパーミッションを指定することさえできません。

パスフレーズの設定

そこで，sshでは秘密かぎは暗号化して保存する仕掛けになっています。暗号化のためのかぎが，「Enter passphrase」で入力するパスフレーズ＊18です。パスフレーズさえ他人に知られなければ，秘密かぎファイル自体は盗まれても大丈夫

```
asao:/home/sengoku % ssh-keygen↵
Generating RSA keys:  ......................................oooooo0.....oooooo0
Key generation complete.
Enter file in which to save the key (/home/sengoku/.ssh/identity):↵
Created directory '/home/sengoku/.ssh'.
Enter passphrase (empty for no passphrase): this is a sample passphrase↵

Enter same passphrase again: this is a sample passphrase↵

Your identification has been saved in /home/sengoku/.ssh/identity.
Your public key has been saved in /home/sengoku/.ssh/identity.pub.
The key fingerprint is:
2e:9b:b8:2e:0e:6a:a3:d9:b1:cf:77:14:63:73:0b:e4 sengoku@asao.gcd.org
```

図7　ssh-keygenコマンドで秘密かぎと公開かぎを作る

＊15　連載第7回「ファイアウォール(前編)」参照。
＊16　サーバーのかぎの指紋(fingerprint)は，サーバー上で「ssh-keygen -l -f /usr/local/etc/ssh_host_key.pub」などと実行することにより知ることができます。この指紋と，図5で表示される指紋が一致するか確認すれば，公開かぎが本物かどうか確認できます。
＊17　最近はテープをバックアップに利用することはほとんど無いかも知れません。
＊18　パスワードが8文字程度の「ワード」(単語)であるのに対し，パスフレーズ(語句)は複数の単語からなる「句」を設定できます。パスワードは辞書攻撃によって比較的簡単に破ることができますが，パスフレーズならば(よほど有名な句を設定しない限りは)破られる心配は無いと言えるでしょう。

です*19。

私の場合，Linux上でssh-keygenを実行して作ったかぎファイルidentityを，Windows98上のディレクトリ「C:\SENGOKU\.ssh\identity」にコピーして使っています。写真2のように，sshクライアントは実行時にパスフレーズの入力を要求しますから，他人に勝手にかぎを使われる心配はありません。

さて，ssh-keygenコマンドがパスフレーズの入力を要求したとき（図7中の「Enter passphrase」の部分）で，そのまま改行を押せば（「empty for no passphrase」），かぎファイルidentityが暗号化されずに作られますが，このようにして作ったかぎは，盗まれてもセキュリティ上問題がない用途に限定するべきです。特殊な事情がない限りは，必ずパスフレーズを設定するようにしてください。

パスフレーズの入力

もう一点，注意しなければならないのは，入力したパスフレーズが盗聴されないか，という点です。例えばtelnetなど（通信路が暗号化されないプロトコル）でリモート・マシンにログインして，ssh-keygenコマンドなどを実行した場合，入力したパスフレーズは平文でネットワークを流れますから，盗聴される危険が無いとは言えません。面倒でもssh-keygenを実行するマシンのコンソールなどから入力するべきです。

X端末は，セキュリティに十分注意を払わないと，キー入力やX端末に表示させた文字などが盗聴される恐れがあります。X 端末上でのパスフレーズの入力は，なるべく避けたほうが無難でしょう。

反面，Windowsマシンを端末として用いる場合など，手元のマシンで端末エミュレータを実行する場合は，マシン自体に第三者が小細工する恐れ*20が無ければ比較的安全です。

従って，手元のマシンがWindowsマシン（クライアントA），少し離れたところと遠隔地にLinuxマシンがある（それぞれサーバーB，サーバーC）場合，次のような手順を踏むと良いでしょう。

(1) サーバーBのコンソールからログインしてssh-keygenコマンドを実行し，パスフレーズを設定してidentityファイルを作成。これをフロッピへコピーした後，サーバーB上のidentityファイルを削除。identity.pubファイルを~/.ssh/authorized_keysへ移動。authorized_keysは他人から読めないようにパーミッションを設定する。例えば，「chmod 600 ~/.ssh/authorized_keys」を実行する。

(2) (1) のフロッピをクライアントAへ入れて identityファイルを適当な位置へコピー。フロッピは内容を完全に消去*21するか，厳重に保管。

(3) クライアントA上のsshクライアントを使ってサーバーBへログイン。この時，キーボードから入力するパスフレーズを肩越しに他人に盗み読まれないように注意（以下同様）。

(4) クライアントAとサーバーBの間の通信は暗号化されているので盗聴される心配はない。従ってサーバーB上で実行するコマンドに対してパスフレーズなどを安全に入力することができる。

(5) サーバーB上でssh-keygenコマンドを実行し，パスフレーズを設定して~/.ssh/identityおよび，~/.ssh/identity.pubを作成する。

(6) 適当な方法（ftp, メールなど）でサーバーB上の~/.ssh/identity.pubファイルの内容，あるいは（1）で~/.ssh/authorized_keysへ移したクライアントAのidentity.pubをサーバーCへ送る。identity.pubは公開かぎなので盗聴されても構わないが，内容が改変されていないか確認する必要はある。

(7) (5) のidentity.pubをサーバーC上の~/.ssh/authorized_keysに登録。

(8) サーバーB上のsshクライアント，あるいはクライアントA上のsshクライアントを使ってサーバーCへログイン。

手元のマシンがLinuxマシンなら，「サーバーB」を手元のマシンと読み替えて，(5) 以降の操作を行うだけで済みます。任意のサーバーに対して(6)から(8)の操作を繰り返せば，sshクライアントのログイン先をいくつでも増やすことが可能です。

上記手順は一見複雑に見えますが，ネットワークを流れるデータはすべて安全ではない，と仮定すれば必然的な手順と言えるでしょう。それに，いったん

すべてのサーバー・マシンと手元の端末間のすべての通信を暗号化する設定にしてしまえば，「安全でない」ネットワークを使う状況が無くなってしまうので，むしろ楽です。

すなわち，1カ所でも平文でデータが流れる部分があると，その部分に重要なデータが流れないか常に気を配らなければならないわけで，それなら最初の一度だけ十分注意を払って平文で流れる部分を完全に無くしてしまう方が理にかなっています。上記手順(8)でsshクライアントを使ってログインする実行例を**図8**に示します。

クライアントAとサーバーB間の通信はsshで暗号化されているので，サーバーB（図8の例では asao.gcd.org）上では，安全にパスフレーズを入力することができます。しかし，rshコマンドと比べるとパスフレーズを入力しなければならない分，面倒に感じるかも知れません。普段，リモート・ログインの手段としてtelnetを使っている人ならば，ユーザーIDとパスワードを入力するのに比べれば楽なので，パスフレーズを入力するのもさほど面倒とは感じないかも知れません。

では，リモート・コピーの場合はどうでしょうか（**図9**）。rcpコマンドならばパスワード無しにファイルがコピーできることを考えると，ファイル1つコピーするたびにパスフレーズを入力するのは手間と感じる人の方が多くなりそうです。

ssh-keygenコマンドを使ってクライアントのかぎを作成するときに，パスフレーズを設定しなければ，パスフレーズ無しにリモート・ログインやリモート・コピーできますが，前述したようにidentityファイルが盗まれるだけで秘密かぎが他人の手に渡ってしまうので，かなり危険です。

では，どうすればいいのでしょうか？

ssh-agent

暗号化したかぎファイルだと，復号化するためにパスフレーズが必要。だからといって復号化したファイルを置いておくと，ファイルを盗まれる恐れがある。ならばファイルでなくメモリー上に置いておけば良い，という発想で作られたのがssh-agentです。メモリー上に置いたデータならばマシンをリブートすれば消えてしまいますから，ファイルに置くよりは安全と言えます。

ssh-agentコマンドは実行するとデーモンとして動作します。sshクライアントなど，他のコマンドと通信するために，ssh-agentコマンドはunixドメイン・ソケットを使用します。実行例を**図10**に示します。

この場合，unixドメイン・ソケット/tmp/ssh-JOmUZ377/agent.377を使用し，デーモンのプロセスIDが378番であることを示します。このファイル名およびプロセスIDをsshクライアントなどに伝えるために環境変数を利用します。つまり，**図11**などと実行してからsshコマンドなどを実行します。

もちろんsetenvコマンドを手で入力するのは面倒なので，ssh-agentを実行するとき，その出力が自動的に実行されるように，evalコマンドを使うと良いでしょう(**図12**)。

```
asao:/home/sengoku % ssh azabu.klab.org↵
Enter passphrase for RSA key 'sengoku@asao.gcd.org': this is a sample passphrase↵
Last login: Mon Oct  9 15:35:15 2000 from asao.gcd.org
azabu:/home/sengoku %
```

図8　sshクライアントを使ってリモート・ログイン

```
asao:/home/sengoku % scp .cshrc azabu:/tmp/.↵
Enter passphrase for RSA key 'sengoku@asao.gcd.org': this is a sample passphrase↵
.cshrc                    |         1 KB |   1.8 kB/s | ETA: 00:00:00 | 100%
```

図9　sshクライアントを使ってリモート・コピー

```
asao:/home/sengoku % ssh-agent ↵
setenv SSH_AUTH_SOCK /tmp/ssh-JOmUZ377/agent.377;
setenv SSH_AGENT_PID 378;
echo Agent pid 378;
```

図10　ssh-agent コマンドと環境変数

```
asao:/home/sengoku % setenv SSH_AUTH_SOCK /tmp/ssh-JOmUZ377/agent.377↵
asao:/home/sengoku % setenv SSH_AGENT_PID 378↵
```

図11　環境変数を設定する

＊19　とは言っても，盗まれないに越したことはないわけで，むやみにコピーしたり，複数の人が使うWindows95/98マシンにインストールするのは避けるべきでしょう。
＊20　例えば，キーボードとマシンの間のケーブルに盗聴装置を付けられてしまうと，どうしようもありません。もっと原始的に，キーボードを肩越しにのぞく監視カメラを設置するだけでも，パスフレーズを盗むことは可能です。
＊21　もちろん，DELコマンドで消すだけでは復活が可能なので，確実に内容を消す必要があります。

次に，ssh-agentデーモンのメモリーに秘密かぎを登録します。そのためにはssh-addコマンドを用います（**図13**）。ssh-addを実行する前に，あらかじめ図12を実行しておく必要があります。

図13のように，ssh-addはパスフレーズの入力を要求し，このパスフレーズを使ってidentityファイルを復号化し，ssh-agentデーモンのメモリーに登録します。いったんssh-addを使って秘密かぎをssh-agentに登録した後は，パスフレーズ無しにssh/slogin/scpコマンドが使えます。

例えば，リモートのazabu.klab.org上でhostnameコマンドを実行するには，**図14**のように実行します。

```
asao:/home/sengoku % eval `ssh-agent`
Agent pid 378
```

図12　ssh-agentの実行

```
asao:/home/sengoku % ssh-add
Need passphrase for /home/sengoku/.ssh/identity
Enter passphrase for sengoku@asao.gcd.org: this is a sample passphrase
Identity added: /home/sengoku/.ssh/identity (sengoku@asao.gcd.org)
```

図13　ssh-addを使って秘密かぎを登録する

```
asao:/home/sengoku % ssh azabu.klab.org hostname
azabu.klab.org
```

図14　sshを使ったリモート実行
秘密かぎをssh-agentのメモリーに登録してあるため，パスフレーズを入力しなくて済む。

```
RhostsAuthentication     no
RhostsRSAAuthentication  no
PasswordAuthentication   no
PermitEmptyPasswords     no
```

図15　サーバーの設定

```
RSAAuthentication        yes
```

図16　RSAクライアント認証のみ「yes」にする

```
Compression     yes

Host azabu
        HostName        azabu.klab.org

Host asao
        HostName        ube.gcd.org
        Port            443
        ProxyCommand    /home/sengoku/bin/proxy-klab %h %p
```

図17　クライアントの設定ファイル

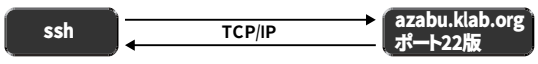

図18　「ssh azabu」を実行

設定ファイル

サーバーの設定ファイル

サーバーの設定ファイルは，デフォルトでは/usr/local/etc/sshd_configにあります。さまざまな設定が可能ですが，ほとんどはデフォルトのままで良いでしょう。ただし，安全性を高めるにはRSAクライアント認証以外の認証方法を禁止すべきです。

具体的には，設定ファイル中，次の項目を「no」に設定します（**図15**）。そして，RSAクライアント認証のみ「yes」にします（**図16**）。

クライアントの設定ファイル

クライアントの設定ファイルは，デフォルトでは/usr/local/etc/ssh_configと~/.ssh/configです。/usr/local/etc/ssh_configには全ユーザーに共通する設定，~/.ssh/configはユーザーごとの設定で，どちらも書式は同じです。職場[*22]のLAN上のLinuxマシンで私が使っている~/.ssh/configの例[*23]を**図17**に示します。

最初の「Host ～」の行までが，すべての接続先サーバーに共通な設定，それぞれの「Host ～」から次の「Host ～」までが，それぞれのサーバーごとの設定です。1行目の「Compression yes」は，クライアントとサーバーとの間でやりとりするデータを圧縮する，という設定です。インターネット経由の通信では，圧縮により帯域の有効利用が期待できます。「Host azabu」は，サーバー「azabu」に接続するときの設定で，「HostName azabu.klab.org」と指定していることか

ら，「ssh azabu」と実行すると，azabu.klab.orgに対してログインします（**図18**）。

同様に「ssh asao」と実行すると，ube.gcd.orgに対してログインします。ただし，「Port 443」と指定しているので，デフォルトのポート22番ではなくポート443番を使います。

さらに，図17の一番最後の行で「ProxyCommand/home/sengoku/bin/proxy-klab %h %p」と指定しているので，ube.gcd.orgのポート443番に直接接続するのではなく，/home/sengoku/bin/proxy-klabコマンドを介して接続します。

「%h %p」は，proxy-klabコマンドへの引数で，「%h」は接続先ホスト，「%p」は接続先ポートで置き換えられます。この場合であれば，「ube.gcd.org 443」という意味になります。

図19のように，sshはproxy-klabコマンドの標準入出力に対してsshプロトコルによる通信を行います。proxy-klabコマンドは，標準入力から受け取ったデータを，実際のサーバーであるube.gcd.orgへ中継し，ube.gcd.orgから受け取ったデータを標準出力へ送り出します。

なぜ，こんなまわりくどいことをするのでしょうか？ それはube.gcd.orgなど，インターネット上のホストに直接TCP/IP接続することができないからです*24。

sshを使ったファイアウォール越え

最近の企業サイトの多くはファイアウォールが設置され，社内のLANからインターネット上のホストへ直接TCP/IP接続することができないケースが多いのではないかと思います。そのような職場でも，Webに限りプロキシ経由の接続が認められているところが大多数でしょう。

ここではWebプロキシのホスト名がproxy.klab.orgで，ポート番号が8080であると仮定します。そして，ube.gcd.orgのポート443番でsshサーバーが動いているとします。

すると，Webプロキシ経由でube.gcd.orgのsshサーバーへ接続することが可能です。例えば，**図20**のように実行します。まず，proxy.klab.orgのポート8080番に接続します（図20中の1行目）。プロキシ（この例の場合は，squid）に接続したら，図20のように「CONNECT ube.gcd.org:443 HTTP/1.0」と入力します。これは https://ube.gcd.org/に接続するときに，Webブラウザがプロキシに対して送信する内容と同じです。つまりプロキシ側から見ると，普通のhttpsプロトコル（SSLで暗号化したhttpプロトコル）のように見えるわけです。

httpsプロトコルはSSLで暗号化されているので，その内容についてはプロキシは一切関知しません。単にWebブラウザとWebサーバーとの間に双方向の通信路を設定するだけです。従ってhttpsの代わりにsshプロトコルを流すことも可能なのです。

図20の最終行はube.gcd.orgのポート443番のsshサーバーが返したバージョン番号で，この後sshプロトコルによる通信が可能です。従って，proxy-klabコマンドは，図20と同じ動作を行った後，標準入力から受け取ったデータをその

図19　「ssh asao」を実行

```
kamiya:/home/sengoku % telnet proxy.klab.org 8080
Trying 10.10.0.3...
Connected to proxy.klab.org.
Escape character is '^]'.
CONNECT ube.gcd.org:443 HTTP/1.0

HTTP/1.0 200 Connection established

SSH-1.99-OpenSSH_2.2.0p1
```

図20　Webプロキシ経由の接続

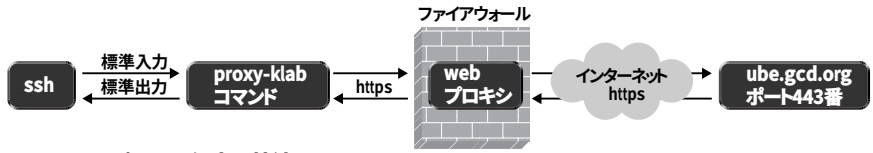

図21　Webプロキシ経由の接続

＊22　（株）K ラボラトリーです。
＊23　若干（かなり?），アレンジしてあります。
＊24　本当は，職場のLANから外へ自由にTCP/IP接続可能です。ここで紹介したproxy-klabコマンドは，親会社のネットワーク・トラブルにより一時的に接続ができなくなったときに，外部へsshでアクセスするために即興で書いたプログラムです。

ままube.gcd.orgのポート443番へ送信し, ube.gcd.orgのポート443番から受け取ったデータをそのまま標準出力へ送ればOKです(**図21**)。proxy-klabコマンドをperlスクリプトで実装した例を**図22**に示します。

Webプロキシの場合と同様に, telnetプロキシなどそのほかのプロキシの場合でも, 適切なProxyCommandを書き, インターネット側で適切なサーバーを立ち上げれば, プロキシ経由のssh接続が可能です。

図22　perlスクリプトで書いたproxy-klabコマンド

```
#! /usr/bin/perl
$PROXY_KLAB = "proxy.klab.org";
$PROXY_KLAB_PORT = 8080;
$Verbose = 0;

while ($_ = shift) {
    last if ! /^-(.*)/;
    if ($1 =~ /^v+$/) { $Verbose += length($&); next; }
    print<<EOF;
Usage: proxy [option...] <host> <port>
Options:
    -v    Verbose mode
EOF
}
$HOST = $_;
if ($_ = shift) {
    $PORT = $_;
} else {
    $PORT = 23;
}
print "Verbose Level: $Verbose\n" if $Verbose;

use Socket;
($name, $aliases, $proto) = getprotobyname('tcp');
($name, $aliases, $type, $len, $thataddr) = gethostbyname($PROXY_KLAB);
$that = sockaddr_in($PROXY_KLAB_PORT, $thataddr);
socket(S, PF_INET, SOCK_STREAM, $proto) || die "socket: $!";
connect(S, $that) || die "connect: $!";

if ($Verbose > 1) {
    $Rin = &fhbits('STDIN S');
} else {
    $Rin = &fhbits('S');
}
&login;
&connect;
exit 0;

# login 処理
sub login {
    &receive(0.1);
    &send("CONNECT $HOST:$PORT HTTP/1.0\r\n\r\n");
    do { &receive(0.1); } until (/HTTP\/[\d\.]+ 200.*\n\n/);
    $Raw =~ m/HTTP\/[\d\.]+ 200.*[\r\n]+/;
    $Raw = $';
}

# connect
sub connect {
    local($rout);
    print "CONNECT\n" if $Verbose;
    $Rin = &fhbits('STDIN S');
    syswrite(STDOUT,$Raw,length($Raw));
    while ((select($rout=$Rin,undef,undef,undef))[0]) {
        if (vec($rout,fileno(S),1)) {
            return if sysread(S,$_,1024) <= 0;  # EOF
            syswrite(STDOUT,$_,length);
        }
        if (vec($rout,fileno(STDIN),1)) {
            return if sysread(STDIN,$_,1024) <= 0; # EOF
            syswrite(S,$_,length);
        }
    }
}

# send(str);
# str を送る
sub send {
    undef $Buffer;
    undef $Raw;
    while( $_ = shift ) {
        print if $Verbose > 2;
        syswrite(S,$_,length);
    }
}

# receive(s);
# s 秒入力が途絶えるまで待つ
sub receive {
    local($timeout) = shift;
    local($rout);
    while ((select($rout=$Rin,undef,undef,$timeout))[0]) {
        if (vec($rout,fileno(S),1)) {
            &abort if sysread(S,$_,1024) <= 0;  # EOF
            $Raw .= $_;
            tr/\r\000\012\021\023\032/\n/d;
            $Buffer .= $_;
            print if $Verbose > 1;
        }
        if (vec($rout,fileno(STDIN),1)) {
            &abort if sysread(STDIN,$_,1024) <= 0; # EOF
            s/\n/\r/g;
            syswrite(S,$_,length);
        }
    }
    $_ = $Buffer;
}

sub fhbits {
    local(@fhlist) = split(' ',$_[0]);
    local($bits);
    for (@fhlist) {
        vec($bits,fileno($_),1) = 1;
    }
    $bits;
}

sub abort {
    exit(1);
}
```