LinuxFocus article number 273
http://linuxfocus.org

by Bernard Perrot
<bernard.perrot(at)univ-rennes1.fr>

About the author:
Bernard has been System and Network Engineer for the CNRS (French National Scientific Research Center) since 1982. He has been in charge of missions concerning computer system security at the "Institut National de Physique Nucléaire et de Physique des Particules" (In2p3). Now, he is working for the Institute of Mathematical Research (IRMAR) in the Physical and Mathematical Sciences (SPM) department.

Translated to English by:
Guy Passemard
<g.passemard(at)free.fr>

Secure your connections with SSH

Abstract:

This article was first published in a Linux Magazine France special issue focusing on security. The editor, the authors and the translators kindly allowed LinuxFocus to publish every article from this special issue. Accordingly, LinuxFocus will bring them to you as soon as they are translated to English. Thanks to all the people involved in this work. This abstract will be reproduced for each article having the same origin.

The ambition of this article is to take a good look at SSH, and why it is used. This isn't a tutorial or installation guide, rather an introduction to the vocabulary and the features of SSH. The links and documentation all along this article, will give you all implementation details.

What is SSH used for ?

First of all (and historically), SSH (the command *ssh*) is a secure version of the *rsh* (and *rlogin*) commands. SSH means "*Secure SHell*" as *rsh* means "*Remote SHell*". So, when *rsh* can give you easy access to a remote shell, but without a mechanism of user authentication, *ssh* provides the same service but with high multi level security.

If we wanted to be very brief for a user who doesn't want to know more (or do more), one could stop here and say administrator has done his job and installed the software (this is quite easy today), all you need is the *ssh* command replacing the *telnet, rsh* or *rlogin* commands, and everything works the same but with more security.

So, when you were doing:

`% rlogin serveur.org` (or `telnet serveur.org`)

you now do:

```
% ssh serveur.org
```

and it's already much better!

To end the short summary, I will suggest that today, all security incidents that could have been avoided by the simple usage of SSH in place of *rsh* (*rlogin*,*telnet*) are mainly a consequence of the victims' negligence.

What are the needs ?

To give more details, here are some of the most crucial and fragile aspects of interactive connections which one wishes to see resolved:

- First of all, avoid that passwords are transmitted over the net;
- use of strong authentication on connected systems, but not only based on the name or IP address which are subject to spoofing;
- execute remote commands in complete security.
- protect file transfer;
- secure X11 sessions which are very vulnerable

To respond to these needs, some solutions exist that are not really satisfying:

- The "*R-commands*": these commands do not transmit the password in clear text but the "rhosts" mechanism used is subject to numerous security problems;
- The "*One Time Password*,(OTP)": protects only authentication not the data flow that follows. Although this system is very attractive security wise, it has constraints that make it difficult to be accepted by users. Consequently it is mostly applied in environments that are not subject to ergonomic considerations;
- telnet with encryption: this solution is only applicable ... to telnet. Besides that, it doesn't include the X11 protocol, often a necessary complement.

And there is SSH, a good solution to:

- replace the *R-commands*: *ssh* instead of *rsh* and *rlogin*, *scp* with *rcp*, *sftp* with *ftp*;
- use strong authentication, based on encrytion algorithms using public keys (for systems as well as

users);
- make it possible to redirect the TCP data flow in a session "tunnel" and in X11 sessions, this can be done automatically;
- Encrypt the tunnel, and when necessary and requested, compress it.

SSH version 1 and SSH version 2

As nothing is perfect in this world, there are two incompatible versions of the SSH protocol: the 1.x version (1.3 and 1.5) and the 2.0 version. To pass from one version to the other is not painful for the user, having at hand the right client and the right server compatible with the version.

The SSH version 1 protocol is integrated, whereas SSH version 2 has redefined the previous protocol into three "layers":

1. SSH Transport Layer Protocol (SSH-TRANS)
2. SSH Authentication Protocol (SSH-AUTH)
3. SSH Connection Protocol (SSH-CONN)

Each protocol layer is specifically defined in a document (draft) normalized by IETF, followed by the fourth draft describing the architecture (SSH Protocol Architecture, SSH-ARCH). One will find all the details at: http://www.ietf.org/html.charters/secsh-charter.html

Without going into too many details, here is what you find in SSHv2:

- the transport layer provides integrity, encryption, and compression, the authentication of systems
- the authentication layer provides ... authentication (password, host-based, public key)
- the connection layer which manages the tunnel (shell, SSH-agent, port forwarding, flow control).

The main technical differences between SSH version 1 and 2 are:

SSH version 1	SSH version 2
monolithic (integrated) design	separation of the authentication, connection and transport functions into layers
integrity via CRC32 (not secure)	integrity via HMAC (hash encryption)
one and only one channel per session	unrestricted number of channels per session
negotiation using only a symmetric cipher in the channel, session identification with unique key on both sides	more detailed negotiation (symmetric cipher, public keys, compression, ...), and a separate session key, compression and integrity on both sides
only RSA for the public key algorithm	RSA and DSA for the public key algorithm
session key transmitted by the client side	session key negotiated thru the Diffie-Hellman protocol
session key valid for the entire session	renewable session key

Handling a bunch of keys

SSH key types, let me quickly define them:

- User key: a pair of keys composed of a public and private key (both asymmetric), user defined and permanent (kept on disk). This key allows user authentication if the public key authentication method is used. (described further on)
- Host key: also a pair of keys composed of a public and private key (both asymmetric), but defined at installation/configuration time by the server administrator and permanent (kept on disk). This key allows authentication between systems
- Server key: again a pair of keys composed of a public and private key (both asymmetric), but generated by a deamon at start up time and regularly renewed. This key remains in memory to secure the exchange of the session key in SSHv1 (With SSHv2,there is no server key as the exchange is secured by the the Diffie-Hellman protocol).
- Session key: this is a secret key used by the symmetric encryption algorithm to encrypt the communication channel. As always in modern cryptographic products, the key is random and perishable. SSHv1 has one key per session, on both sides. SSHv2 has two regenerated session keys, one on each side.

The user adds a pass phrase which protects the private key part of the above keys.This protection is assured by encrypting the file containing the private key with a symmetric algorithm. The secret key used to encrypt the file is derived from the pass phrase.

Authentication methods

There are several user identification methods, the choice is made from needs described in the security policies. The authorized methods are activated or not in the server’s configuration file. Here are the

principal categories:

- **"telnet like":**

  This is the "traditional" password method: when connecting, after having introduced one's identity, the user is invited to enter a password which is sent to the server which compares it to the one associated to the user The residual problem (the one that causes an astronomic number of piracies on Internet) is that the password is passed around *clearly* on the net , and so it can be intercepted by anyone who has a simple "*sniffer*". Here SSH has the same appearance (it's an easy method for a beginner migrating from telnet to SSH, as there is nothing more to learn ...), nevertheless the SSH protocol has encrypted the channel and the clearly readable password is encapsulated.

  A variant even more secure, configurable if one has the necessary stuff on the server and a "*One time password*" (S/Key for example): it's surely better, obviously more secure, but the ergonomic constraints make it applicable only to particular sites. This system operates as follows: after one has given his identity, instead of asking the user for a (static) password, the server sends a "*challenge*" to which the user must respond. The challenge being always different, the answer must also change. In consequence, the interception of the reply is not important as it cannot be reused. The constraint, as mentioned, essentially comes from the coding of the response which needs to be calculated (by a token, software on the client, etc.), and the entry is rather "cabalistic" (six English monosyllables, in the best case).

- **"rhosts like"** (*hostbased*):

  In this case identification is similar to the R-commands with the files such as /etc/rhosts or ~/.rhosts which "certifies" the client sites. SSH only contributes to better site identification, and to the usage of private "*shosts*" files. This is not enough security wise and I don't advise the use of this method if it stands alone.

- **By public keys**

  Here, the authentication system will be based on asymmetric encryption (look at the article on encryption for more details; basically, SSHv1 uses RSA, and SSHv2 has introduced DSA). The user's public key is registered beforehand on the server and the private key is stored on the client. With this authentication system, no secrets travel on the net and are never sent to the server.

  This is an excellent system, still, its security is limited (from my point of view) by being almost exclusively dependent on the user's "seriousness" (this problem is not particular to SSH, but I believe that it's THE major problem of public key systems, such as today's popular PKI): so, to avoid public key compression on the client's computer, the key is normally protected by a password (most often called *pass phrase* to emphasize the need to use more than one *word*). If the user does not carefully protect (or not at all) his private key, someone can easily use it to get access to all of his resources. That is why I say that the security relies on the user's seriousness and his level of confidence, as in this system the server administrator has *no* way of knowing if the private key is protected or not. Today, SSH does not administer revocation lists (not many do , not even PKI...). For example, if a private key is stored without a pass phrase on a home computer (there aren't any bad guy's at home, so why bother yourself with a passphrase...?) and one day it

goes to the repair shop of an important dealer (don't laugh, that's what is going to happen when electronic signatures will become common...), the repairman (his son, his friends) will be able to get the private keys of every computer that transits his table.

Configuring the SSH authentication mechanism for the user is slightly different as one uses SSHv1, SSHv2, or OpenSSH, also for a MacOS^{tm} or Windows^{tm} client. The basic principles and steps to remember are:

- How to generate an "asymmetric key pair" (that is a private/public RSA or DSA key pair), most often on the client system, (if several client systems are connected then we generate the key pair on one of the clients and then replicate the keys on the other clients). Some of the Windows^{tm} and MacOS^{tm} clients don't have service programs generating key pairs, you must do the generation on a Unix box before duplicating the keys. The key pair is registered in the `~/.ssh`user directory.
- Copying a public key on servers that will be used for authentication. This is done by adding a line, corresponding to the generated key pair from the user's ~/.ssh directory, to a server file in the same directory. (The name depends on the SSH version, either `authorized_keys` or `authorization`).
- And that's all ... if the configuration requires further authentication at the server level, then the client will ask for a "*pass phrase*" at connection time.

Also, it is useful to know at least two more elements concerning authentication:

- **ssh-agent**

  One of the reasons why one doesn't protect his private key is the annoyance provoked by having to enter the key at every interactive connection and the key can't be used when the connection is done through background scripts. A remedy exists, the *SSH agent*. It's a service program (*ssh-agent*), which once activated by you, can help you stock a three-fold identifier (username/hostname/pass phrase), and can submit the identifier in your place when required at connection time. On the client side, the password is asked only once, so one could call it SSO (*Single Sign On*).

  Now you have been informed, nothing forces you to protect your private keys, but if you don't, that's negligence, and you will be responsible for the consequences.

- **"*verbose*" mode**

  It happens that the connection fails for reasons that are unknown to the user: just add the "`-v`" option (stands for "*verbose*") in the *ssh* command. Consequently, you will receive numerous detailed messages on your screen during the connection, which will often give you enough information to determine the cause of the refusal.

The encryption algorithms

One must distinguish those for encrypting communication channels (encrypting with secret keys) and those used for authentication (encrypting with public keys).

For authentication, we can choose between RSA and DSA with version 2 of the protocol, and only RSA for version 1 (so no choice ...). Historically DSA was chosen, as RSA was in some countries *patented*. Since the end of the summer of year 2000, RSA is free of rights, and so this constraint has disappeared. I don't really have a preference on the good or bad choice, (just that DSA is a "pure" NSA product)

For symmetric encryption there is almost too much to choose from.... The protocol imposes a common algorithm that has to be present in all implementations : the *triple-DES with three keys*. Accordingly, it will be used if the negotiation between the client and server fails on other algorithms. If you can, try to negotiate with another algorithm, which is better, as the 3DES is now one of the least performing algorithms. Nevertheless we will put aside, unless necessary, the exotic or old ones (arc4, DES, RC4, ...) and limit our self to:

- IDEA: performs better than 3DES, but not completely free from licensing in certain conditions (it was often the default algorithm in Unix versions);
- Blowfish: very fast, probably safe, but the algorithm has to be put to the proof of time;
- AES: the new standard (replaces DES), if it is available to both sides, then use it, it is made for this.

Personally, I question myself about the interest to propose so many algorithms: even though the protocol allows the possibility to negotiate a "private one" (a particular group of users, for example), this seems to me essential, but for normal use, I think that with time, AES will be called to become the standard. If AES should get compromised, then security problems would be greater than those induced by SSH...

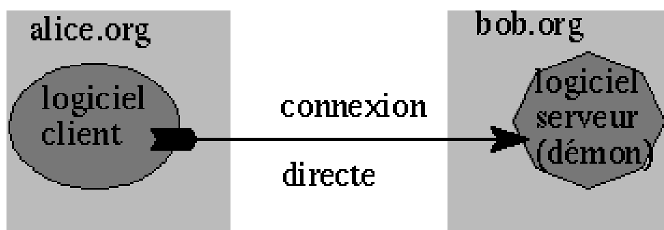
Port Forwarding, tunneling

SSH enables redirection (*forwarding*) of any TCP data flow through a "tunnel" in an SSH session. This means that the application data flow, instead of being directly managed by the client and server ports, is "encapsulated" in a "tunnel" created at connection (session) time. (*refer to the following diagram*).

This is done with no special effort (by the user) for the X11 protocol, with transparent handling of *displays* and allows continuous propagation when they are made via multiple hops.

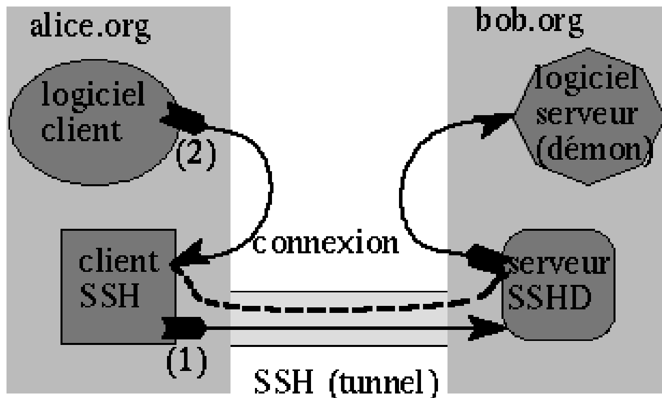
For other flows, there is a command line option, for each side:

- Direct connection between the client and server



(example: `user@alice% telnet bob.org`)

- Local port forwarding (client) to distant port (server)

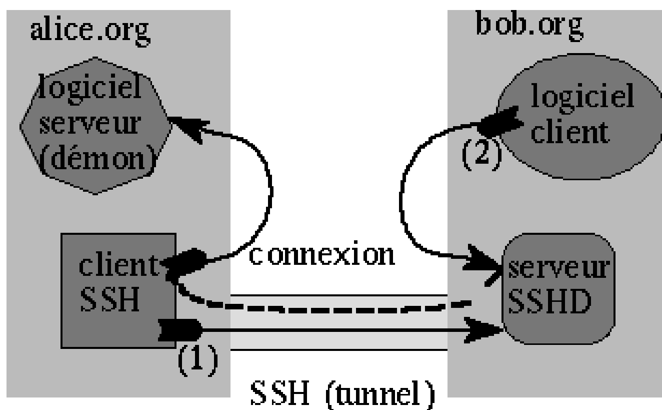


  example: `user@alice% ssh -L 1234:bob.org:143 bob.org`

  this system enables access from "*alice.org*" to the imap server of "*bob.org*",
  outside connections will be refused to its local network. They are made available only via the *localhost* address, port 1234, from the imap client executed on "*alice.org*".

  - (1) the user on "*alice.org*" opens (connection) the SSH tunnel
  - (2) the user on *"alice.org"* configures the client's local imap for access to the imap server located at *localhost* port 1234

- Forwarding a distant port (client) to a local port (server)



  example: `root@alice% ssh -R 1234:bob.org:143 bob.org`

  This is the same as above but the port on the remote host is forwarded. Only root has the privileges required to execute this SSH command but any user can then use this forwarded port/tunnel.

This powerful feature has sometime lead to SSH being called "a tunnel for the poor". One must understand that poverty here means: those who do not have administrator privileges on the client side.

Only in particular cases can a local port be forwarded with lower privileges (port > 1024) and without super-user rights ("*root*"). On the other hand, when privileged local port forwarding is needed, it either has to be done on a *root* account, or the client will have to be installed with super-user privileges ("*suid*") (In fact, a privileged local port allows redefinition of a standard service).

As with IP, it's quite easy to put anything in anything (and the contrary), it's not only possible to forward legacy TCP flows, but also PPP connections, which lets us make a "real" IP tunnel in IP (which is encrypted, therefore secure). The method exceeds the frame of this article, you can read the "Linux VPN-HOWTO" for details and setup scripts (you can also find native VPN solutions for Linux, like "*stunnel*" which you should consider before making a final choice).

Keep in mind that the first possibility is to redirect *telnet* flows: This could seem totally useless, as SSH implements interactive connection by default. However, while forwarding *telnet* connections, you may use your preferred client instead of the SSH interactive mode. You could particularly appreciate this in a Windows^{tm} or MacOS^{tm} environment where the SSH client may not show the user's favorite ergonomy. For example, the "terminal emulation" part of the "*Mindterm*" client (Java's SSH client, present on all modern systems) suffers from the lack of performance of the Java language: it can be advantageous to use this client only to open the SSH tunnel.

In the same way, you can also start a distant client like "*xterm*" (for instance, using automatic X11 forwarding in SSH), which allows us to use SSH on X terminals.

Note that the tunnel stays open as long as there is a flow of data, even if it does not come from the initiator. So, the "*sleep*" command becomes very useful to open an SSH tunnel to forward a new TCP connection.

```
% ssh -n -f -L 2323:serveur.org:23 serveur.org sleep 60

% telnet localhost 2323

... welcome to serveur.org ...
```

The first line opens the tunnel, starts the command "*sleep 60*" on the server, and forwards the local port number 2323 towards the distant (*telnet*) port number 23 . The second starts a *telnet* client on the local port number 2323, and then will use the (encrypted) tunnel to connect to the server's *telnetd* deamon. The "*sleep*" command will be interrupted after a minute (there's only a minute lapse time to start telnet) , but SSH will close the tunnel only when the last client has finished.

Main distributions: available for free

We have to distinguish between clients and/or servers on the different platforms and you should know that SSH version 1 and SSH version 2 are incompatible. The references at the end of the article will help you find other implementations, not listed in the following table which is limited to free products with sufficiently stable features.

product	platform	protocol	link	notes
OpenSSH	Unix	versions 1 and 2	www.openssh.com	details below
TTSSH	Windows[tm]	version 1	www.zip.com.au/~roca/ttssh.html	
Putty	Windows[tm]	version 1 and 2	www.chiark.greenend.org.uk/~sgtatham/putty	only beta
Tealnet	Windows[tm]	version 1 and 2	telneat.lipetsk.ru	
SSH secure shell	Windows[tm]	versions 1 and 2	www.ssh.com	free for non commercial use
NiftytelnetSSH	MacOS[tm]	version 1	www.lysator.liu.se/~jonasw/freeware/niftyssh/	
MacSSH	MacOS[tm]	version 2	www.macssh.com	
MindTerm	Java	version 1	www.mindbright.se	v2 now commercial

Notice that MindTerm is both an independent implementation of Java (you just need a Java *runtime*) and a *servlet* that can be executed inside a compatible well designed Web browser. Unfortunately, the last versions of this excellent distribution have just become commercial products.

OpenSSH

Today this distribution is probably the one to use in a Unix/Linux environment (continuous support, good response time, open-source and free).

OpenSSH development began with the original version (SHH 1.2.12) of Tatu Ylonen (the last really free) in the OpenBSD 2.6 project (via OSSH). Now, OpenSSH is developed by two groups, one developing only for the OpenBSD project, the other continuously adapting the code to make a portable version.

All this has some consequences, particularly the code has become more and more, a constant monster adaptation (I feel the "*sendmail*" syndrome appearing at the horizon and this is not healthy for an application dedicated to encryption which should be extremely rigorous).

Besides clean and readable coding, two other points annoy me:

- OpenSSH uses for its encrytion services the OpenSSL library, and generally this library is dynamically linked. In our case, the implementation of an encryption utility package having characteristics such as a good security level and fully reliable features, makes the preceding approach seem to me totally wrong. Of course, an attack on the library equals an attack on the

product. Other than a perversive attack, the encrytion characteristics (quality) in OpenSSH is/will be those of the library, which will live it's life independently of OpenSSH.

- OpenSSH uses for some of it's sensitive services (for example, a random-number generator) the OpenBSD system services. In only this context, I will make the same remarks of external dependence as in OpenSSL. Even more annoying, the portable version of OpenSSH, that should work on other platforms, delegates services required by OpenBSD to diverse mechanisms on other target platforms. For example, whether a random-number generator is available or not on your system, we will use it or another internal one. Consequently, the effective entropy of OpenSSH becomes dependent of the runtime platform, not to say moderately deterministic.

In my opinion (and I am not alone), a multiplatform encrytion product should have a demonstrated, determined and constant behavior whatever the platform is, as well as taking into account (eliminating) the platform's particular characteristics and their evolution.

This said, we have to admit that, the competitor's implementations are neither numerous nor attractive. I believe that it's more pragmatic to consider that today OpenSSH is the worst implementation excluding all others ...! A useful project for the community would be to redesign and to rewrite the code.

Bad news ...

SSH isn't miraculous! It does well what it is made for, but you can't ask for more. Particularly it will not prevent "authorized" connections: if an account is compromised , it will be possible for the intruder to connect himself via SSH to your computer, even though it's the only way, as he controls the authentication. SSH is fully reliable only when it is correlated by a practical and coherent security policy: if someone uses only one password, and he doesn't use SSH everywhere, the potential risk is only slightly diminished. You can admit that in this situation SSH can "backfire" as the intruder can use a secured encrypted connection with tunneling, he will be able to do almost all he wants without any possibility of efficient tracing for you.

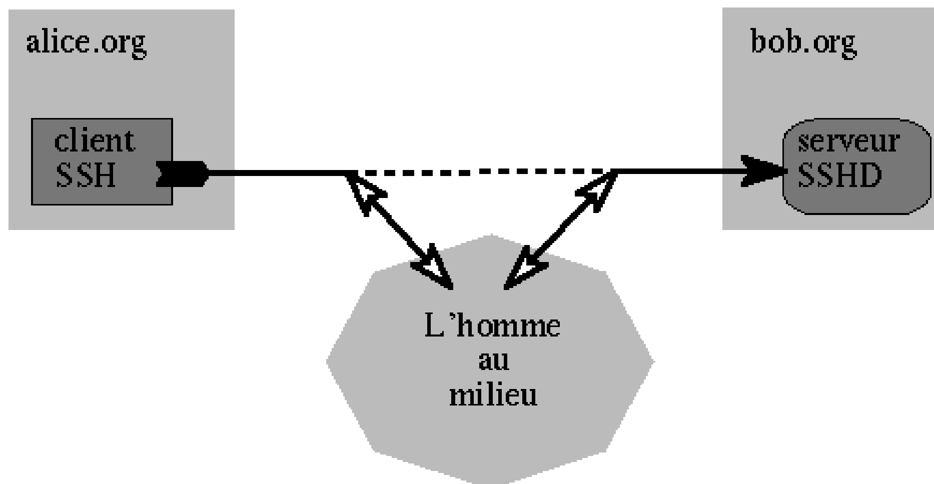
In the same style, one must also take in account well made "rootkits" which generally contain a SSH deamon to make a discreet return to your system, but with a few modifications: it doesn't listen on port 22 of course, it has the delicacy to stop logging, it's named by an ordinary deamon (for example *httpd*), also invisible for a "ps" command (which also has been somewhat modified by the rootkit).

On the contrary, one must not be too worried by the danger that can represent a SSH deamon which could let intruders be even more undercover: you know (I hope) that it is possible to put anything in almost anything in IP, including "misappropriation" of the essential protocols via a firewall: HTML tunneling, ICMP tunneling , DNS tunneling, ... So don't turn on your computer if you want a 100% secure system ;-).

SSH is not exempt of security "loopholes" derived from the implementation (many have been corrected in the past, as there is no perfect program), but also at the protocol level. These "loopholes", even though they may be announced as very alarming, generally concern weaknesses that are difficult to employ being technically complex to manipulate: One must keep in mind that the security incidents that could have been avoided by the use of SSH are daily, whereas those shown to be caused by weak points of SSH are somewhat theoretical. It should be interesting to read the study concerning "*man in the middle*"

attacks: http://www.hsc.fr/ressources/presentations/mitm/index.html. Nevertheless it will be necessary to take in account these potential vulnerabilities for "high security;" applications (banking, military, ...), where the means used by the cracker, highly motivated by the stakes and the profits, can be considerable.

- "Man in the middle" attack:



The aggressor intercepts the packets from both sides, and generates his packets to deceive each side
(different scenarios are possible, to the extent of terminating the connection on one side, and continuing with the other side making it believe it is the normal partner)

I often like to point out an incomprehensible weakness of the protocol concerning the "padding" (known as covered channel): in both version 1 and 2 the packets, have a length which is a multiple of 64 bits, and are padded with a random number. This is quite unusual and therefore sparing a classical fault that is well known in encrypting products: a "hidden" (or "subliminal") channel. Usually , we "pad" with a verified sequence as for example, give the value *n* for the byte rank *n* (*self describing padding*). In SSH, the sequence being (by definition) randomized, it cannot be checked. Consequently, it is possible that one of the parties communicating could pervert / compromise the communication for example used by a third party who is listening. One can also imagine a corrupted implementation unknown by the two parties (easy to realize on a product provided with only binaries as generally are commercial products). This can easily be done and in this case one only needs to "infect" the client or the server. To leave such an incredible fault in the protocol, even though it is universally known that the installation of a covered channel in an encryption product is THE classic and basic way to corrupt the communication, seems unbelievable to me . It can be interesting to read Bruce Schneier's remarks concerning the implementation of such elements in products influenced by government agencies. (http://www.counterpane.com/crypto-gram-9902.html#backdoors).

I will end this topic with the last bug I found during the portage of SSH to SSF (French version of SSH), it is in the coding of Unix versions before 1.2.25. The consequence was that the random generator produced ... predictable... results (this situation is regrettable in a cryptographic product, I won't go into the technical details but one could compromise a communication while simply eavesdropping). At the time SSH's development team had corrected the problem (only one line to modify), but curiously enough without sending any alert, not even a mention in the "changelog" of the product... one wouldn't

have wanted it to be known, he wouldn't have acted differently. Of course there is no relationship with the link to the above article.

Conclusion

I will repeat what I wrote in the introduction: SSH, neither produces miracles, nor solves alone all security problems, but makes it possible to handle efficiently most fragile aspects of the historical interactive connection programs (telnet, rsh...).

Bibliography, essential links

The following two books cover SSH version 1 and SSH version 2:

- *SSH: the Secure Shell*
  Daniel J. Barret & Richard E. Silverman
  O'Reilly - ISBN 0-596-00011-1
- *Unix Secure Shell*
  Anne Carasik
  McGraw-Hill - ISBN 0-07-134933-2
- openssh: http://www.openssh.com

And if you want to spend some money, here's the place to start ...:

- http://www.ssh.com

Exploiting the covered channel (induced by the random padding in SSHv1)

Here is a way to exploit the covered (subliminal) channel made possible by the usage of random padding in SSHv1 (and v2). I decline all responsibility of heart attacks that may affect the most paranoid.

The SSHv1 packets have the following structure:

offset (bytes)	name	length (bytes)	description
0	size	4	packet size, field size without padding, thus: size = length(type)+length(data)+length(CRC)
4	padding	p =1 to 8	**random** padding : size adjusted so that the ciphered part is a multiple of eight
4+p	type	1	packet type
5+p	data	n (variable >= 0)	
5+p+n	checksum	4	CRC32

Only one field isn't encrypted: the "size". The length of the encrypted part is always a multiple of eight, adjusted by "padding". The padding is always performed, if the length of the last three fields is already a multiple of 8 then the padding will be eight bytes long ($5+p+n$ remainder 0 modulo 8). Considering the ciphering function C, symmetric, used in CBC mode, and the deciphering function C^{-1}. To simplify this demonstration, we shall take only the packets with eight bytes padding. As one of the packets arrives, instead of padding it with a random number, we will place a value $C^{-1}(M)$, of eight bytes in this case. This means the deciphering of a message M with the function C used to encrypt the channel (the fact that M is "deciphered" without being beforehand ciphered has no importance from a strict mathematical point of view, I will not detail here the practical implementation.). Next, we carry out the normal processing of the packet, that is the ciphering by blocks of eight bytes.

The result will be :

offset	contents	notes
0	size	4 bytes not ciphered
4	8 bytes padding (ciphered)	so $C(C^{-1}(M))$
12... end	type, data, CRC	

What is amazing here? The first encrypted block contains $C(C^{-1}(M))$. So as C is a symmetric encryption function, $C(C^{-1}(M)) = M$. This first block is sent decrypted in a crypted data flow! This only means that anyone eavesdropping on the communication who has knowledge of the stratagem will know how to exploit the information. Of course, one can assume that the message M is itself encrypted (against a public key, for example, which avoids putting a secret in the perverted code), which still keeps it from being decrypted for someone who's not informed.

For example, it takes three packets of this type to pass the triple-DES(168 bit) session key, after which the flux sniffer can decrypt all the communication. When the key is sent, it's no longer necessary to "pre-decipher" the perverted padding before injecting it into the packet, it is possible to use paddings of any sizes if one wants to add even more information.

The usage of this covered channel is *absolutely undetectable* ! (One must be careful to encrypt each

element of the message as previously explained, so that the entropy of the block doesn't reveal the stratagem. Undetectable because the padding is randomized, which eliminates possible validation tests. Random Padding should *never* be applied in cryptographic products.

What renders this channel even more dangerous than others in the protocol is induced by messages like SSH_MSG_IGNORE where you can use it *without* having knowledge of the encrypted key.

To avoid the perversive effects of random padding, one just has to define in the protocol the use deterministic padding: commonly called "*self describing padding*", meaning that the offset byte *n* contains *n*. Random padding has remained in SSH v2, it's a choice, so keep it in mind...

To conclude, I will just say that if I criticize the covered channel, it's because I would like a product like SSH, which claims to be of high security, to really offer a maximum of security. Now you are able to imagine that there exists many potential tampering opportunities in commercial products: only open source products can offer a solution to the primary requirement, the possibility to check the code (even though the checking often needs to be done).

Webpages maintained by the LinuxFocus Editor team © Bernard Perrot "some rights reserved" see linuxfocus.org/license/ http://www.LinuxFocus.org	Translation information: fr --> -- : Bernard Perrot <bernard.perrot(at)univ-rennes1.fr> fr --> en: Guy Passemard <g.passemard(at)free.fr>

2005-01-14, generated by lfparser_pdf version 2.51