lsh

Table of Contents

- Introduction
  - Threats
  - Features
  - Related programs and techniques
    - ssh-1.x
    - ssh-2.x
    - Kerberos
    - IPSEC
- Installation
- Getting started
  - Initializing the randomness generator
  - lsh basics
  - Port forwarding
  - lshd basics
  - Using public-key user authentication
  - Using SRP authentication
  - Examining keys and other sexp files
  - Converting keys from ssh2 and OpenSSH
- Invoking lsh
  - Algorithm options
  - Host authentication options
  - User authentication options
  - Action options
  - Verbosity options
- Invoking lshg
- Invoking lshd
- Files and environment variables
- Terminology
- Concept Index

Node: Top, Next: Introduction, Previous: (dir), Up: (dir)

This document describes lsh and related programs. The lsh suite of programs is intended as a free replacement for the ssh suite of programs. In turn, ssh was intended as a secure replacement for the rsh and rlogin programs for remote login over the Internet.

lsh is a component of the GNU system.

This manual explains how to use and hack lsh; it corresponds to lsh version 2.0.

- Introduction:
- Installation:
- Getting started:
- Invoking lsh:
- Invoking lshg:
- Invoking lshd:
- Files and environment variables:
- Terminology:
- Concept Index:

--- The Detailed Node Listing ---

Introduction

- Threats:
- Features:
- Related techniques:

Related programs and techniques

- ssh1: SSH version 1
- ssh2: SSH version 2
- Kerberos: Kerberos
- ipsec: IP Sec

Getting started

- lsh-make-seed: Initializing the randomness generator
- lsh basics: Connection with lsh
- tcpip forwarding: Forwarding TCP/IP ports
- lshd basics: Starting the lshd deamon
- public-key: Using public-keys
- srp: Using SRP authentication
- sexp: Examining keys and other S-exp files
- Converting keys:

Invoking `lsh`

- Algorithms: Selecting algorithms.
- Hostauth options:
- Userauth options:
- Actions: What to do after login.
- Messages: Tuning the amount of messages.

Introduction

What is this thing called computer security anyway? Why would you want to use a program like `lsh`?

This chapter explains the threats `lsh` tries to protect you from, and some of the threats that remain. It also describes some of the technologies used in `lsh`.

From time to time in this manual, I will speak about the *enemy*. This means anybody who is trying to eavesdrop or disturb your private communication. This usage is technical, and it does not imply that the enemy is somehow morally inferior to you: The enemy may be some awful criminals trying to eavesdrop on you, or it may be the police trying to eavesdrop on the same criminals.

The enemy can be a criminal, or a competitor, or your boss who's trying to find out how much you tell collegues at competing firms. It may be your own or somebody else's national security officials. Or your ex-boyfriend who happens to be too curious.

So what can the enemy do to your communications and your privacy? Remember that just because you're paranoid that doesn't mean that nobody is trying to get you...

- Threats:
- Features:
- Related techniques:

Node: Threats, Next: Features, Previous: Introduction, Up: Introduction

Threats

When logging in to some other machine via the Internet, either in the same building or a few continents away, there are several things that may be under enemy attack.

Local attacks
: The enemy controls your local environment. He or she may be looking over your shoulder. Your local machine might be cracked. Or there may be some device planted inside your keyboard transmitting everything you type to the attacker. About the same problems occur if the attacker has taken control over your target machine, i.e. the remote machine you have logged in to.

Denial-of-service attacks
: The enemy has cut your network cable, effectively stopping your communication. Even without doing physical damage, the enemy may be able to flood and overload computers or network equipment. Or disrupt network traffic by sending fake packets to hangup your TCP/IP connections.

Passive eavesdropping
: The enemy may be able to listen to your communication somewhere along its path. With the global Internet, it's difficult to predict who might be able to listen. Internet traffic between buildings just a few hundred meters apart have been observed temporarily being routed through half a dozen countries, perhaps a few thousand kilometers.

  And even without routing anomalies, it is possible that the enemy has been able to take control of some nearby machine, and can listen in from there. Of course, passive eavesdropping is most dangerous if you transmit cleartext passwords. This is the main reason not to use vanilla telnet to login to remote systems. Use a telnet with support for SSL or Kerberos, or use a program like `lsh` or `ssh`.

  A passive eavesdropper is assumed not to do anything nasty with your packets beyond listening to them.

Name resolution attacks
: The translation from symbolic DNS names to numeric ip-addresses may be controlled by the attacker. In this case, you may think that you are connecting to a friendly machine, when in fact you are connecting somewhere else.

Fake packets
: It is fairly easy to fake the source address of an IP-packet, although it is more difficult to get hold on the replies to the faked packets. But even without any replies, this can cause serious problems.

Man-in-the-middle attack
: In this attack, the enemy sits between you and the target. When communicating with you, he pretends to be the target. When communicating with the target, he pretends to be you. He also passes all information on more or less unmodified, so that he is invisible to you and the target. To mount this attack, the enemy either needs physical access to some network equipment on the path between you and the target, or he has been able to fool you to connect to him rather than to the target, for example by manipulating the DNS-system.

`lsh` makes no attempt to protect you from local attacks. You have to trust the endpoint machines. It seems really difficult to uphold any security if the local machine is compromised. This is important to keep in mind in the "visitor"-scenario, where you visit a friend or perhaps an Internet caf&#65533; and want to connect to some of the machines at home or at work. If the enemy has been able to compromize your friend's or the caf&#65533;'s equipment, you may well be in trouble.

Protection from denial-of-service attacks is also a very difficult problem, and `lsh` makes no attempt to protect you from that.

Instead, the aim of `lsh`, and most serious tools for cryptographic protection of

communications across the net, is to isolate the vulnerabilities to the communication endpoints. If you know that the endpoints are safe, the enemy should not be able to compromize your privacy or communications. Except for denial-of-service attacks (which at least can't be performed without you noticing it).

First of all, `lsh` provides protection against passive eavesdropping. In addition, if you take the appropriate steps to make sure that hostkeys are properly authenticated, `lsh` also protects against man-in-the-middle attacks and in particular against attacks on the name resolution. In short, you need only trust the security at the end points: Even if the enemy controls all other network equipment, name resolution and routing infrastructure, etc, he can't do anything beyond the denial-of-service attack.

And at last, remember that there is no such thing as absolute security. You have to estimate the value of that which you are protecting, and adjust the security measures so that your enemies will not find it worth the effort to break them.

Node: Features, Next: Related techniques, Previous: Threats,
Up: Introduction

Features

`lsh` does not only provide more secure replacements for `telnet`, `rsh` and `rlogin`, it also provides some other features to make it convenient to communicate securely. This section is expected to grow with time, as more features from the wish-list are added to lsh. One goal for `lsh` is to make it reasonable easy to extend it, without messing with the core security functionality.

`lsh` can also be used in something called gateway mode, in which you can authenticate once and set up a connection that can later be used for quickly setting up new sessions with `lshg` (see Invoking lshg).

`lsh` can be configured to allow login based on a personal key-pair consisting of a private and a public key, so that you can execute remote commands without typing your password every time. There is also experimental support for Thomas Wu's Secure Remote Password Protocol (SRP). Kerberos support is on the wish list but not yet supported (see Kerberos).

The public-key authentication methods should also be extended to support Simple Public Key Infrastructure (SPKI) certificates, including some mechanism to delegate restricted logins.

Forwarding of arbitrary TCP/IP connections is provided. This is useful for tunneling otherwise insecure protocols, like telnet and pop, through an encrypted `lsh` connection.

lsh also features a SOCKS-proxy which also provides tunneling of TCP/IP connections, but can be easily used, e.g. from within popular web browsers like Mozilla and Firefox for tunneling web traffic. There are also programs like tsocks that performs transparent redirection of network access through a SOCKS proxy.

Convenient tunneling of X was one of the most impressive features of the original ssh programs. Both lsh and lshd support X-forwarding, although lshg does not.

Whan X forwarding is in effect, the remote process is started in an environment where the DISPLAY variable in the environment points to a fake X server, connections to which are forwarded to the X server in your local environment. lsh also creates a new "fake" MIT-MAGIC-COOKIE-1 for controlling access control. Your real X authentication data is never sent to the remote machine.

Other kinds of tunneling that may turn out to be useful include authentication (i.e. ssh-agent), general forwarding of UDP, and why not also general IP-tunneling.

Node: Related techniques, Previous: Features, Up: Introduction

Related programs and techniques

This sections describes some other programs and techniques related to lsh. The ssh family of programs use mostly the same kind of security as lsh. Kerberos and IPSEC operate quite differently, in particular when it comes to protection against man-in-the-middle attacks.

- ssh1: SSH version 1
- ssh2: SSH version 2
- Kerberos: Kerberos
- ipsec: IP Sec

Node: ssh1, Next: ssh2, Previous: Related techniques, Up: Related techniques

ssh-1.x

The first of the Secure shell programs was Tatu Yl&#65533;nen's ssh. The latest of the version 1 series is ssh-1.33 which speaks version 1.5 of the protocol. The "free" version of ssh-1.33 does not allow commercial use without additional licensing, which makes ssh-1.33 non-free software according to Debian's Free Software Guidelines and the Open Source Definition.

The version 1 protocol has some subtle weaknesses, in particular, all support

for using stream ciphers was disabled by default a few versions back, for security reasons.

There also exists free implementations of `ssh-1`, for both Unix and Windows. `ossh` and later OpenSSH are derived from earlier version av Tatu Yl&#65533;nen's `ssh`, and are free software.

Node: ssh2, Next: Kerberos, Previous: ssh1, Up: Related techniques

`ssh-2.x`

`ssh2` implements the next generation of the Secure Shell protocol, the development of which is supervised by the IETF secsh Working Group. Besides `lsh`, some well known implementations of this protocol includes

- OpenSSH (which supports version 2 of the protocol since May 2000).
- The `ssh2` series of proprietary programs sold by the SSH company. `lsh` interoperates with current versions of these programs, but not with version 3.0 and earlier (the older versions get some details of the protocol wrong, probably because it predates the protocol specification). The license for the SSH company's `ssh2` programs is similar to that for recent versions of `ssh1`, but with a narrower definition of "non-commercial use".
- `putty`, a free `ssh` implementation for Microsoft Windows.

There a numerous other implementations, both free and proprietary. The above list is far from complete.

Node: Kerberos, Next: ipsec, Previous: ssh2, Up: Related techniques

Kerberos

Kerberos is a key distribution system originally developed in the late 1980:s as a part of Project Athena at MIT. Recent development have been done at The Royal Institute of Technology, Stockholm (KTH).

Kerberos uses a central trusted ticket-granting server, and requires less trust on the local machines in the system. It does not use public-key technology.

Usually, Kerberos support is compiled into applications such as telnet, ftp and X-clients. The ssh family of programs, on the other hand, tries to do all needed magic, for instance to forward X securely, and then provides general TCP/IP forwarding as a kitchen sink.

I believe Kerberos' and lsh's protection against passive eavesdropping are mostly equivalent. The difference is in the set of machines and assumptions you have to trust in order to be safe from a man-in-the-middle attack.

I think the main advantage of `lsh` over Kerberos is that it is easier to install and use for on ordinary mortal user. In order to set up key exchange between two different Kerberos systems (or *Kerberos realms*), the respective system operators need to exchange keys. In the case of two random users at two random sites, setting up `lsh` or some other program in the ssh family is likely easier than to get the operators to spend time and attention. So `lsh` should be easier to use in an anarchistic grass-roots environment.

Another perspective is to combine ssh features like X and TCP/IP forwarding with authentication based on Kerberos. Such an arrangement may provide the best of two worlds for those who happen to have an account at a suitable ticket-granting server.

Node: ipsec, Previous: Kerberos, Up: Related techniques

IPSEC

IPSEC is a set of protocols for protecting general IP traffic. It is developed by another IETF working group, and is also a required part of IP version 6.

Again, the main difference between IPSEC, Kerberos and ssh is the set of machines that have to be secure and the keys that have to be exchanged in order to avoid man-in-the-middle attacks.

Current protocols and implementations of IPSEC only provide authentication of machines; there's nothing analogous to the user authentication in ssh or Kerberos.

On the other hand, IPSEC provides one distinct advantage over application level encryption. Because IP and TCP headers are authenticated, it provides protection against some denial-of-service attacks. In particular, it makes attacks that cause hangup of a TCP connection considerably more difficult.

So it makes sense to use both IPSEC and some application level cryptographic protocol.

Also note that it is possible to use the *Point-to-Point Protocol* (PPP) to tunnel arbitrary IP traffic accross an ssh connection. This arrangement provides some of the functionality of IPSEC, and is sometimes referred to as "a poor man's Virtual Private Network".

Node: Installation, Next: Getting started, Previous: Introduction, Up: Top

Installation

You install `lsh` with the usual `./configure && make && make install`. For a full listing of the options you can give to `configure`, use `./configure --help`. For example, use

`--without-pty` to disable pty-support.

The most commonly used option is `--prefix`, which tells configure where lsh should be installed. Default prefix is `/usr/local`. The `lshd` server is installed in `$prefix/sbin`, all other programs and scripts are installed in `$prefix/bin`.

The configure script tries to figure out if the linker needs any special flags specifying where to find dynamically linked libraries at run time (one case where this matters is if you have a dynamic libz.so installed in a non-standard place). Usually, you can use

```
./configure --with-lib-path=/opt/lib:/other/place
```

to specify extra library directories, and the configure script should do the right thing. If this doesn't work, or you believe that you know your system better than `./configure`, just set LDFLAGS and/or LD_LIBRARY_PATH to the right values instead.

Node: Getting started, Next: Invoking lsh, Previous: Installation, Up: Top

Getting started

This section tells you how to perform some common tasks using the `lsh` suite of programs, without covering all options and possibilities.

- lsh-make-seed: Initializing the randomness generator
- lsh basics: Connection with lsh
- tcpip forwarding: Forwarding TCP/IP ports
- lshd basics: Starting the lshd deamon
- public-key: Using public-keys
- srp: Using SRP authentication
- sexp: Examining keys and other S-exp files
- Converting keys:

Node: lsh-make-seed, Next: lsh basics, Previous: Getting started, Up: Getting started

Initializing the randomness generator

Several of the lsh programs requires a good pseudorandomness generator for secure operation. The first thing you need to do is to create a seed file for the generator. To create a personal seed file, stored as `~/.lsh/yarrow-seed-file`, run

```
lsh-make-seed
```

To create a seed file for use by `lshd`, run

```
lsh-make-seed --server
```

as root. The seed file is stored as `/var/spool/lsh/yarrow-seed-file`.

Node: lsh basics, Next: tcpip forwarding, Previous: lsh-make-seed,
Up: Getting started

`lsh` basics

`lsh` is the program you use for connection to a remote machine. A few examples are:

```
lsh sara.lysator.liu.se
```

Connects to `sara.lysator.liu.se` and starts an interactive shell. In this example, and in the rest of the examples in this section, lsh will ask for your password, unless you have public-key user authentication set up.

The first time you try to connect to a new machine, `lsh` typically complains about an "unknown host key". This is because it has no reason to believe that it was the right machine that answered, and not a machine controlled by the enemy (see Threats). The default behaviour is to never ever accept a server that is not properly authenticated. A machine is considered authentic if it follows the protocol and has an acl-entry for its public hostkey listed in `~/.lsh/host-acls`.

To make lsh less paranoid, use

```
lsh --sloppy-host-authentication sara.lysator.liu.se
```

Then `lsh` will display a *fingerprint* of the host key of the remote machine, and ask you if it is correct. If so, the machine is considered authentic and a corresponding acl-entry is appended to the file `~/.lsh/captured_keys`. You can copy acl-entries you have verified to `~/.lsh/host-acls`.

You can even use

```
lsh --sloppy-host-authentication --capture-to ~/.lsh/host-acls
```

to get `lsh` to behave more like the traditional `ssh` program.

```
lsh -l omar sara.lysator.liu.se
```

Connects, like above, but tries to log in as the user "omar".

```
lsh sara.lysator.liu.se tar cf - some/dir | (cd /target/dir && tar -xf -)
```

Copies a directory from the remote machine, by executing one remote and one local `tar` process and piping them together.

```
CVS_RSH=lsh cvs -d cvs.lysator.liu.se:/cvsroot/lsh co lsh
```

Checks out the `lsh` source code from the CVS repository.

```
lsh -G -B sara.lysator.liu.se
```

Opens an ssh connection, creates a "gateway socket", and forks into the background.

```
lshg sara.lysator.liu.se
```

creates a new session using an existing gateway socket, without the overhead for a new key exchange and without asking for any passwords.

Node: tcpip forwarding, Next: lshd basics, Previous: lsh basics,
Up: Getting started

Port forwarding

One useful feature of `lsh` and other ssh-like programs is the ability to forward arbitrary connections inside the encrypted connection. There are two flavors: "local" and "remote" forwarding.

An example of local forwarding is

```
lsh -L 4000:kom.lysator.liu.se:4894 sara.lysator.liu.se
```

This makes `lsh` listen on port 4000 on the *local* machine. When someone connects, `lsh` asks the server to open a connection from the *remote* machine (i.e. `sara`) to port 4894 on another machine (i.e. `kom`). The two connections are piped together using an encrypted channel.

There are a few things that should be noted here:

- By default, `lsh` only listens on the loopback interface, so only clients on the same machine can use the tunnel. To listen on all interfaces, use the `-g` flag.
- A connection through the tunnel consists of three parts:
    1. From a client socket to the local port (4000 in this example) that `lsh` listens on.
    2. The tunnel itself, from the local machine to the tunnel endpoint,

which is `sara` in this example.
3. The connection from the tunnel endpoint to the ultimate target, in this example from `sara` to `kom`.

Only the middle part is protected by `lsh`: all data flowing through the tunnel is sent across the first and last part *in the clear*. So forwarding doesn't offer much protection unless the tunnel endpoint and the ultimate target machine are close to each other. They should usually be either the same machine, or two machines connected by a local network that is trusted.

- Port forwarding is very useful for traversing firewalls. Of course, you don't need to use lsh-style forwarding just to get out, there are other tools like HTTPTunnel for that. But `lsh` helps you get out through the firewall in a secure way.
- Port forwarding is done in addition to anything else `lsh` is doing. In the example above, a tunnel is set up, but `lsh` will also start an interactive shell for you. Just as if the `-L` option was not present. If this is not what you want, the `-N` or `-B` option is for you (see Invoking lsh)

Remote forwarding is similar, but asks the *remote* machine to listen on a port. An example of remote forwarding is

```
lsh -g -R 8080:localhost:80 sara.lysator.liu.se
```

This asks the remote machine to listen on port 8080 (note that you are probably not authorized to listen on port 80). Whenever someone connects, the connection is tunnelled to your local machine, and directed to port 80 on the same machine. Note the use of `-g`; the effect is to allow anybody in the world to use the tunnel to connect to your local webserver.

The same considerations that apply to forwarded local ports apply also to forwarded remote ports.

At last, you can use any number of `-L` and `-R` options on the same command line.

Node: lshd basics, Next: public-key, Previous: tcpip forwarding,
Up: Getting started

`lshd` basics

There are no global configuration files for `lshd`; all configuration is done with command line options (see Invoking lshd).

To run `lshd`, you must first create a hostkey, usually stored in `/etc/lsh_host_key`. To do this, run

```
lsh-keygen --server | lsh-writekey --server
```

This will also create a file `/etc/lsh_host_key.pub`, containing the corresponding public key.

A typical command line for starting lshd in daemon mode is simply

```
lshd --daemonic
```

You can find init script for `lshd` tailored for Debian's and RedHat's GNU/Linux systems in the `contrib` directory.

It is also possible to let `init` start `lshd`, by adding it in `/etc/inittab`.

Node: public-key, Next: srp, Previous: lshd basics, Up: Getting started

Using public-key user authentication

Public-key user authentication is a way to authenticate for login, without having to type any passwords. There are two steps: Creating a key pair, and authorizing the public key to the systems where you want to log in.

To create a keypair, run

```
lsh-keygen | lsh-writekey
```

This can take some time, but in the end it creates two files `~/.lsh/identity` and `~/.lsh/identity.pub`.

If you want to use the key to login to some other machine, say `sara`, you can do that by first copying the key,

```
lsh sara.lysator.liu.se '>my-key.pub' < ~/.lsh/identity.pub
```

then authorizing it by executing, on `sara`,

```
lsh-authorize my-key.pub
```

By default, `lsh-writekey` encrypts the private key using a passphrase. This gives you some protection if a backup tape gets into the wrong hands, or you use NFS to access the key file in your home directory. If you want an unencrypted key, pass the flag `-c none` to `lsh-writekey`.

For security reasons, you should keep the private key `~/.lsh/identity` secret. This is of course particularly important if the key is unencrypted; in that case, anybody who can read the file will be able to login in your name to any

machine where the corresponding public key is registered as an authorized key.

Naturally, you should also make sure not to authorize any keys but your own. For instance, it is inappropriate to use an insecure mechanism such as unauthenticated email, `ftp` or `http` to transfer your public key to the machines where you want to authorize it.

If you have accounts on several systems, you usually create one keypair on each of the systems, and on each system you authorize some or all of your other public keys for login.

Node: srp, Next: sexp, Previous: public-key, Up: Getting started

Using SRP authentication

The Secure Remote Password protocol is a fairly new protocol that provides mutual authentication based on a password. To use it, you must first choose a secret password. Next, you create a *password verifier* that is derived from the password. The verifier is stored on the target machine (i.e. the machine you want to log in to).

To create a verifier, you run the `srp-gen` program and type your new password. You have to do it on either the target machine, redirecting the output to ~/.lsh/srp-verifier, or you can generate it on some other machine and copy it to the target.

The main advantage of using SRP is that you use the password not only to get access to the remote machine, but you also use it to authenticate the remote machine. I.e. you can use it to connect securely, *without* having to know any hostkeys or fingerprints beforehand!

For instance, you could connect using SRP to fetch the hostkey fingerprint for the remote machine, as a kind of bootstrapping procedure, and then use traditional authentication methods for further connections.

For this to work, the verifier *must* be kept *secret*. If the enemy gets your verifier, he can mount some attacks:

- He can mount a *dictionary attack* on your password, i.e. generate a large list of likely password and check if any of them matches yours.
- He can impersonate the server. That means that if you try to connect to the remote machine using SRP, and the attacker can intercept your connection (e.g. by attacking the name resolution or routing system) he can successfully pretend to be the real server.

If you use SRP to get the hostkey or fingerprint for the remote machine, as outlined above, the impersonation attack destroys security, you could just as

well connect the hostkey presented by the remote server without verifying it at all.

If you use SRP exclusively, the situation seems somewhat different. As far as I can see, an attacker knowing your verifier can not mount a traditional man-in-the-middle-attack: He can play the server's part when talking to you, but in order to play your part when talking to the real server, he needs to know your password as well.

SRP support is disabled by default, but can be enabled by the `--srp-keyexchange` option to `lshd` and `lsh` (naturally, it won't be used unless enabled on both sides). At the time of this writing, SRP is too new to be trusted by conservative cryptographers (and remember that conservatism is a virtue when it comes to security).

And even if SRP in itself is secure, the way `lsh` integrates it into the `ssh` protocol has not had much peer review. The bottom line of this disclaimer is that the SRP support in `lsh` should be considered experimental.

As far as I know, using SRP as a host authentication mechanism is not supported by any other `ssh` implementation. The protocol `lsh` uses is described in the `doc/srp-spec.txt`. Implementations that use SRP only as a user authentication mechanism are not compatible with `lsh`.

Node: sexp, Next: Converting keys, Previous: srp, Up: Getting started

Examining keys and other sexp files

Keys and most other objects `lsh` needs to store on disk are represented as so called S-expressions or *sexps* for short. S-expressions have their roots in the Lisp world, and a variant of them in used in the Simple Public Key Infrastructure (SPKI). Currently, `lsh`'s support for SPKI is quite limited, but it uses SPKI's formats for keys and Access Control Lists (ACL:s).

There are several flavours of the sexp syntax:

- The canonical syntax is somewhere between a text and a binary format, and is extremely easy for programs to read and write.
- The transport syntax, which is suitable when embedding sexps in text files. It is essentially the canonical representation, encoded using base64.
- The advanced syntax, which is intended for humans to read and write, and bears some resemblance to Lisp expressions.

To see what your `~/.lsh/identity.pub` file really contains, try

```
sexp-conv < ~/.lsh/identity.pub
```

The sexp-conv program can also be used to computes fingerprints. The fingerprint of a key (or any sexp, for that matter) is simply the hash of its canonical representation. For example,

```
sexp-conv --hash </etc/lsh_host_key.pub
```

This flavour of fingerprints is different from the ssh fingerprint convention, which is based on a hash of the key expressed in ssh wire format. To produce ssh standard fingerprints, use lsh-export-key --fingerprint.

Node: Converting keys, Previous: sexp, Up: Getting started

Converting keys from ssh2 and OpenSSH

If you are already using ssh2 or OpenSSH, and have created one or more personal keypairs, you need to convert the public keys to lsh's format before you can authorize them. Use the supplied ssh-conv script,

```
ssh-conv <openssh-key.pub >new-key.pub
```

You can then use the usual lsh-authorize on the converted keys. ssh-conv supports both DSA and RSA keys.

Conversion of keys the other way is also possible, by using the lsh-export-key program. It reads a public key in the SPKI format used by lsh on stdin, and writes the key in ssh2/OpenSSH format on stdout.

If you want to use your lsh key to log in to another system running and OpenSSH server, you can do like this:

```
lsh-export-key --openssh < .lsh/identity.pub >sshkey
```

And on the other machine, after having somehow copied the sshkey file, just add it to the end of your authorized_keys file:

```
cat sshkey >> ~/.ssh/authorized_keys
```

lsh-export-key can also be used to check the fingerprint of keys (just like ssh-keygen).

```
lsh-export-key --fingerprint < /etc/lsh_host_key.pub
```

show the MD5 and Bubble babble fingerprint of the server public key.

There are currently no tools for converting private keys.

Invoking `lsh`

You use `lsh` to login to a remote machine. Basic usage is

`lsh [-p` *port number*`] sara.lysator.liu.se`

which attempts to connect, login, and start an interactive shell on the remote machine. Default *port number* is whatever your system's `/etc/services` lists for `ssh`. Usually, that is port 22.

There is a plethora of options to `lsh`, to let you configure where and how to connect, how to authenticate, and what you want to do once properly logged in to the remote host. Many options have both long and short forms. This manual does not list all variants; for a full listing of supported options, use `lsh --help`.

Note that for many of the options to `lsh`, the ordering of the options on the command line is important.

- Algorithms: Selecting algorithms.
- Hostauth options:
- Userauth options:
- Actions: What to do after login.
- Messages: Tuning the amount of messages.

Algorithm options

Before a packet is sent, each packet can be compressed, authenticated, and encrypted, in that order. When the packet is received, it is first decrypted, next it is checked that it is authenticated properly, and finally it is decompressed. The algorithms used for this are negotiated with the peer at the other end of the connection, as a part of the initial handshake and key exchange.

Each party provides a list of supported algorithms, and the first algorithm listed by the client, which is also found on the server's list, is selected. Note that this implies that order in which algorithms are listed on the server's list doesn't matter: if several algorithms are present on both the server's and the client's lists, it's the client's order that determines which algorithm is selected.

Algorithms of different types, e.g. data compression and message

authentication, are negotiated independently. Furthermore, algorithms used for transmission from the client to the server are independent of the algorithms used for transmission from the server to the client. There are therefore no less than six different lists that could be configured at each end.

The command line options for lsh and lshd don't let you specify arbitrary lists. For instance, you can't specify different preferences for sending and receiving.

There is a set of default algorithm preferences. When you use a command line option to say that you want to use *algorithm* for one of the algorithms, the default list is replaced with a list containing the single element *algorithm*. For example, if you use `-c arcfour` to say that you want to use `arcfour` as the encryption algorithm, the connection will either end up using `arcfour`, or algorithm negotiation will fail because the peer doesn't support `arcfour`.

Option	Algorithm type	Default	
`-z`	Data compression	`none`, `zlib`	The default preference list supports zlib compression, but prefers not to use it.
`-c`	Encryption	`aes256-cbs`, `3dec-cbc`, `blowfish-cbc`, `arcfour`	The default encryption algorithm is aes256. The default list includes only quite old and well studied algorithms. There is a special algorithm name `all` to enable all supported encryption algorithms (except `none`).
`-m`	Message Authentication	`hmac-sha1`, `hmac-md5`	Both supported message authentication algorithms are of the HMAC family.

As a special case, `-z` with no argument changes the compression algorithm list to `zlib`, `none`, which means that you want to use `zlib` if the other end supports it. This is different from `-zzlib` which causes the negotiation to fail if the other end doesn't support `zlib`. A somewhat unobvious consequence of `-z` having an *optional* argument is that if you provide an argument, it must follow directly after the option letter, no spaces allowed.

Node: Hostauth options, Next: Userauth options, Previous: Algorithm options, Up: Invoking lsh

Host authentication options

As described earlier (see Threats), proper authentication of the remote host is crucial to protect the connection against man-in-the-middle attacks. By default, `lsh` verifies the server's claimed host key against the *Access Control Lists* in `~/.lsh/host-acls`. If the remote host cannot be authenticated, the connection is dropped.

The options that change this behaviour are

`--host-db`
: Specifies the location of the ACL file.

`--sloppy-host-authentication`
: Tell `lsh` not to drop the connection if the server's key can not be authenticated. Instead, it displays the fingerprint of the key, and asks if it is trusted. The received key is also appended to the file `~/.lsh/captured_keys`. If run in quiet mode, `lsh -q --sloppy-host-authentication`, `lsh` connects to any host, no questions asked.

`--strict-host-authentication`
: Disable sloppy operation (this is the default behaviour).

`--capture-to`
: Use some other file than `~/.lsh/captured_keys`. For example,

```
lsh --sloppy-host-authentication --capture-to ~/.lsh/host-acls
```

: makes `lsh` behave more like the `ssh` program.

`--srp-keyexchange`
: Try using SRP for keyexchange and mutual authentication.

Node: Userauth options, Next: Action options, Previous: Hostauth options, Up: Invoking lsh

User authentication options

`-l`
: Provide a name to use when logging in. By default, the value of the `LOGNAME` environment variable is used.

`-i`
: Try the keys from this file to log in. By default, `lsh` uses `~/.lsh/identity`, if it exists. It ought to be possible to use several `-i` options to use more than one file, but that is currently not implemented.

`--no-publickey`
: Don't attempt to log in using public key authentication.

Node: Action options, Next: Verbosity options, Previous: Userauth options, Up: Invoking lsh

Action options

There are many things `lsh` can do once you are logged in. There are two types of options that control this: *actions* and *action modifiers*. For short options, actions use uppercase letters and modifiers use lowercase.

For each modifier `--foo` there's also a negated form `--no-foo`. Options can also be negated by preceding it with the special option `-n`. This is mainly useful for negating short options. For instance, use `-nt` to tell `lsh` not to request a remote pseudo terminal. Each modifier and its negation can be used several times on the command line. For each action, the latest previous modifier of each pair apply.

First, the actions:

`-L`
: Requests forwarding of a local port. This option takes a mandatory argument of the form *listen-port*:*target-host*:*target-port*. This option tells `lsh` to listen on *listen-port* on the local machine. When someone conects to that port, `lsh` asks the remote server to open a connection to *target-port* on *target-host*, and if it succeeds, the two connections are joined together through an the `lsh` connection. Both port numbers should be given in decimal.

`-R`
: Requests forwarding of a remote port. It takes one mandatory argument, just like `-L`. But in this case `lsh` asks the *remote* server to listen on *listen-port*. When someone connects to the remote hosts, the server will inform the local `lsh`. The local `lsh` then connects to *target-port* on *target-host*.

`-D`
: Requests SOCKS-style forwarding. It takes one optional argument, the port number to use for the SOCKS proxy (default is 1080). Other applications can then use socks version 4 or version 5, to open outgoing connections which are forwarded via the SSH connection. Note that for short options the port number must be in the same argument if given (i.e. `-D1080` is correct, `-D 1080` is not).

`-E`
: This option takes one mandatory argument, which is a command line to be executed on the remote machine.

`-S`
: Start an interactive shell on the remote machine.

`-G`
: Open a gateway on the local machine. A gateway is a local socket, located under /tmp, that can be used for controlling and using the ssh connection. It is protected using the ordinary file permissions.

`-N`
: This is a no-operation action. It inhibits the default action, which is to start an interactive shell on the remote machine. It is useful if you want to set up a few forwarded tunnels or a gateway, and nothing more.

`-B`
: Put the client into the background after key exchange and user authentication. Implies `-N`

`--subsystem`
: Specifies a subsystem to connect to, implies `--no-pty`. Example usage: `--subsystem=sftp`

If there are trailing arguments after the name of the remote system, this is equivalent to a `-E` option, with a command string constructed by catenating all the remaining arguments, separated by spaces. This implies that the arguments are usually expanded first by the local shell, and then the resulting command string is interpreted again by the remote system.

If there are no trailing arguments after the name of the remote system, and the `-N` option is not given, the default action is to start a shell on the remote machine. I.e. this is equivalent to the `-S` option.

There are a few supported modifiers:

`-t`
: Request a pseudo terminal. `lsh` asks the remote system to allocate a pseudo terminal. If it succeeds, the local terminal is set to raw mode. The default behaviour is to request a pty if and only if the local `lsh` process has a controlling terminal. This modifier applies to actions that create remote processes, i.e. `-E` and `-S`, as well as the default actions.

  Currently, this option is ignored if there is no local terminal.

`-x`
: Request `X` forwarding. Applies to the `-E` and `-S` and the default actions.

`--stdin`
: Redirect the stdin of a remote process from a given, local, file. Default is to use `lsh`'s stdin for the first process, and `/dev/null` for the rest. This option applies to the `-E` and `-S` options as well as to the default actions. The option applies to only one process; as soon as it is used it is reset to the default.

`--stdout`
: Redirect the stdout of a remote process to a given, local, file. Default is to use `lsh`'s stdout. Like `--stdin`, it is reset after it is used.

`--stderr`
: Redirect the stdout of a remote process to a given, local, file. Analogous to the `--stdout` option.

`--detach`
: Detach from terminal at session end.

`--write-pid`
: Applies to `-E`. Write PID of backgrounded process to stdout.

`-e`
: Set the escape character (use "none") to disable. Default is "~" if a tty is allocated and "none" otherwise.

`-g`

Remote peers, aka global forwarding. This option applies to the forwarding actions, i.e. `-L`, `-R` and `-D`. By default, only connections to the loopback interface, ip 127.0.0.1, are forwarded. This implies that only processes on the same machine can use the forwarded tunnel directly. If the -g modifier is in effect, the forwarding party will listen on *all* network interfaces.

Node: Verbosity options, Previous: Action options, Up: Invoking lsh

Verbosity options

These options determines what messages `lsh` writes on its stderr.

`-q`
: Quiet mode. Disables all messages and all questions, except password prompts and fatal internal errors.

`-v`
: Verbose mode. Makes `lsh` a little more verbose. The intention is to provide information that is useful for ordinary trouble shooting, and makes sense also to those not familiar with `lsh` internals.

`--trace`
: Trace mode. Prints some internal information to aid tracking `lsh`'s flow of control.

`--debug.`
: Debug mode. Dumps *a lot* of information, including dumps of all sent and received packets. It tries to avoid dumping highly sensitive data, such as private keys and the contents of `SSH_MSG_USERAUTH_REQUEST` messages, but you should still use it with care.

`--log-file`
: This option redirects all messages to a file. Takes one mandatory argument: The name of that file.

Note that all these options are orthogonal. If you use `--trace`, you usually want to add `-v` as well; `--trace` does not do that automatically.

Node: Invoking lshg, Next: Invoking lshd, Previous: Invoking lsh, Up: Top

Invoking `lshg`

You use `lshg` to login to a remote machine to which you have previously used `lsh` to set up a gateway (see Action options). Its usage is very similar to that of `lsh` (see Invoking lsh), except that some options are not available.

Basic usage is

`lshg [-l` *username*`]` *host*

which attempts to connect to the gateway that should previously have been established by running `lsh [-l` *username*`] -G` *host*)

The *username* and *host* are used to locate the gateway. The default value for *username* is determined in the same way as for `lsh` (see Invoking lsh).

As `lshg` uses almost the same options as `lsh` (see Invoking lsh), only options that are not available or have a different meaning in `lshg` are listed here.

The algorithm options (see Algorithm options) as well as most of the userauth (see Userauth options) and hostauth (see Hostauth options) are not available in `lshg` as they are only used by session setup, which is already handled by `lsh`.

Due to technical reasons, X11-forwarding cannot be performed by `lshg`, thus the `--x11-forward` option (see Action options) is not available.

To summarize, these are the options that are new, not available or that have different meanings:

`-G`
: For `lsh -G` requests a gateway to be set up. For `lshg` it means that if no usable gateway is found `lsh` should be launched with the same arguments instead.

`--send-debug`
: Not available in `lsh`. Sends a `debug` message to the remote machine.

`--send-ignore`
: Not available in `lsh`. Sends a `ignore` message to the remote machine.

`-x`
: (`--x11-forward`) Not available in `lshg`.

`-c`
: (`--crypto`) Not available in `lshg`.

`-z`
: (`--compression`) Not available in `lshg`.

`-m`
: (`--mac`) Not available in `lshg`.

`--hostkey-algorithm`
: Not available in `lshg`.

`--capture-to`
: Not available in `lshg`.

`--strict-host-authentication`
: Not available in `lshg`.

`--sloppy-host-authentication`
: Not available in `lshg`.

`--host-db`
: Not available in `lshg`.

`--publickey`
: Not available in `lshg`.

`--no-publickey`
: Not available in `lshg`.

`--dh-keyexchange`
: Not available in `lshg`.

`--no-dh-keyexchange`
: Not available in `lshg`.

`--srp-keyexchange`
: Not available in `lshg`.

`--no-srp-keyexchange`
: Not available in `lshg`.

`-i`
: `--identity` Not available in `lshg`.

Node: Invoking lshd, Next: Files and environment variables,
Previous: Invoking lshg, Up: Top

Invoking `lshd`

`lshd` is a server that accepts connections from clients speaking the Secure Shell Protocol. It is usually started automatically when the systems boots, and runs with root privileges. However, it is also possible to start `lshd` manually, and with user privileges.

There are currently no configuration files. Instead, command line options are used to tell `lshd` what to do. Many options have `--foo` and `--no-foo` variants. Options specifying the default behaviour are not listed here.

Some of the options are the shared with `lsh`. In particular, see Algorithm options and Verbosity options.

Options specific to the `lshd` server are:

`-p`
: Port to listen to. The mandatory argument is a decimal port number or a service name. Default is "ssh", usually port 22.

  It should also be possible to use several -p options as a convenient way to make lshd listen on several ports on each specified (or default) interface, but that is not yet implemented.

  Note that if you use both `-p` and `--interface`, the order matters.

`--interface`
: Network interface to listen on. By default, `lshd` listens on all interfaces.

An interface can be specified as a DNS name, a literal IPv4 address, or a literal IPv6 address enclosed in square brackets. It can optionally be followed by a colon and a port number or service name. If no port number or service is specified, the default or the value from a *preceding* `-p` is used.

Some examples: `--interface=localhost`, `--interface=1.2.3.4:443`, `--interface=[aaaa::bbbb]:4711`. To make `lshd` listen on several ports and interfaces at the same time, just use several `--interface` options on the command line.

`-h`
: Location of the server's private key file. By default, `/etc/lsh_host_key`.

`--daemonic`
: Enables daemonic mode. `lshd` forks into the background, redirects its stdio file descriptors to `/dev/null`, changes its working directory to `/`, and redirects any diagnostic or debugging messages via syslog.

  `lshd` should be able to deal with the environment it inherits if it is started by `init` or `inetd`, but this is not really tested.

`--pid-file`
: Creates a locked pid file, to make it easier to write start and stop scripts for `lshd`. The mandatory argument provides the filename. This option is enabled by default when operating in daemonic mode, and the default filename is `/var/run/lshd.pid`.

`--no-syslog`
: Disable the use of the syslog facility. Makes sense only together with `--daemonic`

`--enable-core`
: By default, `lshd` disables core dumps, to avoid leaking sensitive information. This option changes that behaviour, and allows `lshd` to dump core on fatal errors.

`--no-password`
: Disable the "password" user authentication mechanism.

`--no-publickey`
: Disable the "publickey" user authentication mechanism.

`--root-login`
: Enable root login. By default, root can not log in using `lshd`.

`--login-auth-mode`
: This option is highly experimental. Bypass `lshd`'s user authentication, and allow users to spawn their login-shell without any authentication. Usually combined with `--login-shell`, to set the login shell to a program that performce password authentication.

`--kerberos-passwords`
: Verify passwords against the kerberos database. This is implemented using the `lsh-krb-checkpw` helper program. Note that this does *not* use the Kerberos infrastructure in the Right Way. Experimental.

`--password-helper`

Tells `lshd` to use a helper program for verifying passwords. This is a generalization of `--kerberos-passwords`, and it could be used for verifying passwords against any password database. See the source files `lsh-krb-checkpw.c` and `unix_user.c` for details.

`--login-shell`
: Use the specified program as the login shell for all users, overriding the login shell in the passwd database.

`--srp-keyexchange`
: Enable SRP keyexchange and user authentication.

`--no-pty-support`
: Disable support for pseudo terminals.

`--no-tcp-forward`
: Disable support for tcp forwarding, in both directions.

`--subsystems`
: Specifies a list of subsystems and corresponding programs. Example usage: `--subsystems=sftp=/usr/sbin/sftp-server,foosystem=/usr/bin/foo`

Node: Files and environment variables, Next: Terminology,
Previous: Invoking lshd, Up: Top

Files and environment variables

This chapters describes all files and all environment variables that are used by `lsh`, `lshd`, and related programs.

There are a few environment variables that modifies the behaviour of the `lsh` programs. And there are also a handful of variables that are setup by `lshd` when starting user processes.

`DISPLAY`
: When X-forwarding is enabled, `DISPLAY` specifies the local display. Used by `lsh`.

`HOME`
: User's home directory. Determines where client programs looks for the `~/.lsh` directory. When `lshd` starts a user program, it sets `HOME` from the value in the `/etc/passwd` file, except if `lshd` is running as an ordinary user process. In the latter case, the new process inherits `lsh`'s own value of `HOME`.

`LOGNAME`
: The user's log in name. Used as the default name for logging into remote systems. Set by `lshd` when starting new processes.

`LSH_YARROW_SEED_FILE`
: If set, it points out the location of the seed-file for the randomness generator. Recognized both by `lshd` and the client programs.

`LSHFLAGS`
: If set, lsh will parse any options as had they been given on the command

line.

`LSHGFLAGS`
: If set, lshg will parse any options as had they been given on the command line.

`POSIXLY_CORRECT`
: Affects the command line parsing of programs which by default accept options mixed with arguments.

`SEXP_CONV`
: The location of the `sexp-conv` program. If not set, the default `$prefix/bin/sexp-conv` is used.

`SSH_CLIENT`
: This variable may be set by lshd for established sessions. If it is set it consists of three parts separated by whitespace, the first part contains the address of the connecting client. The second part contains the tcp port used on the connecting client and the third part contains the tcp port used on the server.

`SSH_TTY`
: This variable may be set by lshd for established sessions. If it is set it is the name of the tty allocated.

`SHELL`
: User's login shell. When `lshd` starts a user process, it sets `SHELL` to the value in `/etc/passwd`, unless overridden by the `--login-shell` command line option.

`TERM`
: The type of the local terminal. If the client requests a pty for a remote process, the value of `TERM` is transferred from client to server.

`TMPDIR`
: Determines where the unix socket used by `lshg` is located in the filesystem.

`TZ`
: Time zone. Processes started by `lshd` inherit the value of this variable from the server process.

Files used by the lsh client, stored in the `~/lsh` directory:

`captured_keys`
: Keys for remote hosts, saved when running `lsh --sloppy-host-authentication`. Or more precicely, each key is stored together with an as SPKI (Simple Public Key Intrastructure) ACL:s (Access Control Lists).

`identity`
: Your private key file. Usually created by `lsh-keygen | lsh-writekey`. Read by `lsh`. Should be kept secret.

`identity.pub`
: The corresponding public key. You can copy this file to other systems in order to authorize the private key to login (see Converting keys).

`host-acls`
: Host keys (or more precisely, ACL:s) that lsh considers authentic.

Entries have the same format as in `captured_keys`.

`yarrow-seed-file`
: The seed file for the randomness generator. Should be kept secret.

Files used by `lshd`, some of which are read from user home directories:

`/etc/lsh_host_key`
: The server's private host key.

`/etc/lsh_host_key.pub`
: The corresponding public key.

`/var/spool/lsh/yarrow-seed-file`
: The seed-file for `lshd`'s randomness generator.

`~/.lsh/authorized_keys`
: This is a directory that keeps a "database" of keys authorized for login. With the current implementation, a key is authorized for login if and only if this directory contains a file with a name which is the SHA1 hash of the key. The usual way to create files is by running the script `lsh-authorize`.

`~/.lsh/srp-verifier`
: If you use the experimental support for SRP (see srp), the server reads a user's SRP verifier from this file.

Node: Terminology, Next: Concept Index, Previous: Files and environment variables, Up: Top

Terminology

Node: Concept Index, Previous: Terminology, Up: Top

Concept Index