

Device tools

Timo Savola

tsavola@movial.fi

Device tools
by Timo Savola

Revision history

Version:	Author:	Description:
2004-05-15	Savola	Initial version
2004-05-17	Savola	sbrsh updates
2004-06-07	Savola	Fixed typos

Table of Contents

1. Introduction.....	1
2. sbrsh.....	2
2.1. Installing the server.....	2
2.1.1. Compiling the server.....	3
2.2. Configuring user accounts.....	4
2.3. Usage.....	4
2.4. Debugging.....	5
3. fakeroot-net.....	7
3.1. Installing.....	7
3.1.1. Compiling.....	8
3.2. Known issues.....	8
3.3. Debugging.....	9
4. Implementation of networked fakeroot.....	10
Bibliography.....	11
A. sbrsh reference.....	12
A.1. Client usage.....	12
A.2. Daemon usage.....	12
A.3. Client configuration file.....	13
A.4. Daemon configuration file.....	13
A.5. Environment variables.....	14

Chapter 1. Introduction

Scratchbox[1] contains two custom-made tools that help in running programs built for non-native architectures: *sbrsh* and *fakeroot-net*. They consist of parts that are compiled for the host system and the target system.

The **scratchbox-core** package provides the host-side tools and the **scratchbox-devicetools** package provides target-side binaries compiled with all toolchains. See *Installing Scratchbox*[2] for instructions about obtaining these packages.

sbrsh and *fakeroot-net* are licensed under the *GNU General Public License*[3]. Their source code repositories can be found at the *ViewCVS* section on the Scratchbox website[1].

Chapter 2. sbrsh

The *Scratchbox Remote Shell* is a remote command execution system similar to *rsh* and *ssh*. It is designed with slow devices and *Scratchbox*'s special requirements in mind. It supports common types of program execution (including terminal emulation), but it is optimized for non-interactive usage. The communication happens on a TCP/IP connection and is not encrypted—*sbrsh* is meant to be used only on trusted networks (such as a company's LAN or an USB network between a PC and a handheld device).

The server daemon (*sbrshd*) is run on a device having the same CPU architecture as the compilation target that is being used in *Scratchbox*. It executes the commands issued by the client (*sbrsh*) inside a "sandbox" that is created by mounting network filesystems (typically exported by the host that runs the client) and binding local directories (such as `/dev`).

sbrshd contains support functionality that makes remote *fakeroot*[4] sessions possible (this is described in Chapter 4). Due to this, *sbrsh* has a compile time dependency to *fakeroot-net*.

2.1. Installing the server

The target device should be running some kind of more-or-less standard Linux installation with kernel version 2.4 or newer. *sbrshd* has been tested with *Familiar*[5] and *EE*[6] distributions using iPAQ hardware.

Only the **sbrshd** binary is needed to be installed, but its init script is also recommended. If you have installed the **scratchbox-devicetools** package, you can copy a binary suitable for your target system and the init script from the `/scratchbox/device_tools/sbrsh-version/toolchain/` directory. If there isn't a binary available that matches your system (for example C library version is different), you can try a statically linked one. There is a statically linked version in the `/scratchbox/device_tools/sbrsh-version/cpu-linux-static/` directory for each CPU architecture that has a *uClibc*[7] toolchain. Finally, you can compile *sbrshd* by yourself if you have a (cross-)compiler for your target system. See Section 2.1.1 for instructions.

There are a few things to know before starting the daemon. *sbrshd* uses the **mount** command to bind parts of the directory tree to other locations. The **mount** command provided by the standard *util-linux* software package uses the "`--bind`" option for this, but this may not be the case with all versions of **mount**. For example, *Busybox*[8] uses the "`-obind`" option instead. *sbrshd* tries to auto-detect if the **mount** binary is a symlink to **busybox**, but sometimes it may be necessary to pass the correct bind option to *sbrshd* as the parameter of the "`-b`" option.

sbrshd also needs a list of valid login shells in order to verify that users are authorized to login to the target device. The preferred way is to provide a `/etc/shells` file that lists the paths to the shells. The alternative is to pass a list of paths to the "`-s`" option. The init script provides a list of a few common shells if `/etc/shells` is not available.

Finally: sbrshd needs to be started as **root**. It needs super-user privileges for mounting and for *chrooting* to the “sandbox” directory. The command execution is done under the user and group IDs of the user account that is used to execute the command.

If you installed the init script to `/etc/init.d/`, you can probably make the system run it automatically during boot (refer to your system’s documentation). You can use the script manually to start and stop sbrshd:

```
# /etc/init.d/sbrshd start
Starting Scratchbox Remote Shell daemon: done.
# /etc/init.d/sbrshd stop
Stopping Scratchbox Remote Shell daemon: done.
```

Note that by default the init script looks for **sbrshd** from `/sbin/`. You should edit the init script if you want to pass sbrshd some additional options. If you want to run sbrshd manually, you can simply do:

```
# sbrshd
3965
```

It goes automatically to daemon mode (starts a background process and returns to shell immediately). It prints out the process ID of the daemon, which can be used to kill it later on.

See Section A.2 for a full list of command-line options.

2.1.1. Compiling the server

You can skip this section if you are using a prebuilt sbrshd binary.

The `sbrsh-version.tar.gz` source package provides both **sbrsh** and **sbrshd** programs. Compilation of sbrshd requires fakeroot-net’s header files which are provided by the `fakeroot-net-version.tar.gz` source package. Both of these packages are provided by the **scratchbox-core** package in the `/scratchbox/packages/` directory.

You can compile sbrshd inside or outside Scratchbox. The following procedure should work in most cases:

1. Extract the sbrsh and fakeroot-net source packages.

```
$ tar xzf /scratchbox/packages/sbrsh-version.tar.gz
$ tar xzf /scratchbox/packages/fakeroot-net-version.tar.gz
```

2. Go to the sbrsh source directory. Create a link called `fakeroot-net` that points to the directory that contains fakeroot-net’s header files.

```
$ cd sbrsh-version
```

```
$ ln -s ../fakeroot-net-version fakeroot-net
```

3. Build the **sbrshd** binary. You may need to explicitly specify the compiler you want to use.

```
$ make sbrshd CC=sparc-linux-gcc
```

2.2. Configuring user accounts

The target device must have normal user accounts (i.e. not **root**) for executing commands via sbrsh. Each user must have a `.sbrshd` (server-side) configuration file in her home directory that is used by the sbrsh daemon for authentication. It lists all IP addresses (not hostnames) of the hosts that the user uses to connect to the device and corresponding passwords. The passwords are not encrypted at any point, so do not use any important ones. The `.sbrsh` file format is described in Section A.3.

Each user account of the Scratchbox installation has separate `.sbrsh` (client-side) configuration file that corresponds to the server-side one. It is located in the user's home directory *inside the Scratchbox sandbox*. It lists some or all of the user's compilation targets. The target settings include the IP address and port of the server, the password that matches the one in `.sbrshd` and a list of network and local filesystems that make up the sandbox. The `.sbrshd` file format is described in Section A.4.

The filesystem configuration needs to recreate the environment that exists inside the Scratchbox sandbox, with a few exceptions. The sandbox on the target device should use its “native” `/proc`, `/dev` and `/dev/pts` directories for things to work. It is also customary to use the device's `/tmp` directory. The order of mounting matters; the filesystem that becomes the sandbox's root must be listed first in the configuration, and so on. The configuration that should be used with a typical Scratchbox installation is described in *Installing Scratchbox*[2].

2.3. Usage

sbrsh is normally invoked implicitly by Scratchbox's *CPU-transparency* feature, but it can also be used manually. It needs to know which target should be used and which program should be executed, but not much more. However, it may be a good idea to pass the current working directory for it with the “`-d`” option:

```
> sbrsh MY-TARGET -d $PWD ./hello
hello world
```

One notable difference to *ssh* is that *sbrsh* doesn't use the shell to execute commands, and thus does not read any profile or resource files. It creates the target environment by copying variables from the source environment.

Sometimes you may not want to pass some architecture-specific variables for the target binaries or want to change them. *sbrsh* can override or unset variables in the remote end by using variables prefixed with "SBOX_ENV_". The following example should illustrate the behaviour:

```
> export SBOX_ENV_FOO=bar
> sbrsh MY-TARGET env | grep ^FOO
FOO=bar
> export SBOX_ENV_PATH=(UNSET)
> sbrsh MY-TARGET env
sbrsh server: Can't execute command: env (No such file or directory)
```

sbrsh doesn't copy the resource limit settings to the remote environment, but they can be tuned with environment variables of the form "SBRSH_RLIMIT_*resource*":

- Limit total CPU time of the process to 10 seconds:


```
> export SBRSH_RLIMIT_CPU=10
```
- Always create core dumps:


```
> export SBRSH_RLIMIT_CORE=unlimited
```
- Run a program under these conditions:


```
> sbrsh MY-TARGET gnomovision
```

See Appendix A for a full list of command-line options and environment variables.

2.4. Debugging

The daemon reports all error conditions to syslog. It also supports a debug log where it writes lot of information about its state and what happens during command execution. It can be enabled with the "-d" command-line option, which takes the log filename as its parameter.

sbrshd does not have to be restarted in order to enable logging: it will open the log when it receives the `USR1` signal. Logging can also be turned off with the `USR2` signal. If a log filename was not specified

with the “-d” option (i.e. logging was not initially enabled), the log will be written to `/tmp/sbrshd-port.log` (“*port*” being the listening port of the daemon—1202 by default).

Here is an example session where the debug log is written to the terminal device:

```
# sbrshd -d `tty`
3992
01-06-1970 03:41:45.504 3992 DAEMON Debugging enabled
01-06-1970 03:41:45.506 3992 DAEMON sbrshd version 1.4.4 (protocol version 4)
01-06-1970 03:41:45.506 3992 DAEMON Listening at port 1202
01-06-1970 03:41:45.506 3992 DAEMON Valid login shells: /bin/sh /bin/bash
/bin/zsh /bin/ash /bin/tcsh
01-06-1970 03:41:45.507 3992 DAEMON Mounts expire after 900 seconds
01-06-1970 03:41:45.507 3992 DAEMON Waiting for connection
# kill -USR2 3992
01-06-1970 03:42:11.513 3992 DAEMON User defined signal 2
01-06-1970 03:42:11.513 3992 DAEMON Debugging disabled
# kill -USR1 3992
01-06-1970 03:42:14.370 3992 DAEMON Debugging enabled
01-06-1970 03:42:14.371 3992 DAEMON Checking for expired mounts
01-06-1970 03:42:14.371 3992 DAEMON Waiting for connection
```

The type of the process is displayed next to the process ID in the log. sbrshd has four types of processes that write log entries: DAEMON accepts connections and handles mounts, HANDLER handles I/O between the client and the command process, RELAY relays fakeroot messages and COMMAND “bootstraps” and executes the command.

Chapter 3. fakeroot-net

fakeroot[4] is a utility that can run programs in an environment that looks as if they were run with super-user privileges. It is used primarily for setting file ownerships and modes before packaging them. You can for example create device nodes and store them in a tarball while logged in as a normal user. Of course, the programs run from a fakeroot session cannot really do privileged system calls; fakeroot keeps an in-memory database of file ownerships and such things.

The command execution inside Scratchbox can jump from host to target (via *sbrsh*) within a fakeroot session. Since both ends use the same filesystems (via *NFS*), they must also use the same fakeroot session. This is not possible with the original design, so a modified version of fakeroot with the cheesy name *fakeroot-net* was developed. *fakeroot-net* uses TCP/IP sockets for its internal communication, but that alone isn't enough. The *sbrsh* server is used to filter the information passed between the remote fakeroot environment and the fakeroot daemon (*faked*) that keeps the database. More on this in Chapter 4.

3.1. Installing

The **scratchbox-core** package provides a *fakeroot-net* installation that can be used to start fakeroot sessions inside Scratchbox:

```
> fakeroot
# touch newfile
# ls -l
-rw-rw-r--  1 root  root           0 May 15 13:01 newfile
```

However, any target binaries are not installed by default. If you are using one of the default toolchains and have the **scratchbox-devicetools** package installed, it is very simple to install the fakeroot library for your target:

```
> sbx-config --copy-libfakeroot
Copying libfakeroot from /scratchbox/device_tools/fakeroot-net-1.0.5/arm-linux-gcc-3.3_3.3.2ds5-glibc-2.3.2.ds1/lib to /usr/lib
```

Now you can launch target binaries inside a fakeroot session:

```
> cc -o whoami whoami.c
> fakeroot ./whoami
root
```

If you are using a custom toolchain or want to compile *fakeroot-net* for some other reason, see Section 3.1.1.

3.1.1. Compiling

*You can skip this section if you are using a prebuilt *fakeroot* library.*

Cross-compilation of *fakeroot-net* needs to be done inside Scratchbox since its configuration requires a working *CPU-transparency* environment. Using the target you want to use *fakeroot-net* with, follow these steps to build and install the whole *fakeroot-net* package:

1. Extract the source package. Go to the source directory.

```
> tar xfz /scratchbox/packages/fakeroot-net-version.tar.gz
> cd fakeroot-net-version
```

2. Configure it. Scratchbox's host-side **fakeroot** expects to find the target library from `/usr/lib/`.

```
> ./configure --prefix=/usr
```

3. Build it. Install it.

```
> make
> make install
```

If you are using the *Debian devkit*[9] and want to create a binary package, do this instead of steps (2) and (3):

```
> dpkg-buildpackage -rfakeroot -b
```

3.2. Known issues

The *fakeroot* environment is imposed upon a process by using the C library's "LD_PRELOAD" environment variable. `libfakeroot.so` is preloaded by the dynamic linker whenever it loads a binary. This means that *fakeroot* does not work with statically linked binaries.

There is also another side-effect. Since `libfakeroot.so` is loaded into the same process image with the "victim" program, they share the same file descriptor table. Some programs (such as the **configure**

scripts) use hard-coded descriptor numbers. `libfakeroot.so` needs one file descriptor for its communication socket, and if the program starts to use the same file descriptor, there will be trouble. `fakeroot` tries to monitor the status of its descriptor so that it can open a new socket if the descriptor has been changed. If you start seeing messages about hijacked file descriptors, you can try to make `fakeroot` use some other file descriptor with the “`--fd-base`” option. Its default value is (`descriptor_table_size - 100`).

3.3. Debugging

The `fakeroot` daemon can be launched with debug enabled and left running on the foreground:

```
> faked --debug --foreground
33366:5027
```

The first number is the TCP/IP port it listens to, and the second number is its process ID. Now, in another terminal, setup a `fakeroot` session manually that uses the daemon we started:

```
> export FAKEROOTKEY=33366
> export LD_PRELOAD=/scratchbox/tools/lib/libfakeroot.so.1
```

Now you can run programs in the hand-made `fakeroot` session and see the daemon’s cryptic debug output in the other terminal. This way you can also use `gdb`[10] to debug a program within a `fakeroot` environment.

When using a remote `fakeroot` session, the communication can be traced using the `sbrsh` server’s debug log. See Section 2.4 for instructions.

Chapter 4. Implementation of networked fakeroot

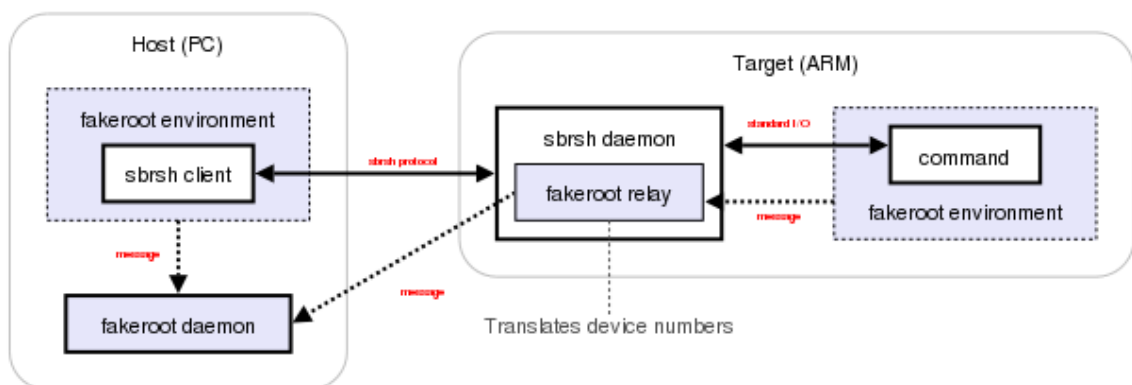
This chapter deals with technical details; it is not requisite for installing or using sbrsh or fakeroot-net.

faked maintains a list of entries based on their device and inode numbers of the files that have been modified during a fakeroot session. The entries contain a data structure that is essentially the same as the one used by the `stat` system call. *fakeroot-net* introduces an additional “`remote`” field in the entry, which works like a “namespace” for the devices and inodes. All files on the local filesystems belong to the default namespace (`remote` is not set).

When a remote command is run within a fakeroot session, **sbrsh** resolves the device numbers of the NFS filesystems that are listed in the `.sbrsh` file for the used target. If they are not exported by the local host but some third host, it tries to find out if the NFS filesystems are mounted on the local host and use the device numbers of the mount points.

sbrshd receives the list of mount entries and finds out what their device numbers are on the target device. Then it creates a “relay” process that listens for connections from local fakeroot sessions. When it receives one, it makes a corresponding connection to the **faked** running on the Scratchbox host. It maintains as many connection pairs as there are processes running within the local fakeroot session. The relay copies messages from the local session to the remote daemon and responses from the daemon to the session, and translates the device numbers in the messages between the local and remote device number “spaces”.

If the relay finds an unlisted device number in one of the incoming messages, it does not translate it but sets the value of the `remote` field to the IP address of the host it is running at. This way **faked** can serve unknown filesystems without the danger of device number/inode collisions.



Bibliography

- [1] *Scratchbox website* (<http://www.scratchbox.org/>).
- [2] *Installing Scratchbox* (<http://www.scratchbox.org/documentation/docbook/installdoc.html>), Valteri Rahkonen.
- [3] *GNU General Public License* (<http://www.gnu.org/licenses/gpl.html>), Free Software Foundation.
- [4] *Debian GNU/Linux—fakeroot* (<http://packages.debian.org/unstable/utils/fakeroot.html>).
- [5] *The Familiar Project website* (<http://familiar.handhelds.org/>).
- [6] *EE-distro - Scratchbox Support Distribution for ARM*
(<http://www.scratchbox.org/documentation/general/eedistro.html>), Karri Niskala.
- [7] *uClibc website* (<http://uclibc.org/>).
- [8] *Busybox website* (<http://busybox.net/>).
- [9] *Debian devkit* (http://www.scratchbox.org/documentation/docbook/devkit_debian.html), Valteri Rahkonen.
- [10] *GDB: The GNU Project Debugger* (<http://www.gnu.org/software/gdb/gdb.html>).

Appendix A. sbrsh reference

A.1. Client usage

Executing remote command:

```
sbrsh target [-c|--config path] [-d|--directory dir] [command [args]]
```

Unmounting all filesystems of a *target* (see Section A.3):

```
sbrsh target [-c|--config path] --umount-all
```

<i>target</i>	symbolic name of the target configuration; must be specified before any options (see Section A.3)
<i>path</i>	the absolute path to the user's configuration file (default is <code>.sbrsh</code> in the home directory)
<i>dir</i>	the user's current directory at the target device (default is the filesystem root)
<i>command</i>	name of the command to be executed (may be looked up from <code>PATH</code>); if command is not specified, the user's login shell at the remote host is executed
<i>args</i>	zero or more arguments for <i>command</i>

A.2. Daemon usage

```
sbrshd [-p|--port port] [-d|--debug log] [-m|--mount-bin mount]
        [-u|--umount-bin umount] [-t|--mount-tab mtab] [-b|--bind-opt opt]
        [-e|--mount-expiration mins|none] [-s|--shells file] [-S|--shell-list list]
```

<i>port</i>	sets a custom port number (default is 1202)
<i>log</i>	enables debugging to a log file
<i>mount</i>	specifies the mount binary path (default is <code>/bin/mount</code>)
<i>umount</i>	specifies the umount binary path (default is <code>/bin/umount</code>)
<i>mtab</i>	specifies the mount table path (default is <code>/proc/mounts</code>)
<i>opt</i>	specifies the option used when binding a path to a mount point (default is “ <code>--bind</code> ”, or “ <code>-obind</code> ” if mount binary is <i>Busybox</i>)

<i>mins</i>	specifies the number of minutes to wait before expiring unused mount points (default is 15); 0 means that filesystems are unmounted immediately after commands exit; “none” means that filesystems are unmounted only when sbrshd exits
<i>file</i>	specifies the path to a file that lists all valid login shells (default is <code>/etc/shells</code>)
<i>list</i>	specifies a colon-separated list of valid login shells; <code>/etc/shells</code> is not read if this is specified

A.3. Client configuration file

sbrsh configuration file lists all known *targets* (see Section A.1). The first line of a *target* block must not contain whitespaces before the name of the *target*. The subsequent lines must be indented. “#” is a line end comment character. The root of the command’s sandbox will be the `'ipaddress-target'` directory under the user’s home directory.

The layout of the first line:

```
target [username@]ipaddress[:port] password
```

The subsequent lines define the mounts needed by the *target* (*type* is either “nfs” or “bind”):

```
type filesystem point [options]
```

Here’s an example configuration:

```
ARM john@10.0.0.3 asdf
  nfs 10.0.0.2:/scratchbox/users/john/targets/ARM / rw,nolock,noac
  nfs 10.0.0.2:/scratchbox/users/john/home /home rw,nolock,noac
  bind /dev /dev
  bind /dev/pts /dev/pts
  bind /proc /proc
  bind /tmp /tmp
```

A.4. Daemon configuration file

sbrshd configuration file lists all known client IP addresses and passwords. Each user has her own `.sbrshd` file in his home directory. “#” is a line end comment character.

The layout is:

```
ipaddress password
```

A.5. Environment variables

The command execution environment at the target device can be controlled via a few environment variables.

“SBOX_ENV_” prefix will be stripped from all variables having one. If a corresponding variable without the prefix exists, it will be overridden. If the variable’s value is “(UNSET)”, the corresponding variable will be removed from environment. For example the dynamic linker can be controlled this way (via the LD_* variables) without affecting the sbrsh client itself.

The resource limits can be set using variables with the “SBRSH_RLIMIT_” prefix. The value can be either an integer or “unlimited”. The supported settings are:

SBRSH_RLIMIT_CPU	CPU time in seconds
SBRSH_RLIMIT_FSIZE	max filesize
SBRSH_RLIMIT_DATA	max data size
SBRSH_RLIMIT_STACK	max stack size
SBRSH_RLIMIT_CORE	max core file size
SBRSH_RLIMIT_RSS	max resident set size
SBRSH_RLIMIT_NPROC	max number of processes
SBRSH_RLIMIT_NOFILE	max number of open files
SBRSH_RLIMIT_MEMLOCK	max locked-in-memory address space
SBRSH_RLIMIT_AS	address space (virtual memory) limit