Single User Secure Shell

Adrian Steinmann

***Abstract*—Unix systems traditionally do integrity checks and other initialization before bringing up network services. System administration tasks, like operating system upgrades, system disk reformatting or partitioning, very often need to be done in single user mode on a console, and not via the network. We describe how a 'Secure Shell Maintenance RAMdisk Environment' can be built and launched very early in the boot process. This environment can be used to remotely fix a problem when the machine is stuck in single user mode. Our method has already been in use for a number of years to upgrade remote managed firewall systems [3] from one release to the next.**

I. Introduction

ANYONE who has needed to unexpectedly commute to a production machine stuck in single user mode has wished that it would possibly allow a remote SSH login. In fact, in most cases the problem in question does not require network access to be blocked. It is much more a policy decision that the system first checks the root filesystem and runs other early startup scripts which may stall before launching network services.

In this paper we describe a way to build a 'Secure Shell Maintenance RAMdisk Environment' which can be started even before the root filesystem is checked. This environment can be useful in many different situations:

Root filesystem fails to check: When a system crashes, it may damage the root filesystem so that it cannot be automatically fixed. Traditionally, the system then stays in single user mode awaiting input on the console.

System partitions need to be resized: As more software is installed, the operating system partitions `/`, `/usr`, or `/var` occasionally need to be resized. This usually calls for a `dump`, `bsdlabel`, `newfs`, `restore` cycle, which is difficult or impossible while the system is running in multi user mode.

Operating system upgrade via a clean install: As is the case for the FreeBSD 4.x to 5.x migration, it may be desirable to `newfs` all the system partitions to take advantage of new features or simply to do a 'clean' install.

Root filesystem should be on a RAID: Even in single user mode, it is difficult to transform an already installed operating system onto a GEOM-based RAID because the system is using the non-RAID devices. Similarly, `atacontrol create` will fail on the disk with the system partitions because they are busy.

Minimal installations on small systems: Full installations on small machines – so-called *embedded systems* – with only compact flash (PC-Engines [1], Soekris [2]) from a standard distribution or via CD may not be practicable.

II. How to build the Secure Shell RAMdisk Image

SMALL size of the Secure Shell Maintenance RAMdisk filesystem is the prime concern, yet it should include all the important tools that are needed for remote system administration, in particular a SSHv2 daemon and its required configuration files. In earlier releases, we managed to fit a gzipped kernel and a gzipped RAMdisk image onto one 1.44 MB floppy by using 'small' versions of tools (see, for example `ports/shells/sash` and `src/release/picobsd/tinyware/` as well as SSHv1 `ports/security/ssh`).

Adrian Steinmann is founder of Webgroup Consulting AG, CH-8032 Zürich, Switzerland, <ast@webgroup.ch>

As of FreeBSD release 5.x, fitting everything on one floppy became impossible, and in fact most systems deployed nowadays do not even carry a floppy drive. Nonetheless, the Secure Shell Maintenance RAMdisk Environment is still viable because even systems with only 64MB of compact flash have ample space to store such a RAMdisk image alongside a whittled-down FreeBSD distribution.

A. *Use* `crunchgen` *to minimize RAM utilization*

In our implementation, the following programs are available in the RAMdisk environment:

```
RAMdisk# ls /bin
-sh              ex               kldconfig        red
[                expr             kldload          restore
atacontrol       fastboot         kldstat          rm
badsect          fasthalt         kldunload        rmdir
boot0cfg         fdisk            ldconfig         route
bsdlabel         fsck             link             rrestore
bunzip2          fsck_4.2bsd      ln               scp
bzcat            fsck_ffs         ls               sed
bzip2            fsck_ufs         mdconfig         sh
camcontrol       gbde             mdmfs            sleep
cat              gconcat          mini_crunch      slogin
chflags          geom             mkdir            ssh
chgrp            ggatec           mknod            sshd
chmod            ggated           mount            stty
chown            ggatel           mount_cd9660     swapctl
chroot           glabel           mount_devfs      swapoff
clri             gmirror          mount_fdescfs    swapon
cp               gnop             mount_linprocfs  sync
date             graid3           mount_nfs        sysctl
dd               gshsec           mount_procfs     tar
df               gstripe          mount_std        test
dhclient         gunzip           mv               touch
dhclient-script  gzcat            newfs            tset
diskinfo         gzip             pax              tunefs
disklabel        halt             ping             umount
dmesg            hostname         ps               unlink
du               ifconfig         pwd              vi
dump             init             rdump            zcat
dumpfs           kenv             realpath
ed               kill             reboot

RAMdisk# du -k /bin/*
2928    /bin/-sh
8       /bin/dhclient-script
```

First and foremost, `sshd`, `ssh`, `scp` as well as the network filesystem utilities `mount_nfs`, `ggated`, and `ggatec` are present with the requisite network configuration utilities `ifconfig`, `route`, and `dhclient`. The standard archiving tools `dump`, `restore`, `tar`, and `pax` with `gzip` and `bzip2` are also available. Furthermore, since this is primarily an environment for system administration, the low-level `fdisk`, `bsdlabel`, `newfs`, and `tunefs` utilities are included, with the added luxury of the `vi` editor (albeit with a small `termcap` file supporting only `xterm`, `screen`, `vt220`, `at386`, and `cons25` terminals). Finally, `atacontrol`, `camcontrol`, GEOM-based RAID, GBDE, and DHCP are supported when the underlying kernel is adequately configured.

B. *Use* `mdconfig` *to create a RAMdisk image*

The script `src/release/scripts/doFS.sh` takes a filesystem hierarchy and creates a file containing a RAMdisk image of it. For example, `src/release/Makefile` uses it to create the 'install' and 'fixit' environments on the standard FreeBSD distribution CD.

This RAMdisk image can be gzipped and placed, say, into a `/boot/maint/` subdirectory, where it can be accessed at boot time.

C. *Use the loader to boot into RAMdisk*

When FreeBSD boots, one can enter the `/boot/loader` environment on the console by choosing *6. Escape to loader prompt* at the 'beastie' menu:

```
OK ls /boot/maint
/boot/maint
    k.CUSTOM.gz
    fs_img.gz
    params
    loader.rc
OK load -t md_image /boot/maint/fs_img
OK set vfs.root.mountfrom=ufs:/dev/md0
OK autoboot
Hit [Enter] to boot immediately, or any other ...
Booting [/boot/kernel/kernel] in 9 seconds...
```

By setting the `vfs.root.mountfrom` variable, the kernel mounts the RAMdisk as the root filesystem instead of the one found in `/etc/fstab`.

III. Some Minor Hurdles

The implementation of the described plan is straightforward, except for the following minor difficulties:

Crunching SSHv2: The standard build of FreeBSD `sshd` requires many libraries, yet most are unnecessary for the RAMdisk environment. We will show how we can get by with only linking a fraction of those libraries.

Supporting runtime loader in a crunched binary: Some programs require runtime loading; this means we must link some libraries statically and some dynamically – although we are using `crunchgen`.

Personalizing a generic RAMdisk image flexibly: It is better to keep the personalization of the RAMdisk image separate, so that deployment consists in one or two binary files and one or more editable text files.

A. *Crunching SSHv2 without too many libraries*

Even if we specify lots of `NO*` options in the `crunchgen` configuration file (figure 1), the link phase fails because `libpam.a`, among others, is still referenced.

Nevertheless, by also turning off the variables `LIBWRAP`, `USE_PAM`, `HAVE_LIBPAM`, `HAVE_PAM_GETENVLIST`, `HAVE_SECURITY_PAM_APPL_H`, and `XAUTH_PATH` directly in `src/crypto/openssh/config.h`, the link phase is successful for the `crunchgen` fragment in figure 1. One additionally required library (`-lmd`) will be made available as dynamically loadable shared object in the RAMdisk environment.

```
buildopts -DNOPAM -DNOSECURE -DNOCRYPT -DNO_KERBEROS
buildopts -DNONETGRAPH -DNOIPSEC -DNOINET6
buildopts -DNOATM -DNO_IPFILTER -DNO_X
srcdirs /usr/src/secure/usr.bin
srcdirs /usr/src/secure/usr.sbin
progs scp
progs ssh
ln ssh slogin
progs sshd
libs -lssh -lutil -lz -lcrypto -lcrypt
```

Fig. 1. A `crunchgen` configuration file fragment for SSHv2.

B. *Building mostly statically linked crunched binaries*

Although the rest of the `crunchgen` configuration file is straightforward, the `geom` programs require the runtime loader for their *dlopen()* calls. As an added complication, the `/lib/geom/geom_*.so` libraries also expect `libmd.so` to be dynamically loaded.

Mostly statically linked binaries can be built simply by replacing `$(CC) -static ...` in the Makefile created by `crunchgen` with `$(CC) -Xlinker -Bstatic ... -Xlinker -Bdynamic -lmd`, leaving the crunched binary dynamically linked to `libmd.so` and `libc.so`.

For our RAMdisk environment to be functional, we will thus need the `/lib/geom/` shared objects as well as `libmd.so` and `libc.so`:

```
RAMdisk# du -k /lib
116     /lib/geom
1054    /lib

RAMdisk# ls -FR /lib
geom/           libc.so.5       libmd.so.2

/lib/geom:
geom_concat.so  geom_nop.so     geom_stripe.so
geom_label.so   geom_raid3.so
geom_mirror.so  geom_shsec.so
```

To date, we haven't investigated if further dynamical linking of a crunched binary would result in other advantages.

C. *One RAMdisk image for many systems*

Another important design goal is to have one RAMdisk image for all machines, somehow personalizing it via a separate configuraton file. This requirement can be solved by passing information via the kernel environment. The RAMdisk environment can then use `kenv` to configure the network and, in particular, to create the `/root/.ssh/authorized_keys` file there.

```
set maint.ifconfig_XX0="192.168.0.254/24"
set maint.ifconfig_XX1="192.168.1.254/24"
set maint.ifconfig_YY0="dhcp"
set maint.defaultrouter="192.168.0.1"
set maint.host="GENERIC"
set maint.domain="SETME.com"
set maint.sshkey_01a="ssh-d.. (120 chars) ..qP"
set maint.sshkey_01b="leQXQ.. (120 chars) ..9d"
set maint.sshkey_01c="b7Zd+.. (120 chars) ..zu"
set maint.sshkey_01d="KrdBn.. (120 chars) ..tw"
set maint.sshkey_01e="7eMec.. (120 chars) ..4G"
set maint.sshkey_01j="hdTLKVUokhU4lQ== 200507"
```

Fig. 2. A `/boot/maint/params` file describing machine personality.

In our setup, the `/boot/maint/params` file describes the machine personality (see figure 2) and is included by the loader when booting into the RAMdisk environment (figure 4).

A limitation of `kenv` is that the key and value lengths may not exceed 128 characters. Since we need to craft a `/root/.ssh/authorized_keys` file in the RAMdisk environment, we split its contents into smaller pieces and then paste them back together before launching the SSH daemon.

IV. Putting it all together

FOR the 'Single User Secure Shell' to be launched early, the startup script '`sussh`' needs to be placed into the `/etc/rc.d/` directory with the correct REQUIRE and BEFORE keywords:

```
#!/bin/sh
PATH=/rescue:/usr/bin:/bin:/usr/sbin:/sbin
export PATH

# REQUIRE: preseedrandom
# PROVIDE: sussh
# KEYWORD: nojail
# BEFORE:  initdiskless
```

On a standard FreeBSD system, this means it will be started second:

```
5.4-STABLE$ rcorder /etc/rc.d/* | head -3
/etc/rc.d/preseedrandom
/etc/rc.d/sussh
/etc/rc.d/initdiskless
```

To control launching of the Single User Secure Shell, `/etc/rc.conf` has these knobs and tunables:

```
## sussh_enable="NO"
## sussh_mntdir="/boot/maint"
## sussh_fs_img="/boot/maint/fs_img"
## sussh_port="22222"
```

The effect of setting `sussh_enable="YES"` is that at boot time – before practically any other initialization – the `$sussh_fs_img` file is mounted onto `$sussh_mntdir` with `/rescue/mdconfig` and `/rescue/mount` and then the RAMdisk SSH daemon is launched on port `$sussh_port`. Given the correct SSH private key and a correctly configured `/boot/maint/params` file, a root shell can be opened remotely on, say, port 22222, to work in the RAMdisk environment should the machine hang in single user mode.

The additional disk space requirements are modest and hence this method is also very well applicable to embedded devices. A gzipped kernel and a gzipped Secure Shell Maintenance RAMdisk image will be about 3 MB and are integrated into the `/boot` hierarchy (figure 3). Note that the loader can also be gzipped, roughly halving its space requirements. All files which the loader may load can also be gzipped, and the loader automatically searches for `*.gz` files and uncompresses them on-the-fly. Generally we prefer to supply a custom kernel which has the `device md` compiled in and otherwise is stripped of extraneous options not needed in the RAMdisk. Additional non-standard options may be compiled in to support DHCP or `camcontrol`.

Alternatively, instead of launching the `sussh` startup script at boot time, the loader can be instructed to boot directly into the RAMdisk by including the lines in figure 4 into `/boot/loader.rc`. The machine can then be rebooted into the Single User Secure Shell for a console-free system upgrade.

```
5.4-STABLE$ /bin/ls -1sFR /boot
  4 beastie.4th.gz
  2 defaults/
  2 device.hints
  2 frames.4th.gz
  2 kernel/
110 loader*
  4 loader.4th.gz
  2 loader.conf
  6 loader.help.gz
  2 loader.rc
  2 maint/
  2 screen.4th.gz
 10 support.4th.gz

/boot/defaults:
 6 loader.conf.gz

/boot/kernel:
3216 kernel

/boot/maint:
1840 fs_img.gz
1056 k.CUSTOM.gz
   2 loader.rc
   2 params
```

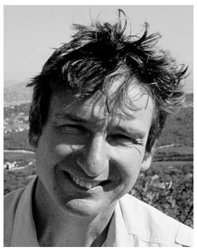
Fig. 3. An example `/boot/` directory hierarchy with gzipped loader files, RAMdisk image, custom RAMdisk kernel, and RAMdisk configuration files.

```
unload
load /boot/maint/k.CUSTOM
load -t md_image /boot/maint/fs_img
include /boot/maint/params
set vfs.root.mountfrom=ufs:/dev/md0
autoboot 10
```

Fig. 4. An example `/boot/maint/loader.rc` file.

V. Future Directions

LOOKING forward, we plan to investigate if a Secure Shell Maintenance RAMdisk Environment could be loaded via a kernel module so that the dependence on the `/etc/` directory could be removed. Today, the kernel already attempts to mount from `/dev/md0` in a fallback situation if `options MD_ROOT` is defined. With this RAMdisk image, the system would then be networked according to the configuration in figure 2.



Adrian Steinmann earned a Ph.D. in Mathematical Physics from Swiss Federal Institute of Technology in Zürich and has over 15 years experience as a technical consultant and software developer. He has been working on FreeBSD since 1993 (version 1.0) and since 1997 he maintains and develops the base system for a remote managed firewall called 'STYX' [3]. He is fluent in Perl, C, English, German, Italian, and has passion and flair for finding simple solutions to intricate problems.

References

[1] PC Engine WRAP: *Wireless Router Application Platform; 2002–2005*; 266 MHz AMD Geode SC1100 CPU, 128MB SDRAM, CF, 2-3 LAN, 1-2 Mini-PCI; `www.pcengines.ch`

[2] Soekris net4501: *Compact, low power, low-cost, communication computer; 2001–2005*; 133 MHz, 64MB SDRAM, CF, 3 LAN, 1 Mini-PCI; `www.soekris.com`

[3] STYX Firewall: *FreeBSD-based Remote Managed Firewall; 1997–2005*; `www.styx.ch`