OpenBSD rc.d(8)

Antoine Jacoutot

ajacoutot@openbsd.org
The OpenBSD project

Abstract

In this paper we will present the OpenBSD rc.d(8) framework and rc.subr(8) daemon control routines written by Robert Nagy (robert@openbsd.org) and myself with the valuable input of Ingo Schwarze (schwarze@openbsd.org).

While it resembles other implementations, it was written from scratch to match the project objectives (simple, sequential, non-intrusive). We will describe the internals of rc.d and rc.subr as well as the rcctl(8) tool. We will also talk about the implications that these had on the traditional BSD start-up sequence.

1. Introduction

OpenBSD has always used the traditional static BSD initialization script: */etc/rc*.
While dependable, it did not allow for easy integration with monitoring, configuration management software and/or any kind of tools requiring automated service handling.
rc.d(8) was developed to abstract service management while pertaining the existing behavior like predictive and sequential start-up ordering (dependency-less).

The following sections will briefly describe our requirements as well as the existing implementations and alternatives and explain why it was decided to write one from scratch. We will see how we managed to plug ourselves into the existent without having to transform it. We will learn how to use the rc.d control scripts then talk about the internals of rc.subr and how start-up scripts look like. We will then introduce rcctl: an all-in-one utility for managing rc(8) daemons and services and look at how it helped orchestration and configuration management tools to work on OpenBSD.

2. Description of rc on OpenBSD

The way OpenBSD boots hasn't changed much since its inception. The boot loader will launch the kernel which will execute init(1) as the last stage of the boot process; init will then run the sequence of events described in */etc/rc*.

/etc/rc starts by doing some housekeeping, checking the disks, mounting partitions, setting the NIS domain name, setting up ttys flags... Note that whether we are booting into single-user mode or multi-user mode, the behavior is different (e.g. network and daemons aren't started in single-user mode). Only the multi-user mode is relevant to rc.d and this is what we will describe now.

Before rc starts most system daemons, the */etc/netstart* script is executed. Its role is to setup anything network-related (IP, carp(4), multicast, …). It is important to note that this is not controlled by the rc.d framework which is only meant for handling daemons.

At this stage we need to understand and differentiate the concept of daemon and service. A daemon is just your usual evil program configured to run in the background (e.g. sshd, ntpd) while a service is a facility (e.g. pf, ipsec, check_quotas); rc.d only handles daemons.

To decide whether a daemon should be started or a service should be enabled, rc will read its configuration from two files:

- */etc/rc.conf* which contains the default configuration
- */etc/rc.conf.local* which contains rc.conf(8) overrides

Starting a daemon and setting its flags is configured using a single configuration variable: *daemon_flags*. A value of *NO* means the daemon is not enabled and will not start. Any other value will start the daemon using the flags contained in *daemon_flags* (which can be empty in which case the daemon will start without any flags).

Starting a service is configured using the *service* name variable. A value of *YES* will enable the service and *NO* will disable it.

Values listed in rc.conf.local(8) will always take precedence: if rc.conf contains *sshd_flags=* and rc.conf.local contains *sshd_flags=NO*, then sshd will not start.

There are a few other shell scripts that are not part of rc.d per se but that */etc/rc* will run. These files do not exist by default and are always executed using sh(1).

- */etc/rc.securelevel*, actions that must be run before the securelevel(7) changes
- */etc/rc.firsttime*, actions that are only run once (rc will remove rc.firsttime afterward)
- */etc/rc.local*, misc actions (used to start non-base daemons before rc.d was introduced)
- */etc/rc.shutdown*, actions needed to run before the system shuts down

This is how things worked before and after rc.d was introduced. An exhaustive list of actions done by */etc/rc* during boot can be seen in Appendix A.

3. Existing implementations and alternatives

Although several implementations and alternatives exist, it was decided not to base OpenBSD rc.d on any of them. There are various reasons for that amongst which were:

- the need to preserve the existing behavior (not a replacement: we must plug rc.d on top)
- the need to only handle daemons
- the requirement for a small, robust, comprehensive, maintainable and easily debuggable framework
- something that needs to be both simple and simple (we do not like knobs!)

We did not want to implement a complete replacement, such as using event driven or socket activated services. Parallelization was out of scope because it's often hard to debug and boot-up speed was not a concern.

Amongst the numerous existing alternatives and implementations, some were dismissed pretty much right away because they are targeted to a specific operating system (Linux, Solaris, Mac OS), have a bad license (GPL is not acceptable), would replace existing components like init(8), inetd,(8) cron(8)... or more generally would require a huge porting effort which would transform the current paradigm and probably introduce dangerous bugs.
e.g. *systemd*, *SMF* and *launchd*.

OpenRC (from Gentoo Linux), *runit* and *daemontools* are interesting alternatives. They do not have to replace init but still provide too much. There's a lot of support for specific things we don't need. They would also transform the entire start-up sequence which would require many modifications to keep the existing behavior working without much gain compared to a brand new implementation.

Slackware Linux provides an rc.d implementation which somewhat consists of a mix between SysV and traditional BSD init scripts. However there's no factorization (each script re-implements everything by itself) and the addition of an rc.d script from an external package requires a modification to rc.M or rc.local. In addition to that, changing the flags of a particular daemon is done by editing the script itself (which would require a special attention at upgrade time) and there's no easy way to change the start-up order. It's a nice and simple implementation but too static for our needs.

FreeBSD and NetBSD *rc.d* and *rcorder* could have been the obvious choice but again, they provided too much. The rc.d scripts are also too flexible (which can be a feature but not in our case) and the framework provides a dependency-based dynamic start-up ordering which was against our goals because we wanted to keep full control over the start-up sequence. It is our opinion that automatic determination of daemon start-up is pointless because if a machine is running a very large number of services in production, that's a bad system design in the first place and it's easier to order a small number by hand than to debug issues with automatic ordering.

There are a few other less known alternatives but eventually we agreed that such functionality was better implemented close to the underlying operating system and so it was decided to write something from scratch that is really small and targeted to our own requirements; nothing more, nothing less. We decided that some of the features provided by other alternatives did not belong in rc.d, like supervision which often leads to processes flapping due to automatic restarts. These specific needs are usually better satisfied using specialized software run on top of rc.d.

To summarize, other alternatives were too clever for us and would take care of much more than what we wanted: starting the network, mounting file-systems, setting up ttys... were out-of the scope for the framework we wanted to build.

4. Plug-in ourselves into the existent

When rc.d was first introduced in OpenBSD on October 2010, it was designed for ports only. Moving base was not part of the initial scope because we needed to prove the solution was viable and non intrusive before it could be considered for. It was however the ultimate goal.
Having seen users struggle with how to interact with services (kill, apachectl, rndc...) we were confident it would be a welcome change.

The first rule we enforced upon ourselves was to use the already available and standard facility for signaling daemons: kill(1). The use of PID files was discarded for the usual known reasons (racy, left-over files...). We knew it would not be enough for everything out there (some daemons require a helper to cleanly shutdown...) but it was good enough that we could make it a default. That's the reason why something like start-stop-daemon(8) to control creation and termination of the daemon processes was not considered. Besides since the whole rc framework was already just a bunch of shell scripts, extending it meant that we had to keep using shell, for better or worse.

Since we had to be able to plug this new framework into the existent without any disruption, we decided to start by not modifying */etc/rc* at all but instead use the */etc/rc.local* script where external packages daemons were traditionally started from.

And so we did:

```
for _r in $rc_scripts; do
```

```
    [ -x /etc/rc.d/${_r} ] && \
    /etc/rc.d/${_r} start && \
    /echo -n " ${_r}"
done
```

At that time, */etc/rc.d/rc.subr* which is sourced by the rc.d scripts and which provides all subroutines for rc.d was 54 lines long. It couldn't get simpler than that but of course needed to be extended to cope with all the weird and different daemons out there. This will be described in the upcoming sections.

A few code iterations and one release later, rc.d eventually found its way into handling base system daemons. Besides the obvious benefits of having a generic way to interact with processes, a nice side effect was environment sanitation. Under some circumstances some unwanted variables could end up being injected into the process environment leading to unexpected and/or dangerous behavior.
Restarting applications would have the user environment leaked to the started daemon because it was not started in isolation.
The “old” way was not so secure after all and could retrospectively be considered as “broken”. Thanks to the use of su(1) in rc.d, the calling user environment is discarded. That was the decisive factor for rc.d adoption in base.

As of today, rc.subr is about 219 lines of code which is very small considering that */etc/rc* lost about more than a 150 lines and the feature gain we won.

5. Features and usage

There are 4+1 actions available:

- *start* → /path/to/daemon --flags
- *stop* → pkill
- *reload* → pkill -HUP
- *check* → pgrep
- *restart* → *stop* && *start*

Actions must be run as a privileged user (sudo(8), doas(1), root) except for *check* (albeit not always true as we will see further on).

In addition to these, there are 2 optional flags that can be passed to a script:

- -d → debug mode; describe the action taken and display the daemon stdout/stderr output
- -f → force mode; allow starting a base daemon when it is not enabled (i.e. *daemon_flags=NO* – similar to *onestart* in FreeBSD and NetBSD); packages rc.d scripts usually never need this flag because they do not have default flags registered in rc.conf.

Keeping the number of actions down really matters because that's the main user interface.

start

This action will start the daemon with its corresponding flags, timeout, user and login class.

```
su -l -c daemon -s /bin/sh root -c \
    "/path/to/daemon –flags"
```

stop

This action will stop the daemon by sending a SIGTERM to the matching process name.

```
pkill -xf "process name"
```

reload

This action will ask the daemon to re-read its configuration file by sending a SIGHUP to the matching process name.

```
pkill -HUP -xf "process name"
```

check

This action will check whether the daemon is running by grep(1)ping the process list for the exact matching process pattern.

```
pgrep -q -xf "process pattern"
```

restart
This action will run the *stop* action then if successful it will run *start*.

```
/etc/rc.d/daemon stop && /etc/rc.d/daemon start
```

By default, the “process pattern” known as the *pexp* in rc.d is constructed using the daemon path (set by the rc.d script) and its flags (set by the rc.d script or rc.conf{.local}).
When the rc.d script runs the *start* action, it stores the value of pexp under */var/run/rc.d/daemon*. We use this information when a daemon rc.d configuration is changed to make sure we match the currently running daemon process line.

All these actions are fully configurable and overridable as will be explained in the next section.

Comparing to a SysV init script, this is how a typical rc.d script looks like; most do not need to contain anything more than:

```
#!/bin/sh

# path to the executable we will run
daemon="/path/to/daemon"
# source rc.subr to get functions and variables
. /etc/rc.d/rc.subr
# run rc_cmd() with the given action
rc_cmd $1
```

As seen before, enabling a daemon is done by adding *daemon_flags=* to */etc/rc.conf.local*. However, this only works for base system daemons because */etc/rc* has no knowledge of external packages rc.d scripts (it only sequentially runs the enabled base daemon scripts). To cope with this, a new variable was introduced: *pkg_scripts*. Any daemon listed in this variable will be started in the provided order.

Four variables can modify the behavior of an rc.d script:

- *daemon_class* (default: *daemon*): BSD login class the daemon will run under (resource limits...)
- *daemon_flags* (default: *<empty>*): flags passed to the daemon
- *daemon_timeout* (default: *30*): maximum time in seconds to stop/reload a daemon
- *daemon_user* (default: *root*): user the daemon will run as (i.e. target user for su)

These defaults can be overridden by the rc.d script itself, */etc/rc.conf* (operating system defaults) or by editing */etc/rc.conf.local*.
When rc.conf or rc.conf.local is used to modify these variables, the rc.d script name is substituted to "daemon" in the variable name.
For example, the defaults flags for the netsnmpd daemon are set by its rc.d script as follow:
daemon_flags="-u _netsnmp -I -ipv6"
If we wanted to drop IPv6 support and log addresses, we would add the following to rc.conf.local:
netsnmpd_flags=-u _ netsnmp -a

daemon_class is particular in that regard and is not configured using rc.conf but is instead automatically set to a login class of the same name as the rc.d script if it exists in login.conf(5). It is a simple way to change the nice(1)ness, the environment, the limits... of a daemon without the need to re-implement anything.
For example, if we wanted to modify the netsnmpd daemon login class to change the open file descriptors per process, we would add the following to */etc/login.conf:*

```
netsnmpd:\
        :openfiles-cur=512:\
        :tc=daemon:
```

Here's a typical */etc/rc.conf.local* example:

```
apmd_flags=-A
hotplugd_flags=
saned_flags=-s128
pkg_scripts=messagebus saned cupsd
```

Special cases

There are a couple of special cases that are handled as follow:

- meta rc.d script: regular shell script under /etc/rc.d whose only job is to run several regular rc.d scripts with the given action in a specific order (e.g. */etc/rc.d/samba* will run */etc/rc.d/smdb* and */etc/rc.d/nmbd*)
- multiple instances of the same daemon: in most cases, this can be done by linking to the original rc.d script using another name (e.g. *ln -f /etc/rc.d/snmpd /etc/rc.d/snmpd6*) and using the new name in rc.conf.local to set up specific flags; this requires the pexp variable to be set in such a way that only the corresponding daemon instance is matched

6. Internals

The whole framework is defined under */etc/rc.d/rc.subr*. It's the entry point and is sourced by rc.d scripts to get the necessary functions and default variables.

All standard functions that come by default can be overridden by the rc.d script that is being run. They must only return true(1) or false(1).

Some daemons do not support a particular function in which case the rc.d script will override and disable it by having the function name variable set to *NO*. For example if a daemon cannot reload itself, the rc.d script will contain:
rc_reload=NO

rc_start()

This function is responsible for starting a daemon, it defaults to:
${rcexec} "${daemon} ${daemon_flags} ${_bg}"

rcexec is set by rc.subr as follow:
rcexec="su -l -c ${daemon_class} -s /bin/sh $ {daemon_user} -c"

If the *rc_bg* variable is set to *YES* by the rc.d script, rc.subr will set *_ bg* to "&" to start the daemon into the background. It is used in case a program can no daemonize on its own (daemontools style).

As an example, this is how the SSH secure shell daemon sshd(8) would end up being started by the rc.d framework:
su -l -c daemon -s /bin/sh root -c /usr/sbin/sshd

rc_stop()

This function is responsible for stopping a daemon, it defaults to:
pkill -xf "${pexp}"

pexp can be overridden by the rc.d script and defaults to:
pexp="${daemon}${daemon_flags:+ $ {daemon_flags}}"
It must match the process we want to kill and can be a POSIX regular expression.

rc_reload()

This function is responsible for reloading a daemon, it defaults to:
pkill -HUP -xf "${pexp}"

By reloading, we mean that the daemon is able to re-read its configuration without the need to be fully restarted.

rc_check()

This function is responsible for checking whether a daemon is running or not, it defaults to:
pgrep -q -xf "${pexp}"

A return code of 0 (*true*) means the daemon is running.

rc_cmd()

This is the main function and is the last command

called by an rc.d script. It takes one of the 5 arguments described below.

- **start**: if the daemon is enabled, check that it is not already running then run rc_pre() (see below) and if successful, run rc_start() then wait up to $daemon_timeout seconds for the process to start
- **stop**: check that the daemon is running then run rc_stop(), wait up to ${daemon_timeout} seconds for the process to end then run rc_post() (see below)
- **restart**: run */etc/rc.d/daemon stop* then if successful */etc/rc.d/daemon start*
- **reload**: check that the daemon is running then run rc_reload()
- **check**: run rc_check()

Misc functions and variables

As seen above, the *start* action will invoke rc_pre() before starting a daemon. This is used when a software has pre-launch time requirements. A typical use case for this is creating a directory in which the user the daemon is running as can write a socket file, a PID file...

The rc_post() action is invoked after a daemon process has been killed by rc_stop(). This is typically used to remove dangling lock files or putting the system back into a pristine state in case a daemon makes intrusive modifications.

Some daemons have a specific rc_check() function defined in their script which may require running some helper as a specific user. In this situation, a regular unprivileged user may not be able to *check* whether a daemon is running. To cope with this situation, the *rc_usercheck* variable is set to *NO* to warn the user that root access is needed to run this action.

The only mandatory variable of an rc.d script is *daemon*. It defines the path to the executable that the script will run. In case a program needs a switch to daemonize, it is considered best practice to append it to this variable to prevent users from removing it when they want to override daemon_flags.

e.g. smbd(8) from SAMBA (provides SMB/CIFS services to clients)

```
$ grep ^daemon /etc/rc.d/smbd
daemon="/usr/local/sbin/smbd -D"
```

Modifications to /etc/rc

The amount of modifications needed to plug the rc.d framework into rc ended up being relatively small.

For base system daemons, */etc/rc* will run the start_daemon() function using the daemon rc.d script name. If *daemon_flags* is not set to anything but *NO*, then the daemon will be started.

```
start_daemon() {
    local _daemon

    for _daemon; do
        eval "_do=\${${_daemon}_flags}"
        [[ $_do != NO ]] && \
            /etc/rc.d/${_daemon} start
    done
}
```

e.g.
start_daemon mountd nfsd lockd statd amd

To be able to get the actual *daemon_flags*, we need to source */etc/rc.d/rc.subr* but are interested only by the internal functions it provides (i.e. *FUNCS_ONLY*) to be able to parse the rc configuration. The other rc.subr functionality is only relevant to the rc.d scripts themselves.

```
FUNCS_ONLY=1 . /etc/rc.d/rc.subr
_rc_parse_conf
```

At shutdown time, we only stop the daemons installed from external packages (ports) in the reverse order that they are listed in *pkg_scripts*. The reason we are not shutting down base daemons is that they all properly shutdown when receiving a SIGTERM and their ordering does not matter.

```
pkg_scripts=${pkg_scripts%%*( )}
    if [[ -n $pkg_scripts ]]; then
        echo -n 'stopping package daemons:'
        while [[ -n $pkg_scripts ]]; do
            _d=${pkg_scripts##* }
            pkg_scripts=${pkg_scripts%%*( )$_d}
            [[ -x /etc/rc.d/$_d ]] && \
                /etc/rc.d/$_d stop
        done
        echo '.'
    fi
```

Starting daemons installed from packages is done by iterating the pkg_scripts variable from */etc/rc.conf.local* and running the corresponding rc.d script in the given order. It is done near the end of */etc/rc* right before */etc/rc.local* is executed and late base-system daemons are started (apmd, cron...).

```
# Run rc.d(8) scripts from packages.
if [[ -n $pkg_scripts ]]; then
    echo -n 'starting package daemons:'
    for _daemon in $pkg_scripts; do
        if [[ -x /etc/rc.d/$_daemon ]]; then
            start_daemon $_daemon
        else
            echo -n " ${_daemon}(absent)"
        fi
    done
    echo '.'
fi
```

A few dozen lines got removed when moving away from old constructs like:

```
if [ X"${ntpd_flags}" != X"NO" ]; then
        echo -n ' ntpd'; ntpd $ntpd_flags
fi
```

to:

```
start_daemon ntpd
```

That was the extent of the modifications required to plug-in our new rc.d system.

Note that since the sequence is static, it can be a challenge to inject a command early-on into the boot process. A workaround for this is to use the hostname.if(5) "!command" feature which is executed when the corresponding interface is configured (useful for adding routes for example).

7. rcctl

While having a generic framework opened the door to new possibilities like having monitoring tools or configuration managements systems easily interact with daemons, one thing was still missing to make automation software happy: being able to enable and disable daemons as well as passing options. So we needed an rc.conf.local “editor”.

That's the reason rcctl(8) was born.

```
$ rcctl
usage: rcctl get|getdef|set service | daemon
            [variable [arguments]]
       rcctl [-df] action daemon ...
       rcctl disable|enable|order [daemon ...]
       rcctl ls lsarg
```

rcctl(8) can configure and control daemons and services (add/remove/modify */etc/rc.conf.local* variables), interact with them (by running their rc.d script) or display information about them. Feature wise it is kind of a merge between the service(8) and chkconfig(8) utilities and a sysconfig editor as found in Red Hat.

rcctl was developed to abstract all specific cases (daemon versus service, regular versus meta script... e.g. *multicast=YES* versus *sshd_flags=*) and to present a unified interface to the user.

One particular issue that came to our attention while developing rcctl was the need to source rc.conf, rc.conf.local and the rc.d script to get the current and/or default values. Considering the fact that an rc.d script also sources */etc/rc.d/rc.subr*, the picture suddenly looked very fragile. Multiple sourcing of shell scripts coupled with a shell utility that would modify a system critical configuration made us re-think the way we should do things. That was especially true since at that time users would abuse rc.conf.local by inserting shell code. Since rc.conf and rc.conf.local are in fact configuration files, we decided they should be treated as such and be parsed instead of sourced. That would also prevent a typo or some shell code in one of these files to corrupt the start-up sequence.
An _rc_parse_conf() function was implemented with a white-list of authorized variables so that only these would be evaluated and anything else would end up being ignored.

Below are few sample usage case samples of rcctl.

Enable the NTP daemon:

```
# rcctl enable ntpd
```

Start the NTP daemon:

```
# rcctl start ntpd
ntpd(ok)
```

Enable two instances of the snmpd(8) daemon with specific flags and start them:

```
# rcctl set snmpd status on
# ln -s /etc/rc.d/snmpd /etc/rc.d/snmpd6
# rcctl set snmpd6 status on
# rcctl set snmpd6 flags -D addr=2001:db8::1234
# rcctl start snmpd snmpd6
```

Change the shutdown timeout for SQUID:

```
# rcctl set squid timeout 60
```

Get and change the ordering of the package rc.d scripts sequence:

```
# rcctl order
cupsd cups_browsed messagebus avahi_daemon gdm
# rcctl order messagebus avahi_daemon cupsd
cups_browsed gdm
```

Check if we have daemons that are supposed to run (enabled) but aren't:

```
# rcctl enable sshd
# rcctl stop sshd
# rcctl ls faulty
sshd
```

Get APM daemon flags:

```
# rcctl get apmd flags
-A
```

Mixing the use rcctl(8) with manual edition of */etc/rc.conf.local* is perfectly supported, it is not one or the other. Eventually, support was added in Puppet, SaltStack and Ansible, allowing to replace some wonky code and make OpenBSD become a first class citizen within automated infrastructures.

Conclusion

The aim of the rc.d system on OpenBSD is to provide a simple way to control daemons while keeping the original paradigm intact. It is a compromise between features, ease-of-use and simplicity. The addition of rcctl also gave us a unified interface to the rc framework. It is by no mean a complete replacement of the traditional BSD initialization sequence, a process control framework nor a service supervisor.

While somewhat analogous to a SysV init, we prevented the unmaintainable bloat which in our opinion was its main flaw (how many people really use the non-default runlevels in practice?) and not the fact that it was old and in shell.

Our rc.d does come with one obvious deficiency which is not being able to always match a specific daemon when several occurrences are running. This can happen when a program resets its process title to a generic

name (using setproctitle(3)).

At the time of this writing, there's work in progress to add rdomain(4) support so that a daemon belonging to a specific routing table can be properly handled without interfering with similar instances.

OpenBSD likes simple things and rc.d is as simple as it can get. It is robust, logical and easy to apprehend for anyone who can read shell scripts. It's built on components that have been working for decades and really fits into the UNIX philosophy of using simple tools that do one job and do it well.

By its nature it may not be suitable for all possible uses. Handling elaborate applications stacks is often better done on a case by case basis by using external helpers run from rc.local and rc.shutdown rather than bloating the rc.d framework.
That said, it shows how a shell script can be powerful enough to provide all the mechanics that such a framework needs and can be easily integrated to any operating system.

Code references

- http://cvsweb.openbsd.org/cgi-bin/cvsweb/src/etc/rc
- http://cvsweb.openbsd.org/cgi-bin/cvsweb/src/etc/rc.d/
- http://cvsweb.openbsd.org/cgi-bin/cvsweb/src/etc/rc.d/rc.subr
- http://cvsweb.openbsd.org/cgi-bin/cvsweb/src/usr.sbin/rcctl/rcctl.sh
- http://cvsweb.openbsd.org/cgi-bin/cvsweb/ports/infrastructure/templates/rc.template

Appendix A - Order of actions run by /etc/rc at start-up (handled by the rc.d framework)

- ✔ set the NIS domain name if configured
- ✔ add swap (block-devices only)
- ✔ check the file systems
- ✔ mount the root partition ('/' over NFS is needed if configured this way)
- ✔ set the TTY flags from /etc/ttys
- ✔ set the keyboard encoding
- ✔ configure the console (wscons(4))
- ✔ create an initial temporary PF ruleset (ssh, dhcp, ipv6 routersol, rpc, carp...)
- ✔ set sysctl(3) values
- ✔ start the network by running "sh /etc/netstart" (and associated !command)
- ✔ set the CARP interlock to prevent preemption until the system is fully booted
- ✔ load the PF rules from /etc/pf.conf
- ✔ mount /usr and /var if they are not mounted already (and a mount point exists)
- ✔ create and feed the random seed
- ✔ cleanup left-over spool files
- ✔ copy the dmesg(1) output to /var/run/dmesg.boot
- ✔ create IPsec and SSH keys if they do not exist
- ✔ start early daemons (syslogd, pflogd, ntpd...)
- ✔ load IPsec rules
- ✔ start RPC daemons (mountd, ypbind...)
- ✔ check and mount all remaining unmounted file systems
- ✔ add swap (non block-devices only)
- ✔ save an eventual core dump of the operating system under /var/crash/
- ✔ enforce file systems quotas if configured
- ✔ build kvm(3) and /dev databases (/var/run/dev.db and /var/db/kvm_bsd.db)
- ✔ set proper permissions for the TTY device files
- ✔ check and warn if a password lock file exists (/etc/ptmp)
- ✔ clean-up /tmp
- ✔ create UNIX sockets directories for X.org
- ✔ start pre-securelevel(7) actions by running "sh /etc/rc.securelevel" if it exists
- ✔ set default securelevel(7)
- ✔ patch /etc/motd
- ✔ enable accounting if enabled
- ✔ configure the shared library cache (/usr/local/lib, /usr/X11R6/lib)
- ✔ start network daemons (sshd, snmpd, relayd dhcpd, smtpd, spamd...)
- ✔ start one time actions by running "sh /etc/rc.firsttime; rm /etc/rc.firsttime" if it exists
- ✔ start daemons listed in pkg_scripts
- ✔ start other admin-specified actions by running "sh /etc/rc.local" if it exists
- ✔ disable the CARP interlock
- ✔ configure the audio mixer
- ✔ start local daemons (apmd, cron, hotplugd...)
- ✔ display the date and time