The fedora.us buildsystem

Enrico Scholz

Revision History
Revision 0.1 2003-11-28
Revision 0.2 2004-01-14
fixed/added some links

Table of Contents

Requirements on the build-system ... 1
Attacks from upstream author ... 1
Attacks from the packager ... 2
Effects of attacks ... 3
Summary ... 4
Components ... 5
mach ... 5
vserver ... 5
SELinux ... 7
User Mode Linux (UML) ... 7
QEMU/Bochs ... 7
vserver-djinni ... 7
The current buildsystem ... 11
The physical host ... 11
The buildmaster vserver ... 11
The buildslave environment ... 12
The buildroot(s) ... 12
Problems ... 12
Different rpm-database layouts and rpm-versions ... 12
disttag-replacement ... 12
Bibliography ... 13

Abstract

This paper gives an overview about ways to implement a buildsystem for rpm-packages and about the current fedora.us buildsystem. The official source of this document is http://www.tu-chemnitz.de/~ensc/fedora.us-build where it can be found in a HTML [http://www.tu-chemnitz.de/~ensc/fedora.us-build/html] and PDF [http://www.tu-chemnitz.de/~ensc/fedora.us-build/files/buildsystem.pdf] format.

Requirements on the build-system

There are two critical requirements on the build-system: it must build packages in a reliable manner, and it must be resistant against attacks. Builds without much overhead and buildmaster intervention are other requirements.

Attacks from upstream author

There are various ways for the upstream author to attack the buildsystem. Basically, it is to differ between attacks at package-buildtime, and these at the runtime of the package:

Runtime-attacks are the creation of backdoors in the programs (trojans), or to make them doing bad things at certain times (timebombs). On first glance, the runtime case seems to be uninteresting for the buildsystem itself, but as shown below this matters also.

Buildtime-attacks are possible since arbitrary code will be executed by the make-process. So:

- local root exploits (e.g. backdoors introduced by other packages, overflows in suid'ed programs or daemons, kernel flaws) can be used to gain privileges and to spy out secrets (e.g. ssh keys), or to modify files (e.g. `/etc/passwd` or `/lib/libc.so.6`). Since chroot(2) is easy breakable[1], files of the host system can be compromised although the build happens in a chroot. Creating special devices (e.g. `/dev/hda`) and operating on them would by-pass chroots also; having access to `/dev/kmem` would allow to inject malignant code into the kernel.
- processes which are running with the same uid can be ptrace'ed and killed. Such processes can be parallel builds of other packages, or -- in combination with local root-exploits -- system-processes like init or sshd.
- a process could be spawned in the background and is killing or ptrace'ing processes of subsequent builds, or is modifying files of such builds.
- network-resources of the build-machine could be used to open an hidden warez/porn server, or to run attacks against remote machines.

There are known cases where buildtime-attacks are triggered by the hostname of the build-machine, so that the build appears unsuspiciously on the machine of QA testers.

Altogether, since complex source-code will be compiled and complex code be executed by the make-system, it would require a full audit to preclude upstream attacks, so this kind can not be detected by the QA.

Attacks from the packager

Special crafted .src.rpm files

An attacker could create special crafted (.src).rpm files which are exploiting implementation or design flaws of the rpm-program. So, arbitrary code could be executed on the build-machine or on the machine of QA people, when the package is extracted or queried.

Doing the requested rpm-action after running such code would hide this kind attack effectively. QA will not protect against this kind of attack.

Malicious patches/modified upstream sources

An attacker could introduce code which would have the same effects like attacks from upstream authors. Modified sources should be detected by QA when there exist signed md5sums or signatures of upstream sources. Most patches are small and should be audited by QA also.

Malicious rpm-scriptlets or triggers

On installation of packages, these scriptlets are running as root and can do nearly everything. Malicious shell-code in such scriptlets can and should be detected by the QA, but since this shell-code executes binaries of the current package and/or other packages, it will be impossible to preclude attacks when programs of untrusted packages can be used in scriptlets.

SUID packaging

A packager could set the SUID bit of programs which were never designed to run with root privileges and untrusted input. Doing so, backdoors could be created.

Such an attack can and must be detected by QA and/or automatic QA tools like rpmlint.

[1] http://list.linux-vserver.org/archive/vserver/msg00729.html

Effects of attacks

There are two categories of attacks: those which are attacking the build-machine itself, and those which are modifying the content of packages. In the chapter about bad upstream sources, most of the attacks against the build-machine were mentioned already, and on a compromised build-machine it would be easy to do anything -- inclusive the modification of packages. Therefore, only the aspects of package-modifications will be discussed in this section.

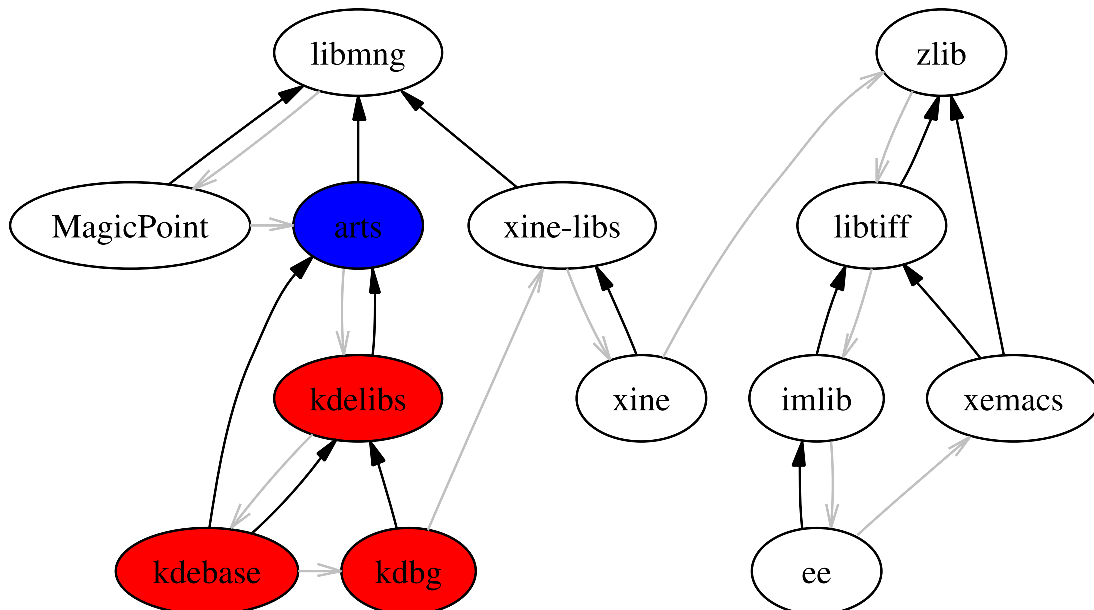
To do reliable builds, certain dependencies must be fulfilled which will lead to the installation of other, untrusted packages. As mentioned in the section called “Attacks from upstream author”, limiting the requirements to packages of trusted packagers will not work since good src.rpms can create bad binary rpms.

At the installation of a package, rpm-scriptlets will be executed with root privileges, so that chroot’s can be broken or files like libraries or programs be replaced. Since the replacement of files or the usage of chroot are common and valid operations for rpm, it is impossible to forbid these operations. Therefore, when the first untrusted package is installed, the build-root must be assumed as compromised.

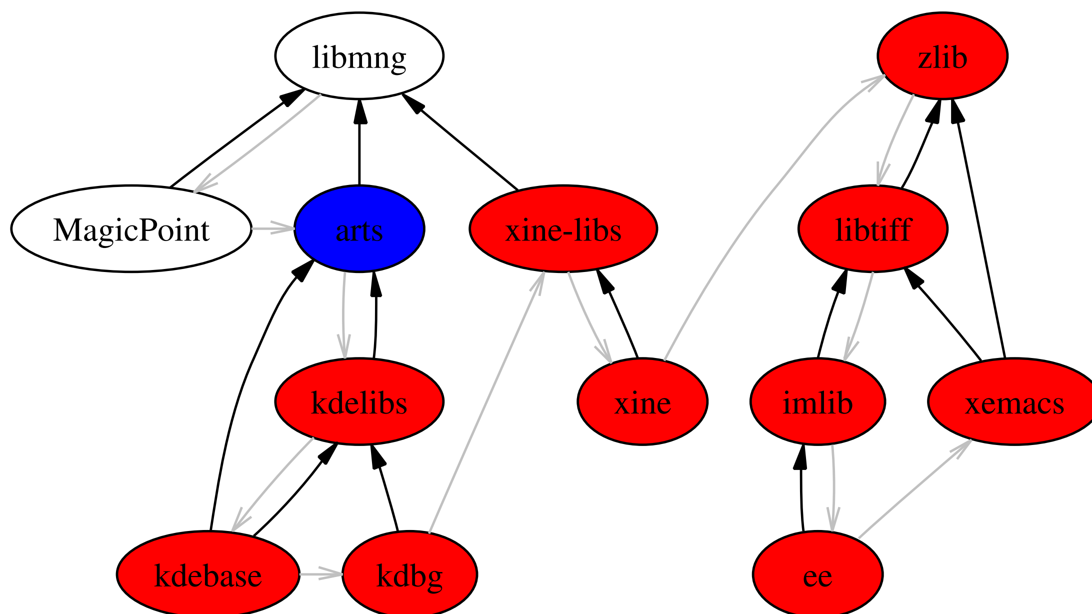
Now, when a build happens in such an hostile environment, injected code can be executed (e.g. by calling a replaced `make` program or a program using a prepared `/lib/libc.so.6`). It will be possible, that the build-results can be modified in such a manner that the created binary package carries the same malicious code as the original code. So, entire dependency-trees can be infected. Yet worse, when using the same build-environment for every package, the entire distribution will be infected.

As an example, Figure 1, “Infection of build-tree vs. infection of build-environment” shows the build of the MagicPoint, kdebase, kdb, xine, ee and xemacs packages inclusive their dependencies (in this order). Malicious code which replaces `/lib/libc.so.6` will be assumed in arts. The first image shows what happens when for each package an own build-system will be used, the second one shows the case when each build reuses the same build-system. The blue node marks the origin of the infection, the red ones the infected packages. The black arrows are symbolizing the BuildRequires:, the gray ones the build-order.

Figure 1. Infection of build-tree vs. infection of build-environment



Creating the build-system from scratch for each build



Reusing an already existing build-system

Summary

To prevent attacks and to have a reliable buildsystem, it must have the following properties:

Process-separation:
: it MUST be impossible to kill or ptrace processes of other buildroots or of the system. Hiding of foreign processes SHOULD be provided.

Device/kernel protection:
: Direct hardware access through `/dev/*` entries or modification of kernel parameters through `/proc` MUST be impossible. Forbidding the creation of such special files is one way to reach it, access restriction another one.

Unbreakable chroots:
: it MUST be impossible for a process in a buildroot to have any kind of access on objects of the systems (e.g. ssh-keys), or write-access on other buildroots.

No buildroot-reusing:
: each build MUST happen in an environment which can not be influenced by previous builds in this environment. This includes both filesystem-objects, and processes.

Resource-restrictions:
: excessive resource-usage (memory, diskspace,...) of a build SHOULD be prevented. Usage of certain resources (e.g. network) MUST be prohibited

Good performance:
: the buildsystem SHOULD should have only a small or non-existing impact on the performance.

Working environment:
: building of common packages MUST succeed. This requires certain `/dev` entries, and a mounted `/proc` filesystem at least.

Mature userinterface:
: the system SHOULD assist the buildmaster and automate the most tasks, so that the spent time will be reduced to a minimum.

Components

The build-system consists of several layers: there is the part which builds the rpms and assists the buildmaster. Currently, mach is the only alternative for this task. Then, there is the part which ensures the security. This one has several, mutually exclusive alternatives: vserver, SELinux, UML and QEMU. Because of the introduced restrictions, most of these components will need special helper. For vserver, this is vserver-djinni. And last but not least, there is needed something which glues the components together and provides a frontend to them.

The current fedora.us buildserver has a working mach & vserver & vserver-djinni setup.

mach

mach is a project founded by Thomas Vander Stichele, stands for “Make A CHroot” and can be found at Sourceforge [http://sourceforge.net/projects/mach]. It prepares a minimal chroot-environment for package-builds, installs needed buildrequirements and executes the build itself. There are other features also like easy configuration of the build-tree (e.g. RHL9, FC1, fedora.us stable,...) or collecting of build-results (binary/source rpms, buildlogs), so that mach became an essential part of the fedora.us buildsystem.

Problems:

- since usual chroots are not secure, additional layers are needed to provide the security aspects.
- mach does some operations like mounting the `/proc` filesystem into the chroot, which might collide with the security setup.
- the current mach version (0.4.2) fails to resolve conditionalized BuildRequires: and such ones which are pathnames (e.g. BuildRequires: /usr/include/db.h)

vserver

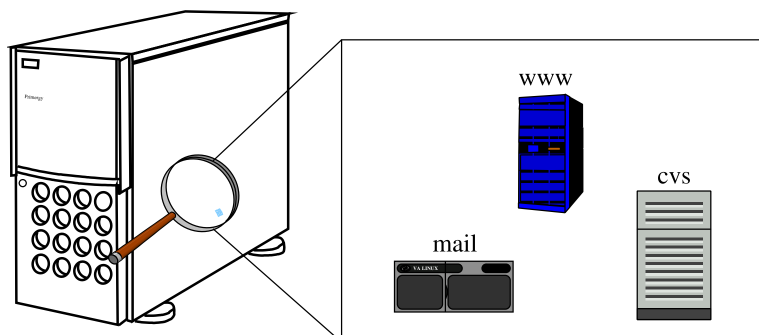
Vserver-Technology [http://www.linux-vserver.org] provides perfect process-separation, unbreakable chroots, is very lightweighted and has nearly no performance impact. Large parts of its security is based on linux capabilities, and the usage of network-resources can be restricted by setting the allowed interfaces/IPs and filtering it with external tools like iptables or iproute.

Vservers are identified by contexts; a process can see processes of the same context only. The setup is very easy and simple: each vserver appears as a standalone machine and is configured as a such one. There are existing extensions which are allowing context-specific disk-quotas. Figure 2, “A typical vserver setup” illustrates a typical vserver setup.

For the fedora.us buildsystem, the package-building happens in a separate vserver.

Problems:

Figure 2. A typical vserver setup




```
[root@cvs]# ps ax
  PID TTY      STAT   TIME COMMAND
    1 ?        S      0:04 init
26930 ?        S      0:00 /sbin/svlogd -t ./main/main ./main/auth ./main/debug
26935 ?        S      0:00 /sbin/socklog unix /dev/log
26979 ?        S      0:00 tcpserver-sshd ... 22 /usr/sbin/sshd -i
27024 ?        S      0:05 crond
27050 ?        S      0:00 tcpserver-cvs ... 2401 ... /usr/local/bin/cvswrap
  682 pts/2    S      0:00 /bin/bash -login
  728 pts/2    R      0:00 ps ax
[root@cvs]#

[root@www]# ps ax
    1 ?        S      0:04 init [3]   --init
12865 ?        S      0:00 /sbin/svlogd -t ./main/main ./main/auth ./main/debug
12866 ?        S      0:00 /sbin/socklog unix /dev/log
12869 ?        S      0:00 tcpserver-sshd -c 80 -q -l 0 ... 22 /usr/sbin/sshd -i
12876 ?        S      0:00 /usr/sbin/crond
12877 ?        S      0:00 /usr/sbin/httpd
13647 ?        S      0:03 /usr/sbin/httpd
13651 ?        S      0:03 /usr/sbin/httpd
  705 pts/0    S      0:00 /bin/bash -login
  708 pts/0    R      0:00 ps ax
[root@www]#

[root@mail]# ps ax
    1 ?        S      0:04 init [3]   --init
19811 ?        S      0:00 /sbin/svlogd -t ./main/main ./main/auth ./main/debug
19812 ?        S      0:00 /sbin/socklog unix /dev/log
19815 ?        S      0:00 tcpserver-sshd -c 80 -q -l 0 ... 22 /usr/sbin/sshd -i
19818 ?        S      0:00 /usr/sbin/crond
27020 ?        S      0:00 milter -O /var/lib/milter/default.py
27021 ?        S      0:00 milter -O /var/lib/milter/default.py
27022 ?        S      0:00 milter -O /var/lib/milter/default.py
28254 ?        S      0:00 sendmail: accepting connections
28262 ?        S      0:00 sendmail: Queue runner@01:00:00 for /var/spool/...
  765 pts/0    S      0:00 /bin/bash -login
  783 pts/0    R      0:00 ps ax
[root@mail]#
```

- vserver is an unofficial kernel-patch only and will not be in the kernel till 2.7/3.0. It conflicts with selinux which is the preferred technology of Red Hat. There does not exist a patch for current RHL9/FC1/RHEL3 kernels yet; vanilla 2.4.x (without NPTL and exec-shield) is required. Therefore, vserver based buildsystem can not be selfhosted by Fedora software.
- these changed environment can cause problems: db4 of RHL9/FC1 does not work on non-NPTL kernels, some packages (e.g. MIT-scheme) require exec-stack and may work well on the build-machine, but fail on real FC1 installations. The marking of pre-FC1 binaries and disabling exec-shield on them will not work, since the build-environment is FC1.
- because the build happens in a vserver with reduced capabilities, mach can not mount `/proc` into the buildroot natively, but requires some kind of helper. Similarly for creating `/dev` entries.
- the safe `chroot(2)` is a big, but working hack: contextes can not enter directories with `000` permissions.

```
if ((mode & 0777) == 0
      && S_ISDIR(mode)
      && current->s_context != 0) return -EACCES;
```

SELinux

How SELinux will/can fulfill the requirements was not explored yet. Restricting of network-resources is supported by SELinux

Open questions:

1. SELinux can protect foreign processes. But is it possible to hide them in `/proc` also?
2. Is `chroot(2)` implemented in a safe manner? Or, can parent directories of build-roots be protected with SELinux policies? Is a safe `chroot(2)` required at all?
3. What is the performance impact of the policy checking?
4. How can disk/memory usage restricted with SELinux? Would CKRM[2] be an option?
5. Can special mount-operations (e.g. `/proc` filesystem) be allowed by the policy, or does this require userspace helper also?
6. Setup of an SELinux policy seems to be very complicated. How possible are holes in a setup?

User Mode Linux (UML)

UML is similarly to vserver but emulates an entire Linux system. It is more heavyweighted, has a more difficult setup and has a performance impact, but offers interesting features like a copy-on-write filesystem which is missing on vserver. It does not require a special host-kernel, so that its chances to come into RHEL/Fedora are much higher than vserver's one.

Since UML provides nearly a full featured Linux environment, `/proc` mounting or device creation would not need userspace helpers.

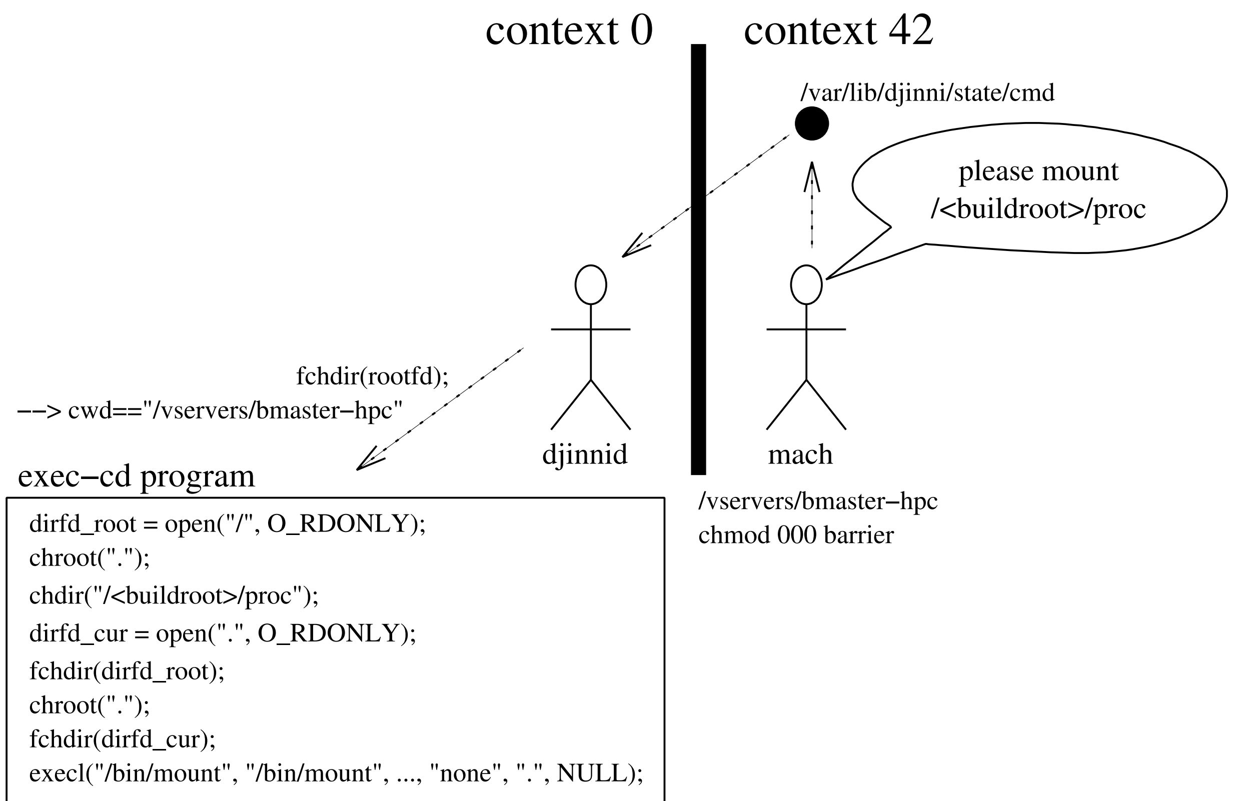
QEMU/Bochs

QEMU [http://fabrice.bellard.free.fr/qemu/] and Bochs [http://bochs.sourceforge.net/] are emulating a complete machine. They are very heavyweighted -- qemu slows down compilation in a factor of 2-4[3] -- are available for common architectures (IA32) only. Therefore, they are not really an option.

vserver-djinni

vserver-djinni is used to do privileged tasks like directory mounting in unprivileged vservers. To do this, a djinnid daemon is running in the privileged host-ctx and listens on commands from the vservers. One of djinni's designgoals was to enable a vserver-in-vserver functionality which is not doable with current vserver patch.

Figure 3. Djinni operation



Beside the command-socket which is used to transmit commands from client to djinni-daemon, there is a second watchdog socket. Before doing any other operation, a “djinni-rub” daemon must connect to this watchdog socket and must keep it open till the last operation. One this sockets gets closed (e.g. by killing the “djinni-rub” daemon) there is no way to re-enable the command socket from within the vserver.

Each djinni process is assigned to exactly one vserver and there are two kinds of commands for djinnid: these ones which are executing a single command and those, which are creating a new djinnid process for a vserver. When executing a single command, this command starts with having the vserver directory as its current working directory. With careful choosing of the following commands, symlink attacks from within the vserver can be prevented effectively.

The other kind of command creates the new sockets within a vserver environment and starts a new djinni process listening on them (command- and watchdog-socket). Upon startup of this new daemon, the top-directory of the new vserver will be entered in a secure way and its filedescriptor internally stored and used on subsequent operations.

One djinnid serves exactly one master which is identified through its uid/gid attributes and its context. The context is determined by the pid of the master-process which is transmitted through SCM_CREDENTIALS messages on the command-socket. Once such a message was transmitted to a newly created djinnid, all subsequent messages must have this origin (uid + ctx). To prevent certain kinds of attacks[4] an additional confirmation step is needed in the communication.

vserver-djinni is configured through a an hierarchical filesystem structure in `/etc/djinni.d`. Each file there which does not begin with a period means a command which can be sent to djinnid. Files beginning with a dot are marking special attributes of the vservers; such attributes are:

`.run`
: The command which will be executed; this file must be executable.

`.params`
: A syntaxdescription of allowed parameters; e.g. “p” for a path, “v” for a vservername, “[...]” for a set of possible values and so on.

`.new`
: When this file exists, the command will be used to create a new vserver. The content of this directory configures the command-set of the new djinnid.

`.trusted`
: When this file exists for current **and** all parent-configurations, the vserver will be assumed as trusted. Starting an untrusted vserver within an untrusted vserver is not supported.

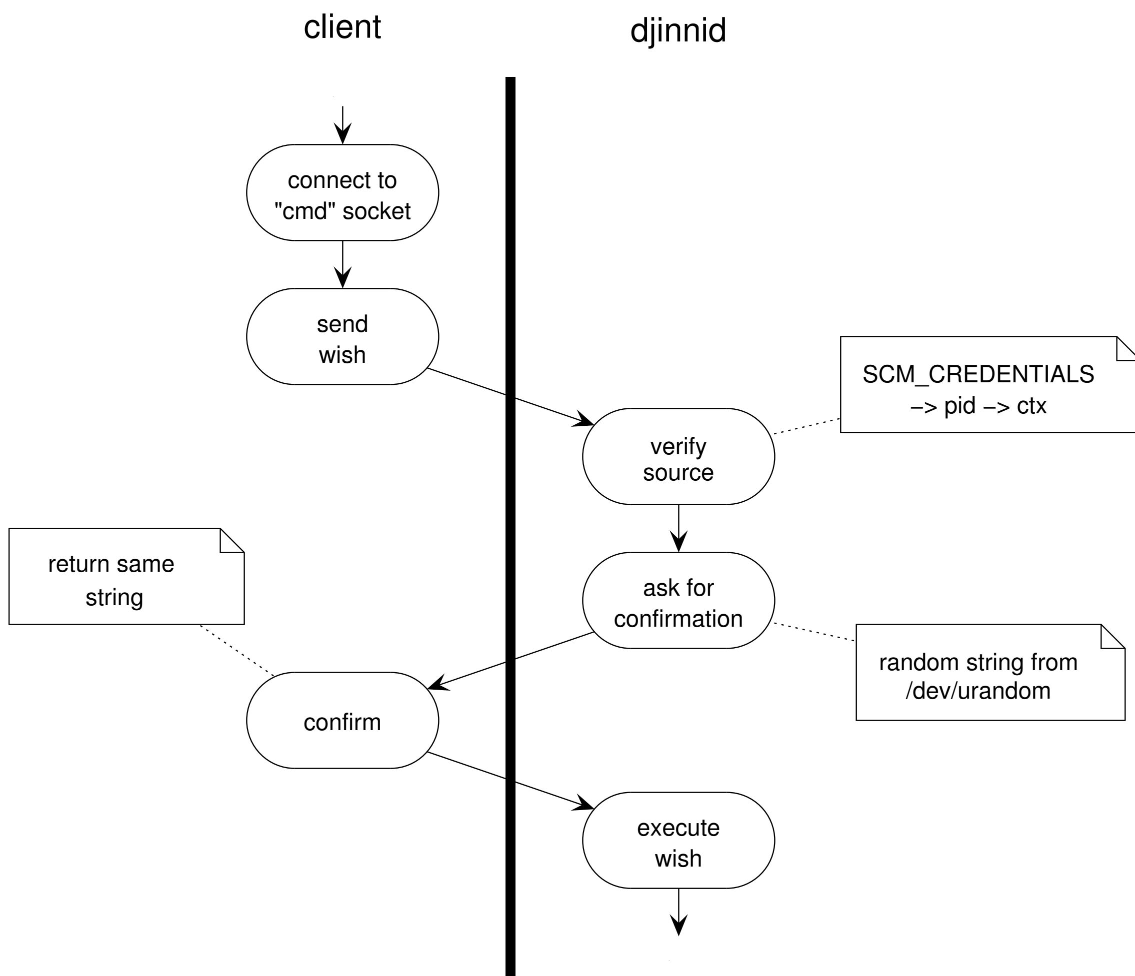
Example 1. Sample djinni.d configuration

[2] http://ckrm.sourceforge.net/

[3] http://fabrice.bellard.free.fr/qemu/benchmarks.html

[4] Attacker from other ctx gives command and terminates the process immediately. Now he enforces process creation in the authorized context (e.g. ssh-login as ordinary user) and speculates on a race between “send wish” and “verify source”.

Figure 4. Djinni filedescriptors and communication



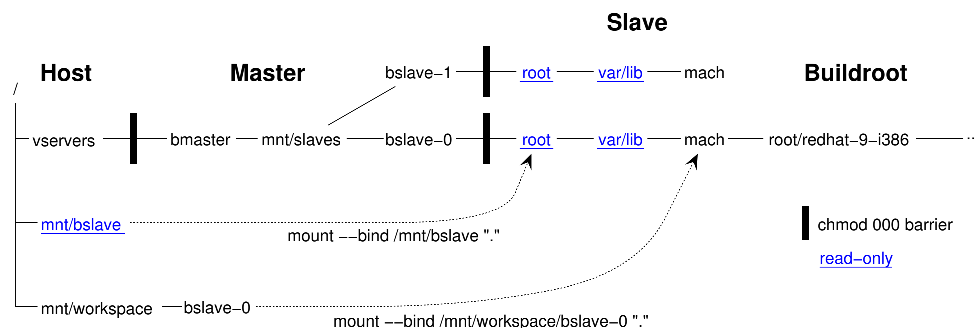
```
/etc/djinni.d
|-- .trusted
`-- new_bmaster
    |-- .new
    |-- .params
    |-- .trusted
    |-- new_bslave
    |   |-- .new
    |   |-- .params
    |   |-- .run
    |   |-- prepare_machroot
    |   |   |-- .params
    |   |   `-- .run
    |   `-- shutdown_machroot
    |       |-- .params
    |       `-- .run
    `-- stop_bslave
        |-- .params
        `-- .run
```

The current buildsystem

The current (and working) fedora.us buildsystem consists of the following parts:

- a vserver capable kernel,
- mach,
- vserver-djinni, and
- fedora.us-build which glues this parts together

Figure 5. Buildsystem structure



The physical host

The physical host is a box running a vserver-capable kernel. System is a usual Red Hat Linux distribution plus some extra-packages (vserver related). It does not need much disk-space; 500 MB should be enough for minimal installations. On this host the 'djinnid' daemon must be running with the commandset defined by the fedora.us-build package [http://www.tu-chemnitz.de/~ensc/fedora.us-build/files].

The buildmaster vserver

This is a vserver, where the buildmaster person(s) are starting the builds and are sending the built packages to a separate signing server. Requirements on this vserver are minimal also.

The build will be initiated by **start-build <dist> <package-url> <md5sum> <gpg-key#>** which is starting a buildslave in a separate context within an unbreakable chroot. This operations will require privileges to mount tmpfs filesystems or to create device inodes. The “new_bslave” command of djinni will be used for this task. In detail, this action

- mounts the readonly buildslave with “--bind”
- creates `/dev` in the slave as a tmpfs and populates it; dev-inodes on a readonly volume are not working, e.g. “echo >/dev/null” would fail there.
- creates `/var` in the slave as a tmpfs
- mounts a slave-specific, writable buildroot with “--bind” at `/var/lib/mach`

There exists a “stop_bslave” command with revokes this actions but does not do any removals/cleanups.

The buildslave environment

This environemt is created as a usual vserver and can be used in this way. But for build-server tasks it is just an ordinary chroot environment (no init step) which uses the unbreakable chroot feature of vserver kernels. This environment *MUST* be mounted readonly or write-protected in other ways.

The current buildsystem solves this by having a readonly mounted loopdevice at `/mnt/bslave` in the host. Now, each “new_bslave” djinni command will execute **mount --bind /mnt/bslave "."** in the root-directory of the buildslave.

The requirements of the buildslave are small; 200 MB should be enough. A buildslave does not build more than one package; at its startup all old data will be wiped and the new buildroot be initialized. The package-build itself is done with a slightly modified mach.

Since buildroot creation requires privileges to mount a `/proc` filesystem and to create special inodes, djinni will be called by mach with the “prepare_machroot” and “shutdown_machroot” commands.

The buildroot(s)

Each buildslave needs a separate buildroot where the real build happens. This buildroot is a directory in a workspace partition and is mounted with “--bind” at `/var/lib/mach`. The workspace partition contains temporary data only and should have a very fast filesystem; ext2 would be ideally for it. It needs much space; 2GB are required e.g. for a glibc build. Since each slave will need this, slave-count * 2GiB should be reserved for this area.

The buildroot-area contains mach files (cache, apt-configuration, ...) and will be wiped for each build. It can be initialized with prebuild filesystem images to speedup the build.

Problems

Different rpm-database layouts and rpm-versions

When using chroot-based builds the bootstrapping of the initial chroot-environment happens from the host. When using the host rpm-version, a later rpmbuild within the chroot may fail since the database format can not be understood by it. E.g. the rpm from Fedora Core 1 will create `/var/.../root/var/lib/rpm/Packages` in db4.1 format. Now, rpmbuild from RHL7.3 understands db3 format only and will fail therefore.

Another effect when bootstrapping with newer/other rpm versions might be a changed behavior of rpm. E.g. rpm-4.2.1 introduced strict epoch handling and will fail on lots of RHL9 or previous packages.

Status: solved by using different rpm binaries

disttag-replacement

fedora.us requires a disttag for clean update-paths. Since there does not exist any policy for packagers how this should happen, it is very difficultly to change the disttag at mass-builds. mach's methods are crude heuristics and are failing at some packages.

Status: most packages are working with the replace-old-suffix-with-new-one method, but a formal policy (forbid disttags entirely, or enforce usage of overridable %{?disttag}) would be highly appreciated.

Bibliography

[Vserver] chroot(safe) issues. http://list.linux-vserver.org/archive/vserver/msg05232.html.

vserver. Vserver project homepage and kernelpatches. http://www.linux-vserver.org.

util-vserver. Vserver userspace tools. http://www.nongnu.org/util-vserver.

vserver-djinni. http://www.tu-chemnitz.de/~ensc/fedora.us-build/files.

mach. http://mach.sf.net.

fedora.us-build. http://www.tu-chemnitz.de/~ensc/fedora.us-build/files.

CKRM. Class based linux Kernel Resource Management. http://ckrm.sourceforge.net.

QEMU. http://fabrice.bellard.free.fr/qemu.

Bochs. http://bochs.sourceforge.net.