Using Linux VServer

7. Juni 2004

Zusammenfassung

To use the full power of the kernel based linux-vserver concept, several user space utilities are needed. On the one side there are simple tools like low-level syscall wrappers, or enhancements of well-known UNIX utilities like ps or kill.

The other side is the management of the virtual servers (VS). Mostly, these VS are chroot environments and are based on ordinary Linux distributions which are not prepared for VS usage. Note, that the VS is independent from the host distribution: it is possible to have Debian and Gentoo VS on a Fedora Core host system (and vice versa). So the tools are providing methods to bootstrap a VS from scratch, mechanisms to start and stop VS (inclusive setting up networking and general machine parameters) and ways to share diskspace between VS.

Because isolation of untrusted applications and selling to unstrusted customers are typical VS applications, these management tasks must be done in a secure way.

The presentation will show the creation of Fedora Core and Debian based VS, their operation and which precautions were taken to prevent manipulations. The configuration scheme which is used by 'util-vserver' to manage the VS will be introduced too.

1 Einführung

1.1 Motivation

Linux bietet die Möglichkeit, eine Vielzahl an Diensten zu betreiben. Sobald diese in typischen Einsatzszenarien wie FTP-, LDAP-, Kerberos- und HTTP-Servern ernsthaft genutzt werden, kommt schnell der Wunsch auf, dafür dedizierte Server zu verwenden.

Die Gründe sind vielfältig: manchmal werden neue Programm- oder Libraryversionen benötigt, welche mit denen anderer Dienste in Konflikt stehen oder nicht abschätzbare Seiteneffekte haben. Außerdem existieren Techniken wie PHP, wo Anwendungen für unterschiedliche Einsatzzwecke im selben Adressraum und unter der selben Nutzerkennung laufen. Getrennte Installationen sind deshalb sowohl aus Sicherheits-, als auch Stabilititätsgründen vorteilhaft.

Es ist auch üblich, funktionsbezogene Hostnamen wie `www` oder `kerberos` zu verwenden. Da einige Dienste diesen Namen oder die eigene Internetaddresse in den Protokolldaten versenden, gerät man beim Betreiben mehrerer Dienste auf der selben Maschine schnell in Konflikte. Für Entwickler mag es auch interessant sein, ihr Projekt auf unterschiedlichen Linuxdistributionen testen zu können, ohne zeitraubende Multi-Boot Umgebungen bemühen zu müssen.

Die klassische Herangehensweise wäre, für jeden Dienst einen eigenen Rechner zu verwenden. In klein- oder mittelständischen Betrieben funktioniert dies aus Kostengründen aber meist nicht, da neben der reinen Anschaffung, auch Aspekte wie Wartung (Lüfter), Raumbedarf, Kühlung und Stromversorgung (USV) ins Gewicht fallen.

Optimal wäre es, wenn mehrere dedizierte Server auf der selben Hardwareeinheit betrieben werden. Im folgenden wird für diese Server der Begriff „virtuelle Server“ verwendet und „Host“ für die Hardwareeinheit an sich.

Neben dem Abgrenzen von Diensten existiert in Form von root-Servern für Kunden ein weiteres Einsatzfeld für dedizierte Server. Oftmals steht nur der Wunsch nach einer Internetanbindung mit Diensten wie einem

eigenen SMTP-Server, ohne dass heutige Hardware dadurch voll ausgelastet wird. Durch Verwendung virtueller Server könnten sich mehrere Kunden einen Rechner teilen und dadurch Betriebs- und Anschaffungskosten reduziert werden.

1.2 Anforderungen und Wünsche

Welche Anforderungen müssten an solche virtuellen Server gestellt werden? An erster Stelle sollten sie sich wie "normale"Rechner verhalten und die selbe Software einsetzbar sein. Es sollte kein Neukompilieren nötig sein, um spezielle Bibliotheken einzubinden, neue Syscalls aufzurufen oder andere Pfade zu vergeben.

Dann sollten die Prozesse zwischen virtuellen Servern abgegrenzt sein. Es muss unmöglich sein, absichtlich oder unabsichtlich DOS Attacken gegen andere Server durchzuführen oder deren Daten auszuspähen. Zum Beispiel sollte **/etc/init.d/sshd restart** nur den eigen SSH-Daemon neustarten, und **kill(2)** oder **ptrace(2)** gegen Prozesse anderer virtueller Server oder gar des Hostes wirkungslos bleiben.

Neben Prozessen muss auch das Filesystem abgegrenzt sein. Kollisionen bei Standardpfaden für Lockfiles oder Datenbereichen sollten ausgeschlossen sein. Geheimnisse wie SSH/GPG-Schlüssel oder andere Daten müssen gewahrt bleiben.

Natürlich dürfen keine Hintertüren zum Umgehen obengenannter Abgrenzungen existieren. Direkter Hardwarezugriff auf `/dev/hda` würde zum Beispiel Zugriff auf Geheimnisse im Filesystem bieten, und direkter Kernelzugriff durch `/dev/kmem` sämtliche Beschränkungen unwirksam machen.

Zusammengefasst muss es für einen virtuellen Server unmöglich sein, die Funktion anderer virtueller Server oder des Hostsystems zu beeinflussen oder Zugriff auf deren Daten zu erlangen (außer über definierte Schnittstellen).

Neben den genannten absoluten Anforderungen existieren natürlich auch Wünsche für diese virtuellen Server. So sollten sie effizient mit der Hardware umgehen, d.h. möglichst hohe Performance für CPU, I/O und Netz bieten und nur geringe Mengen an RAM- und Plattenspeicher verbrauchen.

Virtuelle Server sollten auch leicht managebar sein, d.h. ihre Erstellung und Betrieb sollten unkompliziert, weitgehend automatisch und mit bekannten Tools möglich sein. Ein weiterer Wunsch wäre die leichte Migration auf andere Hardwareeinheiten, um zum Beispiel auf geänderte Lastsituationen zu reagieren.

Bezüglich der root-Server für Kunden wären Garantien bezüglich der nutzbaren Ressourcen (CPU, RAM, Platte, Netz) und die Unterstützung oberer Grenzen ebenfalls wünschenswert.

1.3 Lösungsmöglichkeiten

VMWare ist wohl die bekannteste Möglichkeit, virtuelle Server zu realisieren. Dabei wird ein vollständiger Rechner emuliert, welcher prinzipiell wie ein „normaler“ Computer handhabbar ist. So sind alle gängigen Betriebssysteme darauf lauffähig und Installationen via Distributions-CD möglich. Allerdings ist dies nur durch sehr hohen Ressourcenverbrauch erreichbar und nur für herkömmliche Intel-Hardware implementiert. In die selbe Kategorie wie VMWare fallen „Bochs“ und „QEMU“ .

Im Gegensatz dazu, läuft bei User Mode Linux der virtuelle Server unter einem eigenen speziellen Linuxkernel. Es existieren dafür interessante Technologien wie Copy-On-Write (COW) Dateisysteme; der Performanceverlust ist aber immer noch hoch und Ressourcen können nicht geteilt werden.

Eine relativ neue virtuelle Server Technologie steht in Form von SELinux zur Verfügung. Prinzipiell sind damit die sicherheitstechnischen Anforderungen erfüllbar und der Performanceverlust ist relativ gering ($<$ 7%). Eine vollständige Virtualisierung, die zum Beispiel eigenständige Hostnamen oder IPs vorraussetzt, ist aber nicht erreichbar.

Dieser Artikel wird sich mit „Linux kernel-based virtual servers“ beschäftigen, welche mit BSD Jails, SUN Zones oder FreeVPS vergleichbar sind. Bei diesen Technologien laufen virtuelle Server auf gemeinsamer Hardware und unter dem selben Kernel; die Linuxdistributionen zum Betreiben der virtuellen Server und die des Hostes können aber unterschiedlich sein. Prozesse werden zu Kontexten gruppiert und es tritt nahezu kein Verwaltungsoverhead auf.

2 Grundlagen

Linux- VServer benötigen im Moment einen Kernelpatch welcher von der Projekthomepage <http://linux-vserver.org/> heruntergeladen und in gewohnter Art und Weise angewendet werden kann. Außerdem sind Userspace Tools nötig, wo zwischen util-vserver <http://www.nongnu.org/util-vserver/> und dem älteren vserver <http://www.solucorp.qc.ca/miscprj/s_context.hc> Paket gewählt werden kann.

Auf der Kernelseite wird jedem Prozess eine Kontext- und Netzwerk-ID zugeordnet. Bis auf spezielle Ausnahmen kann kein Prozess einen Prozess mit einer anderen Kontext-ID sehen und damit auch nicht beeinflussen. Kontexten können Attribute wie Hostnamen bzw. gesamte utsname Einträge [1] , Capabilities, Flags, Scheduling Parameter, Namespaces und IPs zugeordnet werden.

Die Sicherheit basiert größtenteils auf den normalen Linux Capabilities [2] , wo zum Beispiel ohne `CAP_MKNOD` keine neuen Devices angelegt, ohne `CAP_NET_ADMIN` keine Netzwerkinterfaces konfiguriert, oder ohne `CAP_SYS_ADMIN` keine Dateisystem gemountet werden können.

Weitere kernelseitigen Features sind Quotas für Kontexte (statt für User oder Gruppen) und Mittel zum Erzeugen ausbruchsicherer `chroot(2)` Umgebungen. Für Details sei hier auf den Artikel von Herbert Pötzl verwiesen.

2.1 Die Userspace Tools

Unter „VServer“ wird im Folgenden die Gesamtheit von chroot-Umgebung und den Konfigurationsdaten bezeichnet. Die chroot-Umgebung enthält Programme, Libraries und Daten, welche vom virtuellen Server ausgeführt bzw. verwendet werden.

Wie schon erwähnt, existieren „vserver“ und „util-vserver“ als Userspace Toolsets, welche sowohl simple Syscallwrapper zur Verfügung stellen, als auch umfangreiche Programme zur VServer -Verwaltung. Da „util-vserver“ ein Fork von Version 0.23 des „vserver“ Paketes ist, bieten beide Toolsets ähnliche Kommandos; die chroot-Umgebungen sind meist unter `/vservers/ < id >` zu finden und die Konfigurationsdaten unter `/etc/vservers` .

Von „util-vserver“ existieren zwei Branches: der stable Branch ist – bis auf ein paar Bugfixes und kleinere Erweiterungen – nahezu identisch mit „vserver“ . Die sehr gute Dokumentation für „vserver“ lässt sich dafür unverändert verwenden und es existiert eine große Nutzerbasis. Die Konfiguration erfolgt im Prinzip über ein einziges Bash-Skriptlet wie das in Beispiel 2.1 .

Beispiel 2.1: `/etc/vservers/ftp.conf`

```
IPROOT="192.168.5.32 192.168.5.64"
IPROOTDEV=eth0
S_HOSTNAME=ftp.nowhe.re
ONBOOT=yes
S_DOMAINNAME=
S_NICE=5
S_FLAGS="lock nproc fakeinit"
ULIMIT="-HS -u 200"
S_CAPS=""
```

Dazu kommt noch ein optionales Skript, welches Aufgaben beim Starten und Stoppen der VServer übernimmt. Insgesamt ist das eine sehr einfache Konfiguration, die viel Handarbeit für Routineaufgaben erfordert.

Die Anfälligkeit gegen Symlinkattacken und andere Races ist ein weiterer Nachteil des stable Branches, weshalb vom Einsatz in feindlichen Umgebungen wie root-Server für Kunden abzuraten ist. So werden beim

[1] siehe `uname(2)`
[2] siehe `/usr/include/linux/capability.h`

VServer -Startup die Anweisungen aus Beispiel 2.2 als root auf dem Host ausgeführt. Die in diesem Beispiel dargestellten Symlinks hätten recht unangenehme Folgen...

Beispiel 2.2: Symlinkattacke, stable Branch

```
cd /vservers/$1
...
touch var/run/utmp
rm -f  var/lock/subsys/*

/vservers/ftp
‘-- var
|-- lock
|   ‘-- subsys -> /lib
‘-- run
‘-- utmp -> /etc/nologin
```

Ähnliche Angriffspunkte und Races ziehen sich durch das gesamte Toolset, so dass ein Redesign erforderlich war und sich die weitere Entwicklung vollständig darauf konzentriert. Wegen mangelnder Entwicklerressourcen werden neuere Kernelfeatures deshalb nicht vom stable Branch unterstützt.

2.2 util-vserver

Das aktuelle „util-vserver“ Paket ist das Ergebnis dieses Redesigns. Die Ziele waren die Resistenz gegen Symlinkangriffe und Races, eingebaute Lösungen für Standardaufgaben wie dem Mounten von Verzeichnissen, die Unterstützung der aktuellen Kernelfeatures und leichte Erweiterbarkeit.

Insgesamt ist ein Großteil der Basiskommandos vom alten „vserver“ / „util-vserver“ Toolset erhalten geblieben. Jedoch sind viele Befehle sicher und leistungsfähiger reimplementiert worden, und es entstand eine Vielzahl neuer Kommandos.

Die augenfälligste Änderung dürfte das neue Konfigurationsschema sein, welches dem Ein-Eintrag-pro-Zeile bzw. Ein-Eintrag-pro-Datei Paradigma folgt. Die in Beispiel 2.1 angegebene Konfiguration würde durch Beispiel 2.3 ersetzt.

Dieses Modell verabschiedet sich von feststehenden Pfaden; ein VServer wird jetzt durch den Pfad des Konfigurationsverzeichnisses (hier: `/etc/vservers/ftp`) identifiziert; der `vdir` Symlink zeigt auf die chroot-Umgebung. Natürlich existieren noch Standardpfade, so dass zum Beispiel **vserver ftp start** äquivalent zu **vserver /etc/vservers/ftp start** ist, und dieser VServer unter `/vservers/ftp` erzeugt wird.

Die Resistenz gegen Symlinkattacken wird dadurch gewährleistet, dass Aktionen nur nach dem *sicheren* Betreten von Verzeichnissen durchgeführt werden. So wird zum Beispiel durch

```
# cd /vservers/ftp
&
&
exec-cd /var/ftp /bin/mount --bind -o ro /var/ftp '.'
```

Beispiel 2.3: `/etc/vservers/ftp` Verzeichnis

```
/etc/vservers/ftp
|-- capabilities
|-- context
|-- flags
|-- fstab
|-- interfaces
|   |-- 00
|   |   |-- ip
|   |   `-- name
|   |-- bcast
|   |-- dev
|   `-- mask
|-- run -> /var/run/vservers/ftp
|-- run.rev -> ../.defaults/run.rev
`-- vdir -> /etc/vservers/.defaults/vdirbase/ftp
```

und dem in Abbildung 1 dargestelltem Programm das `/var/ftp` Verzeichnis sicher nach `/vservers/ftp/var/ftp` gemountet [3] . Eventuelle Symlinks wie

```
/vservers/ftp
`-- var
|-- ftp  ->  /var/lib/ftp
`-- lib
`-- ftp
```

würden sich nur auf den VServer beziehen, und das Verzeichnis in diesem Fall in `/vservers/ftp/var/lib/ftp` eingehängt werden.

Eine weitere Vorsichtsmaßnahme ist, dass Daten innerhalb eines VServers kein Vertrauen entgegengebracht wird. Für einen Angreifer wäre es sonst wahrscheinlich leicht möglich, durch Manipulation der RPM-Datenbank einen Overflow in **rpm –root /vservers/ftp -qa** zu erzwingen.

3 Vserver-Management

Für einfachste Anwendungen kann ein VServer ein einzelnes Kommando sein, welches in einem eigenen Kontext in seiner chroot-Umgebung ausgeführt wird. Häufiger ist es jedoch, dass in der chroot-Umgebung eine komplette Linuxdistribution installiert ist und die bereitgestellten Dienste durch entsprechende Initskripte gestartet werden.

[3] Achtung: die „`-o ro`“ Option funktioniert nur mit dem BME Patch

```
// Usage: exec-cd {dir} {cmd} {args}*
old_fd = open("/", O_RDONLY);
chroot(".");
chdir(argv[1]);
new_fd = open(".", O_RDONLY);
fchdir(old_fd);
chroot(".");
fchdir(new_fd);
execv(argv[2], argv+2);
```

Abbildung 1: exec-cd Programm

3.1 Erzeugung

Da ein gemeinsamer Kernel für VServer und Host zur Anwendung kommt, sind herkömmliche Installationen via CD-ROM oder PXE-Boot für VServer nicht möglich, und da jede Distribution seine eigenen Wege zum fertigen System hat, lässt sich ein einheitliches Installationsverfahren kaum finden.

Stattdessen implementiert „util-vserver“ Methoden für die gängigsten Distributionen; nämlich `apt-rpm` für Fedora bzw. Red Hat basierende VServer und `debootstrap` für Debian VServer . Zum Erzeugen einer allgemeinen Verzeichnisstruktur und der Konfigurationsdaten steht ein generisches `skeleton` Verfahren zur Verfügung.

Angewendet werden diese durch das **vserver ... build** Kommando; Hilfe ist – wie bei anderen Befehlen auch – durch Anhängen von `--help` erhältlich [4] . Als Beispiel seien die in Abbildung 2 angegebenen Befehle genannt.

```
# vserver test0 build -m apt-rpm --hostname test0.nowhe.re \
--interface 10.0.1.0 --netdev eth0 --netprefix 23 \
--context 42 -- -d fc1
# vserver test1 build -m debootstrap --hostname test1.nowhe.re \
--interface 10.0.1.1 --netdev eth0 --netprefix 23 \
--context 43 -- -d sarge
# vserver test2 build -m skeleton --hostname test2.nowhe.re \
--interface 10.0.1.2 --netdev eth0 --netprefix 23 \
--context 44
```

Abbildung 2: VServer -Erstellung

Ihnen ist gemeinsam, dass jeweils ein VServer mit dem Namen `testX` erstellt wird, welcher auch im jeweiligen Hostnamen zur Anwendung kommt. Desweiteren werden IPv4-Adressen auf Netzwerkinterface `eth0` in einem /23er Netz zugewiesen und dem VServer jeweils ein eindeutiger Kontext zugeordnet.

Auf dem ersten VServer wird Fedora Core 1 mittels der `apt-rpm` Methode installiert; der zu verwendende Mirror muss in `/etc/vservers/.distributions/fc1/apt/sources.list` vor Erstverwendung aktiviert werden. Das Verfahren braucht bei 100Mb/s Netzanbindung weniger als einer Minute, nimmt ca. 70 MB Plattenplatz in Anspruch und installiert 50 Pakete.

[4] Speziell bei diesem Kommando ist die Stellung von `--help` wichtig; so zeigt zum Beispiel **vserver –help** nur allgemeine Informationen, während **vserver - build –help** Optionen für den Buildprozess ausgibt.

Das zweite Kommando verwendet Debians `debootstrap` Paket, welches bei Bedarf (z.B. bei Red Hat Hostsystemen) automatisch heruntergeladen wird. Der Zeitbedarf ist geringfügig länger als bei `apt-rpm` und es werden 130 Pakete installiert. Nach Abzug temporärer Dateien bewegt sich der belegte Plattenplatz im Bereich von 80 MB.

Mittels `skeleton` wird der in Beispiel 3.1 dargestellte minimale Dateibaum erzeugt und es können die benötigten Dateien per Hand hineinkopiert werden.

Beispiel 3.1: Dateibaum

```
/vservers/test2
|-- dev
|   |-- full
|   |-- null
|   |-- ptmx
|   |-- pts
|   |-- random
|   |-- tty
|   |-- urandom
|   ‘-- zero
|-- etc
|   ‘-- resolv.conf
‘-- proc


/etc/vservers/test2/
|-- apps
|   ‘-- init
|-- context
|-- fstab
|-- interfaces
|   |-- 0
|   |   ‘-- ip
|   |-- dev
|   ‘-- prefix
|-- name
|-- run -> /var/run/vservers/test2
|-- run.rev -> /etc/vservers/.defaults/run.rev
|-- uts
|   ‘-- nodename
‘-- vdir -> /etc/vservers/.defaults/vdirbase/test2
```

Die Devices unter `/dev` werden durch alle Methoden erzeugt, da dies wegen fehlender `CAP_MKNOD` Capabilities später nicht mehr möglich ist. Ebenso können häufig verwendete Dateien wie `/etc/resolv.conf` oder `/etc/localtime` automatisch in neuerstellte VServer übernommen werden.

Nach der Basisinstallation können mittels dem weiter unten vorgestellten Paketmanagement die dienstbezogenen Pakete wie HTTP-Server, SSH-Server und -Klienten u.ä. installiert werden.

3.2 Start

Mit Standardmethoden erstellte VServer sollten sofort durch **vserver ... start** in Betrieb zu nehmen sein. Durch diesen Befehl wird

- ein neuer Namespace erzeugt,
- eventuelle Netzwerkinterfaces erstellt,
- Verzeichnisse gemountet,
- der Prozess- und Netzwerkkontext angelegt, und
- der initiale Prozess im VServer gestartet. Dieses Programm kann das reguläre `/sbin/init` sein, was aber oftmals eine Menge unerwünschter Aktionen wie das Auslesen der Hardwareuhr, Filesystemchecks, Verzeichnismounten u.ä. zur Folge hat. Das meiste davon schlägt wegen nicht vorhandenen Capabilities fehl und erzeugt zumindest störende Ausgaben. Deshalb ist es oftmals vorteilhafter, ein im init-Prozess später ausgeführtes Programm oder Skript direkt aufzurufen; bei Red Hat wäre dies zum Beispiel **/etc/rc.d/rc 3** .

An dieser Stelle wäre zu beachten, dass mindestens ein Prozess im Kontext laufen muss. Wenn `init` sofort zurückkehren würde (zum Beispiel wegen Fehler oder keinem konfiguriertem Dienst), würde der Kontext und der VServer ebenfalls beendet.

Auf die zwei [5] oben erstellten VServer angewandt, ergibt sich die Ausgabe in Abbildung 3 . Mittels `--debug` Switch lassen sich die ausgeführten Kommandos anzeigen; für den `test0` Fedora Core 1 VServer wäre dies unter anderem die Ausgabe in Abbildung 4 .

```
# vserver test0 start
Systemlogger starten:                                      [  OK  ]
Kernellogger starten:                                      [  OK  ]
# vserver test1 start
Starting system log daemon: syslogd.
Starting kernel log daemon: klogd.
Starting MTA: exim4.
Starting internet superserver: inetd.
Starting deferred execution scheduler: atd.
Starting periodic command scheduler: cron.
#
```

Abbildung 3: VServer -Startup

3.3 Stop

Natürlich müssen einmal gestartete VServer auch wieder gestoppt werden – spätestens beim Reboot des Hostes. Dazu dient das **vserver ... stop** Kommando, welches entweder

[5] der mit der `skeleton` -Methode erstellte `test2` VServer ist nicht lauffähig und wird deswegen nicht weiter betrachtet

```
++ /usr/sbin/chbind --silent --ip 10.0.1.0/23 -- \
/usr/sbin/vcontext --create --silent --xid 42 -- \
/usr/sbin/vnamespace --set -- \
/usr/sbin/vlimit --dir /etc/vservers/test0/rlimits --missingok -- \
/usr/sbin/vsched --xid self -- \
/usr/sbin/vuname --xid self --dir /etc/vservers/test0/uts --missingok -- \
/usr/sbin/vuname --xid self --set -t context=/etc/vservers/test0 -- \
/usr/sbin/vattribute --set --secure -- \
/usr/lib/util-vserver/save_ctxinfo /etc/vservers/test0 \
/usr/sbin/vcontext --migrate-self --endsetup --chroot --silent -- \
/etc/rc.d/rc 3
```

Abbildung 4: **vserver –debug test0 start**

- ein `SIGINT` Signal an den Initprozess des VServer sendet, welches einen Shutdown-Prozess initiiert, oder
- wie schon beim Starten, ein Initskript direkt aufruft; bei Red Hat wäre dies **/etc/rc.d/rc 6**

Nun werden eventuell übriggebliebene Prozesse im Kontext noch durch ein **vkill –xid ... -s 9** terminiert und einige wenige Aufräumarbeiten getan. Bei Verwendung von Namespaces ist ein explizites Unmounten unnötig, da dies implizit beim Beenden des letzten Prozesses geschieht.

3.4 Betreten

Zum Betreten, d.h. dem Ausführen beliebiger Kommandos inklusive einer Shell, existiert die **vserver ... exec** < **command** > , bzw. **vserver ... enter** Befehlsgruppe. Im Prinzip wird dadurch der mit `start` erzeugte Namespace und der Prozess- und Netzwerkkontext betreten. Allerdings sollte das nur für Administrationsaufgaben, aber nicht dem regulären Betrieb verwendet werden. So können Umgebensvariablen falsch gesetzt sein, und `/dev/pts` Einträge fehlen.

Besser ist es, wenn innerhalb des VServer ein SSH Server gestartet und darüber das System betreten wird. Abbildung 5 zeigt neben den zwei bekannten Test- noch einen echten Produktions- VServer ; `test1` wurde schon leicht konfiguriert.

3.5 Paketmanagement

Da übliche Linuxdistributionen in den VServern laufen, liegt der Wunsch nahe, das jeweilige Paketmanagement verwenden zu können. Der einfachste Fall ist das Nutzen von **vserver ... exec rpm/apt-get/...** .

Hier stellt sich aber das Problem des Bootstrappens – das `exec` Kommando setzt das jeweilige Binary voraus, obwohl es noch nicht installiert wurde. Viele Paketmanagement Methoden bieten Optionen wie `--root < dir >` , so dass in chroot-Umgebungen gearbeitet wird. Wie aber schon gesagt, ist dies aus Sicherheitsgründen abzulehnen und führt wahrscheinlich auch unter vertrauenswürdigen Umgebungen zu Problemen, wenn Daemons durch Installationsskripte neu gestartet werden.

```
# vserver test0 exec ps axh
640 ?        S      0:00 syslogd -m 0
1188 pts/1    R      0:00 ps axh
# vserver test1 enter
test1:/#
# uname -a
Linux delenn 2.6.5ensc-0.3 #1 Thu Apr 15 ... 2004 i686 i686 i386 GNU/Linux
# vserver test1 exec uname -a
SCO UnixWare test1.nowhe.re 7.1 #1 Sat Feb 29 ... 2003 s390 GNU/Linux
$ ssh root@www-cache
[root@www-cache root]# ps axu    # (some fields manually removed)
USER       VSZ  RSS TTY      START   TIME COMMAND
root      1372  432 ?        Apr21   0:04 init [3]
svlog       52   36 ?        Apr30   0:00 /sbin/svlogd -t ./main/main ./main/a
socklog     36   24 ?        Apr30   0:00 /sbin/socklog unix /dev/log
root        48   28 ?        Apr30   0:00 tcpserver-sshd -c 80 -q -l 0 -p -u 0
root      7232 2472 ?        06:32   0:00  \_ sshd: root@pts/3
root      2204 1224 pts/3    06:32   0:00      \_ -bash
root      2756  712 pts/3    06:36   0:00          \_ ps axuf
squid    22524 5044 ?        Apr30  41:38 squid -N -D
squid     1532  324 ?        Apr30   0:01  \_ (unlinkd)
root      1596  568 ?        Apr30   0:00 /usr/sbin/crond
[root@www-cache root]#
```

Abbildung 5: Betreten von VServern

Bei Debian bietet sich „debootstrap“ für das Bootstrappen an; bei Red Hat/Fedora wird jedoch ein etwas anderer Weg gegangen: per Default befinden sich sämtliche Paketmanagementdaten (Datenbank und Konfiguration für apt und rpm) außerhalb des VServers . Ein Wrapper für RPM bindet die Datenbank sicher in den VServer ein und lädt via `LD_PRELOAD` eine Bibliothek, welche

- `execv(3)` ersetzt, so dass Skriptlets im jeweiligen Kontext ausgeführt werden. Außerdem werden hier Schritte unternommen, um die temporär zur Verfügung gestellten Paketmanagementdaten unerreichbar für diese Skriptlets zu machen.
- die Funktionen der `getpwnam(3)` Familie überlädt. Die GNU glibc ist leider so konzipiert, dass bei Bedarf Libraries zur Namensauflösung dynamisch nachgeladen werden. Speziell bei RPM besteht dieser Bedarf nach dem Wechsel in die chroot-Umgebung, so dass unbekannte `/vservers/.../lib/libnss_*.so` Bibliotheken geladen und ausgeführt werden. Neben Sicherheitsaspekten ist dies auch funktionell bedenklich, da diese `libnss*.so` mit der Host-glibc inkompatibel sein könnten.

  Gelöst wurde dies dadurch, dass ein – optimal gegen dietlibc gelinktes – Helferprogramm sicher in der chroot-Umgebung ausgeführt wird und die Namensauflösung erledigt.

Für die Praxis sind **vrpm** , **vapt-get** oder **vserver .. install** < **pkg** > gebräuchlich und führen die jeweils angemessenen Befehle aus. Mittels **vserver ... pkgmgmt externalize**|**internalize** kann zwischen dem angesprochen

externen und internen Halten der Paketmanagementdaten umgeschaltet werden. Abbildung 6 zeigt einige Beispiele; beachtenswert ist, dass das `rpm` Paket bei externem RPM nicht selbst installiert sein muss.

```
# vrpm test0 -- -q glibc fedora-release rpm
glibc-2.3.2-101.4
fedora-release-1-3
package rpm is not installed
# vrpm test0 -- -Uvh /tmp/tetex-2.0.2-13.i386.rpm
# vapt-get test0 -- install bzip2-libs
...
Preparing...                ########################################### [100%]
1:bzip2-libs             ########################################### [100%]
Done.
# vapt-get test1 -- install libbz2-1.0
...
Unpacking libbz2-1.0 (from .../libbz2-1.0_1.0.2-1_i386.deb) ...
Setting up libbz2-1.0 (1.0.2-1) ...
```

Abbildung 6: Paketmanagement

3.6 Platzoptimierung

In den meisten Fällen wird für jeden VServer die selbe Distribution verwendet, so dass viele Dateien bzw. Pakete mehrfach auf einem physikalischen Rechner installiert sind und im Betrieb identische Binaries und Libraries geladen werden. Naheliegend wäre es, von jeder Datei nur eine Instanz zu haben, welche mittels Hardlinks in den jeweiligen VServer hineinkopiert wird. Dies würde zum einen Festplattenplatz sparen, und zum anderen den Speicherverbrauch reduzieren, da Programme und Libraries nur einmal, anstatt für jeden VServer gemappt werden müssen.

Einfache Hardlinks sind sicherheitstechnisch nicht akzeptabel, da Änderungen durch einen VServer auch für alle anderen wirksam sind. Linux besitzt leider kein Copy-On-Write (COW) oder union-Filesystem, so dass VServer eine eigene Methode implementieren musste: Dateien können durch ein „iunlink“ Flag [6] markiert werden, so dass sie gelöscht aber nicht modifiziert werden können. Das Erlauben von Lösch-Operationen ist nötig, damit Dateien zum Beispiel durch das Paketmanagement erneuert werden können.

Als low-level Instrument für dieses Flag stehen die **setattr** und **showattr** Befehle zur Verfügung, welche dieses Flag für einzelne Dateien setzen bzw. den augenblicklichen Wert anzeigen. Abbildung 7 zeigt einige Beispiele zu ihrer Verwendung.

Für das Behandeln vollständiger Verzeichnisbäume (z.B. die eines VServer) wurde das **vunify** Tool entwickelt. Dieses sucht in einem Referenzpfad bzw. Referenzvserver nach identischen Dateien, d.h. solchen mit selbem Namen, Größe, Eigentümer usw., setzt dort das iunlink Flag, löscht die ursprüngliche Datei und generiert einen Hardlink. Um Konfigurationsdateien und Nutzerdaten vor Überschreiben zu schützen, werden statische Excludelisten unterstützt und Informationen des Paketmanagements zur Erzeugung dynamischer Ausschlusslisten genutzt.

[6] tatsächlich ist es eine Kombination aus dem standardmäßig vorhandenem „`i`“ Flag und einem vserver-spezifischem Flag in den Extended Attributes (EA).

```
# touch /vservers/test0/{secure,insecure}
# setattr --iunlink /vservers/test0/secure
# ln /vservers/test0/{secure,insecure} /vservers/test1/
# showattr /vservers/test0/{secure,insecure}
---bUI- /vservers/test0/secure
---bui- /vservers/test0/insecure
# vserver test0 enter
[root@test0]# echo 'GOT YOU' >insecure
[root@test0]# echo 'GOT YOU' >secure
bash: secure: Permission denied
[root@test0]#
# vserver test1 enter
test1:/# cat /insecure
GOT YOU
```

Abbildung 7: Immutable-Unlink Flag

Bei einer vollständigen Fedora Core 1 Installation sind von diesen Ausschlusslisten ca. 30 MB Daten betroffen. Deshalb würden zum Beispiel von 20 VServern mit je 2000 MB Grösse nur insgesamt 2600 MB an Plattenplatz belegt.

4 Referenzen

- Projekthomepage <http://linux-vserver.org>
- #vserver auf oftc.net