

Concepts of Linux VServer

7. Juni 2004

Note légale

Dieser Beitrag ist lizenziert unter der GNU Free Documentation License.

Note légale

Dieser Beitrag ist lizenziert unter der GNU General Public License.

Zusammenfassung

A soft partitioning concept based on 'Security Contexts' which allows to create many independent Virtual Private Servers (VPS), similar to normal Linux Servers, which can be run simultaneously on one box at full speed, sharing the hardware resources.

All services, such as ssh, mail, Web and databases, can be started on such a VPS, without (or in special cases with only minimal) modification, just like on any real server.

Each virtual server has its own user account database and root password and doesn't interfere with other virtual servers, except for the fact that they share the same hardware resources.

Linux Capability System, what is it, how can it be used to improve system security, with some examples. Linux File System Attributes and Isolation Concepts.

- chroot() namespace restrictions - chcontext() process space restrictions - chbind() network restrictions

Kernel space implementation, including a short overview how the Linux Kernel works regarding processes, namespace and network. Impact on performance and possible changes in behaviour, especially regarding the network and the scheduler.

Basic examples how to use the Core Tools to create VServer Security Contexts and Network Contexts.

Further aspects of the virtualization like:

- uts_name() machine/node/domain-name - uptime VPS system uptime - reboot VPS system reboot - ipc/tgid namespace separation

Resource Limits

- process limits - scheduler limits - memory limits - per context disk limits - per context user/group quota

1 Linux-VServer Technology

(C)2004 Herbert Pötzl

All Rights Reserved.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation.

1.1 Abstract

A soft partitioning concept based on 'Security Contexts' which allows to create many independent Virtual Private Servers (VPS), similar to normal Linux Servers, which can be run simultaneously on one box at full speed, sharing the hardware resources.

All services, such as ssh, mail, Web and databases, can be started on such a VPS, without (or in special cases with only minimal) modification, just like on any real server.

Each virtual server has its own user account database and root password and doesn't interfere with other virtual servers, except for the fact that they share the same hardware resources.

1.2 Introduction

Over the years, computers have become sufficiently powerful to use virtualization for giving the illusion of many smaller virtual machines, each running a separate operating system instance.

There are several kinds of Virtual Machines (VMs) which are providing similar features and which only differ in the degree of abstraction and the methods used for virtualization.

Most of them accomplish what they do by 'emulating' some real or fictional hardware, which in turn requires 'real' resources from the Host (the machine running the VMs). This approach, used by most System Emulators (like QEMU[U1], Bochs[U2], ...), allows to run an arbitrary Guest Operating System, even for a different Architecture (CPU and Hardware) without any modifications, because the Guest OS isn't aware of the fact that it isn't running on real hardware.

Some of them require small modifications or specialized drivers to be added to Host or Guest to improve performance and minimize the overhead required for the hardware emulation. Although this improves a lot on efficiency, there are still large amounts of resources being wasted in caches and mediation between Guest and Host (examples for this approach are UML[U3] and Xen[U4]).

But suppose you do not want to run many different Operating Systems simultaneously on a single box? Most applications running on a server do not require hardware access or kernel level code, and could easily share a machine with others, if they could be separated and secured ...

1.3 The Concept

Basically, a Linux Server consists of three building blocks: Hardware, Kernel and Applications. The Hardware usually depends on the provider or system maintainer and, while it has big influence on the overall performance, cannot be changed that easily, but for sure will differ from one setup to another.

The main purpose of the Kernel is to build an abstraction layer on top of the hardware to allow processes (Applications) to work with and operate on resources (Data) without knowing the details of the underlying hardware. Ideally, those processes would be completely hardware agnostic, by being written in an interpreted language and therefore not requiring any hardware specific knowledge.

Given that a system has enough resources to drive ten times the number of applications a single Linux server would usually require, why not put ten servers on that box, who will then share the available resources in an efficient manner?

Most applications will assume that they are the only one providing this service, and usually they will also assume a certain filesystem layout and environment. This requires that similar or identical services, for example only differing in their addresses, have to be coordinated. This usually requires a great deal of administrative work which usually reduces overall stability and security.

Therefore the basic idea is to separate the user-space environment into distinct units (sometimes called Virtual Private Server) in such way that each VPS looks and feels like a real server to the processes contained within.

Although different Linux Distributions use (sometimes heavily) patched kernels to provide special support for unusual hardware or extra functionality, most Linux Distributions are not tied to a special kernel, but they bring their own set of tools, and applications.

Linux-VServer uses this fact to allow several distributions to be run simultaneously on a single, shared kernel, without direct access to the hardware, and share the resources in a very efficient way.

1.4 Existing Infrastructure

Recent Linux Kernels already provide many security features which are utilized by Linux-VServer to do its work. Especially the Linux Capability System, the Resource Limits, File Attributes and the Change Root Environment are used by Linux-VServer. The following sections will give a short overview about each of these.

1.4.1 Linux Capability System

In computer science, a capability is a token used by a process to prove that it is allowed to perform an operation on an object. But there is something different, called "POSIX Capabilities" which was designed to split up the all powerful root privilege into a set of distinct privileges.

POSIX Capabilities A process has three sets of bitmaps called the inheritable(I), permitted(P), and effective(E) capabilities. Each capability is implemented as a bit in each of these bitmaps which is either set or unset.

When a process tries to do a privileged operation, the operating system will check the appropriate bit in the effective set of the process (instead of checking whether the effective uid of the process is 0 as is normally done).

For example, when a process tries to set the clock, the Linux kernel will check that the process has the CAP_SYS_TIME bit (which is currently bit 25) set in its effective set.

The permitted set of the process indicates the capabilities the process can use. The process can have capabilities set in the permitted set that are not in the effective set.

This indicates that the process has temporarily disabled this capability. A process is allowed to set a bit in its effective set only if it is available in the permitted set. The distinction between effective and permitted exists so that processes can "bracket" operations that need privilege.

The inheritable capabilities are the capabilities of the current process that should be inherited by a program executed by the current process. The permitted set of a process is masked against the inheritable set during exec(). Nothing special happens during fork() or clone(). Child processes and threads are given an exact copy of the capabilities of the parent process.

Although this concept is intriguing, the implementation in Linux stopped at this point, whereas POSIX Capabilities[U5] would require to add capability sets to files too, to replace the SUID flag (at least for executables)

Capability Overview The list of POSIX Capabilities used with Linux is long, and the 32 available bits are almost used up. While the detailed list of all capabilities can be found in /usr/include/linux/capability.h on most Linux systems, an overview of "considered important" capabilities is given here.

0 CAP_CHOWN

change file ownership and group.

5 CAP_KILL

send a signal to a process with a different real or effective user ID

6 CAP_SETGID

permit setgid(2), setgroups(2), and forged gids on socket credentials passing

7 CAP_SETUID

permit set*uid(2), and forged uids on socket credentials passing

8 CAP_SETPCAP

transfer/remove any capability in permitted set to/from any pid

9 CAP_LINUX_IMMUTABLE

allow modification of S_IMMUTABLE and S_APPEND file attributes

11 CAP_NET_BROADCAST

permit broadcasting and listening to multicast

12 CAP_NET_ADMIN

permit interface configuration, IP firewall, masquerading, accounting, socket debugging, routing tables, bind to any address, enter promiscuous mode, multicasting, ...

13 CAP_NET_RAW

permit usage of RAW and PACKET sockets

16 CAP_SYS_MODULE

insert and remove kernel modules

- 18 CAP_SYS_CHROOT
 permit chroot(2)
- 19 CAP_SYS_PTRACE
 permit ptrace() of any process
- 21 CAP_SYS_ADMIN
 this list would be too long, it basically allows to do everything else, not mentioned in another capability.
- 22 CAP_SYS_BOOT
 permit reboot(2)
- 23 CAP_SYS_NICE
 allow raising priority and setting priority on other processes, modify scheduling
- 24 CAP_SYS_RESOURCE
 override resource limits, quota, reserved space on fs, ...
- 27 CAP_MKNOD
 permit the privileged aspects of mknod(2)

Also see Examples [E01],[E02], and [E03].

1.4.2 Resource Limits

Resources for each process can be limited by specifying a Resource Limit. Similar to the Linux Capabilities, there are two different limits, a Soft Limit and a Hard Limit.

The soft limit is the value that the kernel enforces for the corresponding resource. The hard limit acts as a ceiling for the soft limit: an unprivileged process may only set its soft limit to a value in the range from zero up to the hard limit, and (irreversibly) lower its hard limit. A privileged process may make arbitrary changes to either limit value, as long as the soft limit stays below the hard limit.

Limit-able Resource Overview The list of all defined resource limits can be found in /usr/include/asm/resource.h on most Linux systems, an overview of "relevant" resource limits is given here.

- 0 RLIMIT_CPU
 CPU time in seconds. process is sent a SIGXCPU signal after reaching the soft limit, and SIGKILL on hard limit.
- 4 RLIMIT_CORE
 maximum size of core files generated
- 5 RLIMIT_RSS
 number of pages the process's resident set can consume (the number of virtual pages resident in RAM)
- 6 RLIMIT_NPROC
 The maximum number of processes that can be created for the real user ID of the calling process.
- 7 RLIMIT_NOFILE
 Specifies a value one greater than the maximum file descriptor number that can be opened by this process.
- 8 RLIMIT_MEMLOCK
 The maximum number of virtual memory pages that may be locked into RAM using mlock() and mlockall().
- 9 RLIMIT_AS
 The maximum number of virtual memory pages available to the process (address space limit).

Also see Examples [E11], and [E12].

1.4.3 File Attributes

While this feature started as applicable for ext2 only, all major filesystem now implement a basic set of File Attributes which allow to change certain properties. Here again a short overview of the possible attributes, and what they mean.

- 's' SECRM
When a file with this attribute set is deleted, its blocks are zeroed and written back to the disk.
- 'u' UNRM
When a file with this attribute set is deleted, its contents are saved.
- 'c' COMPR
files marked with this attribute are automatically compressed on write and uncompressed on read. (not implemented yet)
- 'S' SYNC
updates to the file contents are done synchronously
- 'i' IMMUTABLE
A file with this attribute cannot be modified: it cannot be deleted or renamed, no link can be created to this file and no data can be written to the file.
- 'a' APPEND
files with this attribute set can only be opened in append mode for writing.
- 'd' NODUMP
if this flag is set, the file is not candidate for backup with the dump utility.
- 'A' NOATIME
prevents updating the atime record on files when they are accessed or modified.
- 't' NOTAIL
A file with the 't' attribute will not have a partial block fragment at the end of the file merged with other files.
- 'D' DIRSYNC
changes to a directory having this attribute set will be done synchronously.

Also see Examples [E13] and [E14].

1.4.4 The chroot(1) Command

chroot allows you to run a command with a different root directory. This means that all filesystem lookups are done with '/' referring to the new root directory and not to the original one.

While the Linux chroot implementation isn't very secure, it increases the isolation of processes regarding the filesystem, and, if used properly allows to create a filesystem jail for a single process or a restricted user, daemon or service.

See Example [E15]

1.5 Required Modifications

This chapter will describe the essential modifications to implement something like Linux-VServer.

1.5.1 Context Separation

The separation mentioned in the Concepts section requires some modifications to the kernel to allow for the notion of Contexts.

The purpose of this "Context" is to hide all processes outside of its scope, and prohibit any unwanted interaction between a process inside the context and a process belonging to another context.

This separation requires the extension of some existing data structures in order for them to become aware of the newly introduced context, and to allow to have the same uid in different contexts, and still be able to differentiate between them.

It also requires to define a 'default' context which is used when the host system is booted, and to work around the issues resulting from some false assumptions made by some user-space tools (like pstree) that the 'init' process has to exist and to be running under id '1'

To simplify administration while simultaneously allowing for a process overview, the Host Context is basically treated like the other contexts, at least regarding the isolation, and a special 'Spectator' context is used for looking at all processes at once.

See Examples [E21],[E22] and [E23]

1.5.2 Network Separation

While the Context Separation is sufficient to isolate groups of processes, a different kind of separation or better limitation is required to confine processes to a subset of available network addresses.

Several issues have to be considered when doing so, for example the fact that bindings to special addresses like IPADDR_ANY or the local host address have to be handled in a very special way.

Note, that Linux-VServer doesn't make use of virtual network devices (yet) to avoid the resulting overhead, so socket binding and packet transmission has to be adapted.

See Example [E24]

1.5.3 The Chroot Barrier

One major problem of the chroot() system used in Linux, which modifies the 'current' root directory for the process, lies within the fact that this information is volatile, and will be changed on the 'next' chroot() Syscall.

One simple method to escape from a chroot-ed environment, is to create or open a file and keep the file-descriptor, then chroot into a subdirectory at equal or lower level with regards to the file. This causes the 'root' to be moved 'down' in the filesystem, then use fchdir() on the file descriptor to escape from that 'new' root, and consequently from the 'old' one as well, as this was lost in the last chroot() Syscall.

While early Linux-VServer versions tried to fix this by funny methods, the recent version use a special marking, known as the Chroot Barrier, on each 'root' directory, which prevents unauthorized modification and escape from the confinement.

1.5.4 Upper Bound for Caps

Because the current Linux Capability system does not provide the filesystem part, which is required to make set uid and set gid executables secure, and because it is much safer to have a secure upper bound for all processes within a context, an additional per context capability mask has been added to limit all processes belonging to that context to this mask.

The meaning of the capability bound mask is exactly the same as the permitted capability set, but for all processes, and all other capability masks.

1.5.5 Resource Isolation

Most resources are somewhat shared among the different Contexts, but some of them require additional Isolation, either to avoid security issues or to allow for improved accounting.

Those resources are:

- shared memory, IPC
- user and process IDs

- file xid tagging
- Unix ptys
- sockets

1.5.6 Filesystem XID Tagging

Although it can be disabled completely, this modification is required for filesystem level security and context isolation. It also is mandatory for Context Disk Limits and Per Context Quota Support on a shared partition.

While the idea to add the context id (xid) to each file in order to make the context ownership persistent sounds simple, the actual implementation is non trivial - mainly because adding this information either requires to change the on disk representation of the filesystem or to use some tricks.

One non intrusive approach to avoid modification of the underlying filesystem is to use the upper, mostly unused bits of existing fields, like those for UID and GID to store the additional XID.

Having context information available with each inode, it seems logical to extend the access controls to check against context too.

Currently all inode access restrictions are extended to check for the context, with special exceptions for the Host Context and the Spectator Context.

Untagged files belong to the Host Context and are silently treated as if they belong to the current context, which is required for Unification. If such a file is modified from inside a Context, it silently migrates to the new one, changing its xid.

The following Tagging Methods are implemented:

- UID32/GID32 or EXTERNAL

This format uses, up to now unused space within the disk inode to store the context information, this is currently only defined for ext2/ext3 but will be also defined for xfs, reiserfs, and jfs as soon as possible. Advantage: you'll have full 32bit uid/gid values.

- UID32/GID16

This format uses the upper half of the group id to store the context information. This is done transparently, so you'll never notice, except if you change the format without prior file conversion. Advantage: works on all 32bit U/GID FSs. Drawback: GID is reduced to 16 bit.

- UID24/GID24

This format uses the upper quarter of user and group id to store the context information, again transparently. You'll end up with 16 million user and group ids, which should suffice for the majority of all applications. Advantage: works on all 32bit U/GID FSs. Drawback: UID and GID is reduced to 24 bit.

See Examples [E31] and [E32]

1.6 Additional Modifications

In addition to the bare minimum, there are a number of modifications, not really required but extremely useful, and very nice to have, so they were added over time.

1.6.1 Context Flags

It was very soon discovered that some features require a flag, a kind of switch to turn them on and off separately for each Linux-VServer, so a simple flag-word was added.

Nowadays this flag word supports quite a number of flags, a flag word mask, which allows to tell what flags are available, and a special trigger mechanism, providing one-time flags, set on startup, that can only be cleared once, usually causing a special action or event.

Here a list of planned and mostly implemented Context Flags, available in the development branch, together with a short description.

- 0 VXF.INFO.LOCK (legacy, obsoleted)
- 1 VXF.INFO.SCHED (legacy, obsoleted)
schedule all processes in a context as if they were one.
- 2 VXF.INFO.NPROC (legacy, obsoleted)
limit the number of processes in a context to the initial NPROC value.
- 3 VXF.INFO.PRIVATE (legacy)
do not allow to join this context from outside.
- 4 VXF.INFO.INIT (legacy)
show the 'init' process with pid '1'
- 5 VXF.INFO.HIDE (legacy, obsoleted)
- 6 VXF.INFO.ULIMIT (legacy, obsoleted)
- 7 VXF.INFO.NSPACE (legacy, obsoleted)
- 8 VXF.SCHED.HARD
activate the Hard CPU scheduling
- 9 VXF.SCHED.PRIO
use the context token bucket for calculating the process priorities
- 10 VXF.SCHED.PAUSE
put all processes in this context on the hold queue, not scheduling them any longer
- 16 VXF.VIRT.MEM
virtualize the memory information so that the VM and RSS limits are used for meminfo and friends
- 17 VXF.VIRT.UPTIME
virtualize the uptime, beginning with the time of context creation
- 18 VXF.VIRT.CPU
- 24 VXF.HIDE.MOUNT
show empty proc/{pid}/mounts
- 25 VXF.HIDE.NETIF
hide network interfaces and addresses not permitted by the network context

See Example [XX]

1.6.2 Context Capabilities

As the Linux Capabilities have almost reached the maximum number that is possible without heavy modifications to the kernel, it came naturally that adding a simplified version of context capabilities to each context would ease a lot of things.

This capability set doesn't need to be visible to the processes within a context, as they would not know how to modify or verify it. Instead they act like some kind of fine tuning to existing capabilities.

In general there are two ways to use those capabilities:

- require one or a number of context capabilities to be set in addition to a given Linux capability, each one controlling a distinct part of the functionality
for example the CAP_NET_ADMIN could be split into RAW and PACKET sockets, so you could take away each of them separately by not providing the required context capability

- consider the context capability sufficient for a specified functionality, even if the Linux Capability says something different
for example mount() requires CAP_SYS_ADMIN which adds a dozen other things we do not want, so we define a CCAP_MOUNT to allow mounts for certain contexts.

The difference between the Context Flags and the Context Caps is more an abstract logical separation than a functional one, because they are handled very similar.

Again a list of the Context Capabilities and their purpose.

0 VXC_SET_UTSNAME

allow the Context to change the host and domain name with the appropriate kernel Syscall

1 VXC_SET_RLIMIT

allow the Context to modify the resource limits (within the vserver limits).

16 VXC_SECURE_MOUNT

permit 'secure' mounts, which at the moment means that the 'nodev' mount option is added.

1.6.3 Context Accounting

Some properties of a Context are useful to the admin, either for keeping an overview of the resources, to get a feeling for the capacity of the host, or for billing them in some way to the customer.

There are two different kinds of accountable properties, those having a current value which represents the 'state' of the system (for example the speed of a vehicle), and those which monotonic increase over time (like the mileage).

Most of the 'state' type of properties, also qualify for applying some limits, so they are handled special, which is described in the next section.

Good candidates for Context accounting are:

- Amount of CPU Time spent
- Number of Forks done
- Socket Messages by Type
- Network Packets Transmitted and Received

See Example [E41]

1.6.4 Context Limits

Most properties related to system resources, might it be the memory consumption, the number of processes or file-handles, or the current network bandwidth, qualify for imposing limits on them.

To provide a general framework for all kinds of limits, the Context Limits allow to set three different values for each limit-able resource: the minimum, a soft limit and a hard limit (maximum).

At the time this is written, only the hard limits are supported and not all of them are actually enforced, but here is a list of current and planned Context Limits:

- process limits
- scheduler limits
- memory limits
- per context disk limits
- per context user/group quota

See Example [E42]

1.6.5 Virtualization

One major difference between the Linux-VServer approach and Virtual Machines is that you do not have the 'virtualization' part as a side-effect, so you have to do that 'by hand' where it makes sense.

For example, a Virtual Machine does not need to think about uptime, because naturally the running OS was started somewhere in the past and will not have any problem to tell the time it 'thinks' it is running.

A Context can also store the time when it was created, but that will be different from the systems uptime, so in additions, there has to be some function, which 'adjusts' the values passed from kernel to user-space depending on the context the process belongs to.

This is what for Linux-VServer is known as Virtualization (actually it's more faking some values passed to and from the kernel to make the processes 'think' that they are on a different machine).

Currently modified for the purpose of Virtualization are:

- System Uptime
- Host and Domain Name
- Machine Type and Kernel Version
- Context Memory Availability
- Context Disk Space

See Example [E43]

1.6.6 Improved Security

Proc-FS Security provides a mechanism to protect dynamic entries in the proc filesystem from being seen in every context.

The system consists of three flags for each Proc-FS entry: Admin, Watch and Hide.

The Hide flag enables or disables the entire feature, so any combination with the Hide flag cleared will mean total visibility.

The Admin and Watch flags determine where the 'hidden' entry remains visible, so for example if Admin and Hidden are set, the Host Context will be the only one able to see this specific entry.

See Example [E44] and [E45]

1.6.7 Kernel Helper

For some purposes, it makes sense to have an user-space tool to act on behalf of the kernel, when a process inside a context requests something usually available on a real server, but naturally not available inside a context.

The best, and currently only example for this is the Reboot Helper, which allows to handle the reboot() system call, invoked from inside a Context, from Host side user-space, and to take appropriate actions - either reboot or just shutdown (halt) the specified Context.

While the helper is designed to be flexible and handle different things in a similar way there are no other users of this helper at the moment, and it might be replaced by an event interface in near future.

See Example [XX]

1.7 Features and Bonus Material

1.7.1 Unification

Because one of the central objectives for Linux-VServer is to reduce the overall resource usage wherever possible, a truly great idea was born how to 'share' files between different Contexts without interfering with the usual administrative tasks or reducing the level of security created by the isolation.

Files common to more than one Context, which are not very likely going to change, like libraries or binaries, can be hard linked on a shared filesystem, thus reducing the amount of disk space, inode caches, and even memory mappings for shared libraries.

The only drawback is that without additional measures, a malicious Context would be able to deliberately or accidentally destroy or modify such shared files, which in turn would harm the other Contexts.

One step is to make the shared files immutable (by using the Immutable File Attribute and removing the Linux Capability required to modify this attribute). However an additional attribute is required to allow removal of such immutable shared files, to allow for updates of libraries or executables from inside a Context.

Such hard linked, immutable but unlink-able files belonging to more than one Context are called 'unified' and the process of finding common files and preparing them in this way is called Unification.

The reason for doing this is reduced resource consumption, not simplified administration. While a typical Linux Server install will consume about 500MB of disk space, 10 unified servers will only need about 700MB and as a bonus use less memory for caching.

See Example [XX]

1.7.2 Private Namespaces

A recent addition to the Linux-VServer branch was the introduction of Private Namespaces. This uses the already existing Virtual Filesystem Layer of the Linux kernel to create a separate 'view' of the filesystem for the processes belonging to a Context.

The major advantage over the shared namespace used by default is that any modifications to the namespace layout (like mounts) do not affect other Contexts, not even the Host Context.

Obviously the drawback of that approach is that entering such a Private Namespace isn't as trivial as changing the root directory, but with proper kernel support this will completely replace the chroot() in the future.

1.7.3 The Linux-VServer Proc-FS

A structured, dynamically generated subtree of the well known Proc-FS - actually two of them - has been created to allow for inspecting the different values of Security and Network Contexts.

```
/proc/virtual
.../info

/proc/virtual/&lt;pid&gt;
.../info
.../status
.../sched
.../cvirt
.../cacct
.../limit
```

1.7.4 Token Bucket Extensions

While the basic idea of Linux-VServer is a peaceful coexistence of all Contexts, sharing the common resources in a respectful way, it is sometimes useful to control the resource distribution for resource hungry processes.

The basic principle of a Token Bucket is not very new, but it is given here as example for the Hard CPU Limit. The same would also apply to Scheduler Priorities, Network Bandwidth limitation and resource control in general.

The Hard CPU Limit uses this mechanism in the following way: consider a bucket of a certain size S which is filled with a specified amount of tokens R each interval T , until a maximum M is reached - excess tokens are spilled. At each timer tick, a running process consumes exactly one token from the bucket, unless the bucket is empty - in which case the process is put on a hold queue until the bucket has been refilled with a minimum N of tokens. The process is then rescheduled.

A major advantage of a Token Bucket is that a certain amount of tokens can be accumulated in times of quiescence, which later can be used to burst when resources are required.

Where a per process Token Bucket would allow for a CPU resource limitation of a single process, a Context Token Bucket allows to control the CPU usage of all confined processes.

Another approach, which is also done, is to use the current fill level of the bucket to adjust the process priority, thus reducing the priority of processes belonging to excessive Contexts.

See Example [XX]

1.7.5 Context Disk Limits

This Feature requires the use of XID Tagged Files, and allows for independent Disk Limits for different Contexts on a shared partition.

The number of inodes and blocks for each filesystem is accounted, if an XID-Hash was added for the Context-Filesystem combo.

Those values, including current usage, maximum and reserved space, will be shown for filesystem queries, creating the illusion that the shared filesystem has a different usage and size, for each Context.

1.7.6 Per Context Quota

Similar to the Context Disk Limits, Per Context Quota allows to have separate quota hashes for different Contexts on a shared filesystem. This is not required to allow for Linux-VServer quota on separate partitions.

1.7.7 The VRoot Proxy Device

Quota operations (ioctl's) require some access to the block device, which usually is, for security reasons, not available inside a VPS

1.7.8 Stealth

For some applications, for example the preparation of a honey-pot or an especially realistic imitation of a real server for educational purposes, it can make sense to make the Context indistinguishable from a real server.

However, since other freely available alternatives like QEMU or UML are much better at this, and require much less effort, this is not a central issue in Linux-VServer development.

1.8 Linux-VServer Security

Now that we know what the Linux-VServer framework provides and how some features work, let's have a word on security, because you should not rely on the framework to be secure per definition. Instead, you should exactly know what you are doing.

1.8.1 Secure Capabilities

Currently the following Linux Capabilities are considered secure, if you add others to them, you will probably open some security hole.

- CAP_CHOWN
- CAP_DAC_OVERRIDE
- CAP_DAC_READ_SEARCH
- CAP_FOWNER
- CAP_FSETID
- CAP_KILL
- CAP_SETGID

- CAP_SETUID
- CAP_NET_BIND_SERVICE
- CAP_SYS_CHROOT
- CAP_SYS_PTRACE
- CAP_SYS_BOOT
- CAP_SYS_TTY_CONFIG
- CAP_LEASE

CAP_NET_RAW for example is not considered secure although it is often used to allow the 'broken' ping command to work, although there are better alternatives like the hping[U6] command or poink[U7].

1.8.2 The Chroot Barrier

Ensuring that the Barrier flag is set on the root directory of each VPS is vital if you do not want VPS root to escape from the confinement and walk your Host's root filesystem.

1.8.3 Secure Device Nodes

The /dev directory of a VPS should not contain more than the following devices and the one directory for the unix pts tree.

- c 1 7 full
- c 1 3 null
- c 5 2 ptmx
- c 1 8 random
- c 5 0 tty
- c 1 9 urandom
- c 1 5 zero
- d pts

Of course you may add other device nodes like console, mem and kmem, even block and character devices, but you should know exactly what you are doing.

1.8.4 Secure Proc-FS Entries

There has been no detailed evaluation of secure and unsecure entries in the proc filesystem, but there have been some incidents where unprotected (not protected via Linux Capabilities) writable proc entries caused mayhem.

For example /proc/sysrq-trigger is something which should not be accessible inside a VPS without a very good reason.

1.9 Field of Application

The primary goal of this project is to create virtual servers sharing the same machine. A virtual server operate like a normal Linux server. It runs normal services such as telnet, mail servers, web servers, SQL servers.

1.9.1 Administrative Separation

Of course this allows a clever provider to sell something called Virtual Private Server, which uses less resources than other virtualization techniques, which in turn allows to put more units on a single machine.

The list of providers doing so is relatively long, and so this is rightfully considered the main area of application.

1.9.2 Service Separation

Separating different or similar services which otherwise would interfere with each other, either because they are poorly designed or because they are simply incapable of peaceful coexistence for whatever reason, can be easily done with Linux-VServer.

But even on the old fashioned real server machines, putting some extremely exposed or untrusted, because unknown or proprietary, services into some kind of jail can improve maintainability and security a lot.

1.9.3 Enhancing Security

While it can be interesting to run several virtual servers in one box, there is one concept potentially more generally useful. Imagine a physical server running a single virtual server. The goal is isolate the main environment from any service, any network. You boot in the main environment, start very few services and then continue in the virtual server.

The service in the main environment would be

- unreachable from the network.
- able to log messages from the virtual server in a secure way. the virtual server would be unable to change/erase the logs.

So even a cracked virtual server would not be able the edit the log.

- able to run intrusion detection facilities, potentially spying the state of the virtual server without being accessible or noticed.

For example tripwire could run there and it would be impossible to circumvent its operation or trick it.

Another option is to put the firewall in a virtual server, and pull in the DMZ, containing each service in a separate VPS. On proper configuration, this setup can reduce the number of required machines drastically, without impacting performance.

1.9.4 Easy Maintenance

One key feature of a virtual server is the independence from the actual hardware. Most hardware issues are irrelevant for the virtual server installation.

The main server acts as a host and takes care of all the details. The virtual server is just a client and ignores all the details. As such, the client can be moved to another physical server with very few manipulations.

For example, to move the virtual server from one physical computer to another, it sufficient to do the following:

- shutdown the running server
- copy it over to the other machine
- copy the configuration
- start the virtual server on the new machine

No adjustments to user setup, password database or hardware configuration are required, as long as both machines are binary compatible.

1.9.5 Fail-over Scenarios

Pushing the limit a little further, using some replication technology to keep two versions of the same Virtual Server, one running the other dormant, but up to the minute, would allow to do a very fast fail-over if the running server goes offline for whatever reason.

All the known methods to accomplish this, starting with network replication via rsync or drbd, via network devices or shared disk arrays, to distributed filesystems, can be utilized to reduce the down-time and improve overall efficiency.

1.9.6 For Testing

Consider a software tool or package which should be built for several versions of a specific distribution (Mandrake 8.2, 9.0, 9.1, 9.2, 10.0) or even for different distributions.

This is easily solved with Linux-VServer, and given there is plenty of disk space, the different distributions can be installed and running side by side, which simplifies switching from one to the other.

Of course this can be accomplished by chroot() alone, but with Linux-VServer it's much more realistic.

1.10 Non Intel i386 Hardware

Linux-VServer was designed to be mostly architecture agnostic, therefore only a small part, the syscall definition itself, is architecture specific. Nevertheless some architectures have private copies of basically architecture independent code for whatever reason, and therefore often small modifications are required.

Basically the following architectures are supported and some of them are even tested.

- alpha
- ia32 / ia64
- mips / mips64
- hppa / hppa64
- ppc / ppc64
- sparc / sparc64
- s390
- x86_64 (AMD64)
- uml

Adding a new architecture is relatively simple although extensive testing is required to make sure that every feature is working as expected (and of course, the hardware ;)

1.11 Linux Kernel Intro

While almost all of the described features reside in the Linux Kernel, nifty Userspace Tools are required to activate and control the new functionality.

Those Userspace Tools in general communicate with the Linux Kernel via so called System Calls (or short Syscall).

This chapter will give a short overview how Linux Kernel and User Space is organized and how Syscalls, a simple method of communication between processes and kernel, work.

1.11.1 Kernel and User Space

In Linux and similar Operating Systems, User and Kernel Space is separated, and address space is divided into two parts. Kernel space is where the kernel code resides, and user space is where the user programs live. Of course, a given user program can't write to kernel memory or to another program's memory area.

Unfortunately, this is also the case for kernel code. Kernel code can't write to user space either. What does this mean? Well, when a given hardware driver wants to write data bytes to a program in user memory, it can't do it directly, but rather it must use specific kernel functions instead. Also, when parameters are passed by address to a kernel function, the kernel function can not read the parameters directly. It must use other kernel functions to read each byte of the parameters.

Of course there are some helpers which do the transfer to and from user space.

`copy_to_user(void *to, const void *from, long n);` `copy_from_user(void *to, const void *from, long n);`

`get_user()` and `put_user()` Get or put the given byte, word, or long from or to user memory. This is a macro, and it relies on the type of the argument to determine the number of bytes to transfer.

1.11.2 Linux Syscalls

Most libc calls rely on system calls, which are the simplest kernel functions a user program can call.

These system calls are implemented in the kernel itself or in loadable kernel modules, which are little chunks of dynamically link-able kernel code.

Linux system calls are implemented through a multiplexor called with a given maskable interrupt. In Linux, this interrupt is `int 0x80`. When the '`int 0x80`' instruction is executed, control is given to the kernel (or, more accurately, to the `_system_call()` function), and the actual demultiplexing process occurs.

How does `_system_call()` work ?

First, all registers are saved and the content of the `%eax` register is checked against the global system calls table, which enumerates all system calls and their addresses.

This table can be accessed with the extern `void *sys_call_table[]` variable. A given number and memory address in this table corresponds to each system call.

System call numbers can be found in `/usr/include/sys/syscall.h`.

They are of the form `SYS_systemcallname`. If the system call is not implemented, the corresponding cell in the `sys_call_table` is 0, and an error is returned.

Otherwise, the system call exists and the corresponding entry in the table is the memory address of the system call code.

1.12 Kernel Side Implementation

While this chapter is mainly of interest to kernel developers it might be funny to take a small peek behind the curtain to get a glimpse how everything really works.

1.12.1 The Syscall Command Switch

For a long time Linux-VServer used a few different Syscalls to accomplish different aspects of the work, but very soon the number of required commands grew large, and the Syscalls started to have 'magic' values, selecting the desired behavior.

Not too long ago, a single syscall was reserved for Linux-VServer, and while the opinion on that might differ from developer to developer, it was generally considered a good decision not to have more than one syscall.

The advantage of different Syscalls would be simpler handling of the Syscalls on different architectures but this hasn't been any problem so far, as the data passed to and from the kernel has strong typed fields conforming to the C99 types.

Anyway the availability of one system call required the creation of a multiplexor, which decides, based on some selector, what specific command is to be executed, and then passes on the remaining arguments to that command, which does the actual work.

```
extern asmlinkage long
sys_vserver(uint32_t cmd, uint32_t id, void __user *data)
```

The Linux-VServer syscall is passed three arguments regardless of what actual command is specified: a command (cmd), a number (id), and a user-space data-structure of yet unknown size.

To allow some structuring for debugging purposes and some kind of command versioning, the 'cmd' is split into three parts: the lower 12 bit contain a version number, then 4 bit are reserved the upper 16 bit are divided into 8 bit command and 6 bit category, again reserving 2 bit for the future.

So there are 64 Categories with up to 256 commands in each category, allowing for 4096 revisions of each command, which to be honest is more than will ever be required.

Here is an overview of the categories already defined, and their numerical value:

Syscall Matrix V2.6

	VERSION	CREATE	MODIFY	MIGRATE	CONTROL	EXPERIM	SPECIAL	SPECIAL
	STATS	DESTROY	ALTER	CHANGE	LIMIT	TEST		
	INFO	SETUP		MOVE				
SYSTEM	VERSION	VSETUP	VHOST				DEVICES	
HOST	00	01	02	03	04	05	06	07
CPU		VPROC	PROCALT	PROCMIG	PROCTRL		SCHED.	
PROCESS	08	09	10	11	12	13	14	15
MEMORY							SWAP	
	16	17	18	19	20	21	22	23
NETWORK		VNET	NETALT	NETMIG	NETCTL		SERIAL	
	24	25	26	27	28	29	30	31
DISK							INODE	
VFS	32	33	34	35	36	37	38	39
OTHER							VINFO	
	40	41	42	43	44	45	46	47
SPECIAL					FLAGS			
	48	49	50	51	52	53	54	55
SPECIAL					RLIMIT	SYSCALL	COMPAT	
	56	57	58	59	60	TEST 61	62	63

The definition of those Commands is simplified by some macros, so for example the commands to get and set the Context Flags are defined like this:

```
#define VCMD_get_cflags VC_CMD(FLAGS, 1, 0)
```

```

#define VCMD_set_cflags          VC_CMD(FLAGS, 2, 0)

extern int vc_get_cflags(uint32_t, void __user *);
extern int vc_set_cflags(uint32_t, void __user *);

```

Note that the command itself is not passed to the actual command implementation, only the id and the pointer to user-space data.

1.12.2 Utilized Data Structures

There are many different data structures used by different parts of the implementation, and only a few examples are given here, but of course all utilized structures can be found in the source.

The Context Data Structure The Context Data Structure consists of a few fields required to manage the contexts, and handle context destruction, as well as future hierarchical contexts.

Logically separated sections of that structure like for the scheduler or the context limits are defined in separate structures, and incorporated into the main one.

```

struct vx_info {
    struct list_head vx_list;           /* linked list of contexts */
    xid_t vx_id;                       /* context id */
    atomic_t vx_refcount;              /* refcount */
    struct vx_info *vx_parent;         /* parent context */

    struct namespace *vx_namespace;    /* private namespace */
    struct fs_struct *vx_fs;           /* private namespace fs */
    uint64_t vx_flags;                 /* context flags */
    uint64_t vx_bcaps;                 /* bounding caps (system) */
    uint64_t vx_ccaps;                 /* context caps (vserver) */

    pid_t vx_initpid;                  /* PID of fake init process */

    struct _vx_limit limit;             /* vserver limits */
    struct _vx_sched sched;            /* vserver scheduler */
    struct _vx_cvirt cvirt;            /* virtual/bias stuff */
    struct _vx_cacct cacct;            /* context accounting */

    char vx_name[65];                  /* vserver name */
};

```

Here as example the Scheduler Substructure:

```

struct _vx_sched {
    spinlock_t tokens_lock; /* lock for this structure */

    int fill_rate;          /* Fill rate:      add X tokens ... */
    int interval;           /* Divisor:       ... each Y jiffies */
    atomic_t tokens;        /*               current number of tokens */
};

```

```

int tokens_min;          /* Limit:          minimum for unhold */
int tokens_max;          /* Limit:          no more than N tokens */
uint32_t jiffies;        /* bias:          integral multiple of Y */

uint64_t ticks;          /* token tick events */
cpumask_t cpus_allowed; /* cpu mask for context */
};

```

The main idea behind this separation is that each substructure belongs to a logically distinct part of the implementation which provides an init and cleanup function for this structure, thus simplifying maintainability and readability of those structures.

The Scheduler Command Data As an example for the data structure used to control a specific part of the context from user-space, here a scheduler command and the utilized data structure to set the properties:

```

#define VCMD_set_sched          VC_CMD(SCHED, 1, 2)

struct vcmd_set_sched_v2 {
int32_t fill_rate;          /* Fill rate:          add X tokens ... */
int32_t interval;          /* Divisor:            ... each Y jiffies */
int32_t tokens;            /*                    current number of tokens */
int32_t tokens_min;        /* Limit:              minimum for unhold */
int32_t tokens_max;        /* Limit:              no more than N tokens */
uint64_t cpu_mask;         /* Mask:               allowed cpus */
};

```

Example Accounting: Sockets Basically all the accounting and limit stuff is defined as macros or inline functions capable of handling the different resources, and hiding the underlying implementation wherever possible.

```

#define vx_acc_sock(v,f,p,s) \
    __vx_acc_sock((v), (f), (p), (s), __FILE__, __LINE__)

static inline void __vx_acc_sock(struct vx_info *vxi,
int family, int pos, int size, char *file, int line)
{
    if (vxi) {
        int type = vx_sock_type(family);

        atomic_inc(&vxi->acct.sock[type][pos].count);
        atomic_add(size, &vxi->acct.sock[type][pos].total);
    }
}

#define vx_sock_recv(sk,s) \
    vx_acc_sock((sk)->sk_vx_info, (sk)->sk_family, 0, (s))
#define vx_sock_send(sk,s) \

```

```

vx_acc_sock((sk)-&gt;sk_vx_info, (sk)-&gt;sk_family, 1, (s))
#define vx_sock_fail(sk,s) \
vx_acc_sock((sk)-&gt;sk_vx_info, (sk)-&gt;sk_family, 2, (s))

```

And this general definition is then used where appropriate, for example in the `_sock_sendmsg()` function like this:

```

len = sock-&gt;ops-&gt;sendmsg(iocb, sock, msg, size);
if (sock-&gt;sk) {
    if (len == size)
        vx_sock_send(sock-&gt;sk, size);
    else
        vx_sock_fail(sock-&gt;sk, size);
}

```

Example Limits: Virtual Memory

```

#define vx_pages_avail(m, p, r) \
    __vx_pages_avail((m)-&gt;mm_vx_info, (r), (p), __FILE__, __LINE__)

static inline int __vx_pages_avail(struct vx_info *vxi,
    int res, int pages, char *file, int line)
{
    if (!vxi)
        return 1;
    if (vxi-&gt;limit.rlim[res] == RLIM_INFINITY)
        return 1;
    if (atomic_read(&vxi-&gt;limit.res[res]) +
        pages < vxi-&gt;limit.rlim[res])
        return 1;
    return 0;
}

#define vx_vmpages_avail(m,p) vx_pages_avail(m, p, RLIMIT_AS)
#define vx_vmlocked_avail(m,p) vx_pages_avail(m, p, RLIMIT_MEMLOCK)
#define vx_rsspages_avail(m,p) vx_pages_avail(m, p, RLIMIT_RSS)

```

And again the test against those limits at certain places, for example here in `copy_process()`

```

/* check vserver memory */
if (p-&gt;mm && !(clone_flags & CLONE_VM)) {
    if (vx_vmpages_avail(p-&gt;mm, p-&gt;mm-&gt;total_vm)
        vx_pages_add(p-&gt;mm-&gt;mm_vx_info,
            RLIMIT_AS, p-&gt;mm-&gt;total_vm);
}

```

```

        else
            goto bad_fork_free;
    }

```

Example Virtualization: Uptime

```

void vx_vsi_uptime(struct timespec *uptime)
{
    struct vx_info *vxi = current->vx_info;

    set_normalized_timespec(uptime,
        uptime->tv_sec - vxi->cvirt.bias_tp.tv_sec,
        uptime->tv_nsec - vxi->cvirt.bias_tp.tv_nsec);
    return;
}

```

```

    if (vx_flags(VXF_VIRT_UPTIME, 0))
        vx_vsi_uptime(&uptime, &idle);

```

1.13 Future Directions

1.13.1 Hierarchical Contexts

1.13.2 Security Branch

1.13.3 Stealth Branch

1.14 Step by Step Examples

The following examples can be reproduced on recent Linux systems without any modifications.

1.14.1 [E01] Linux Caps Example

```

# grep Cap /proc/self/status

CapInh: 0000000000000000
CapPrm: 00000000fffffeff
CapEff: 00000000fffffeff

```

1.14.2 [E02] Manipulating Linux Caps

```
# lcap -z
# grep Cap /proc/self/status

CapInh: 0000000000000000
CapPrm: 0000000000000000
CapEff: 0000000000000000
```

1.14.3 [E03] CAP_MKNOD Example

```
# lcap -z CAP_MKNOD
# mknod /tmp/null b 0 0

mknod: /tmp/null: Operation not permitted
```

1.14.4 [E11] Resource Limits via BASH

```
bash-2.05# ulimit -H -a

core file size (blocks)      unlimited
data seg size (kbytes)      unlimited
file size (blocks)          unlimited
max locked memory (kbytes)  unlimited
max memory size (kbytes)    unlimited
open files                   1024
pipe size (512 bytes)       8
stack size (kbytes)         unlimited
cpu time (seconds)          unlimited
max user processes          256
virtual memory (kbytes)     unlimited
```

1.14.5 [E12] Example CPU Time Limit

```
bash-2.05# ulimit -t 10
bash-2.05# /tmp/cpuhog

Killed
```

1.14.6 [E13] Normal File without Attributes

```
# touch /tmp/file
```

```
# lsattr /tmp/file

----- /tmp/file

# rm -f /tmp/file
```

1.14.7 [E14] Immutable File Attribute

```
# touch /tmp/file
# chattr +i /tmp/file
# lsattr /tmp/file

----i----- /tmp/file

# rm -f /tmp/file

rm: unable to remove `/tmp/file': Operation not permitted

# chattr -i /tmp/file
# rm -f /tmp/file
```

1.14.8 [E15] Chroot Example

```
# mkdir /tmp/root
# mkdir /tmp/root/bin /tmp/root/lib
# cp -a /lib/* /tmp/root/lib/
# cp -a /bin/bash /tmp/root/bin/
# cp -a /bin/ls /tmp/root/bin/

# chroot /tmp/root /bin/bash

bash-2.05# ls /

bin lib
```

The now following examples require a Linux-VServer kernel and recent util-vserver tools.

1.14.9 [E21] Process Isolation

```
# chcontext ps auxw

New security context is 49152
USER          PID %CPU %MEM  VSZ  RSS TTY      STAT START   TIME COMMAND
```

```
root          37 11.5  1.7 2408  500 tts/0    R    02:36   0:00 ps auxw
```

1.14.10 [E22] Dual Context Isolation

```
# chcontext --ctx 100 sleep 111 &
# chcontext --ctx 200 sleep 222 &
# chcontext --ctx 100 ps auxw
```

New security context is 100

USER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START	TIME	COMMAND
root	23	0.3	1.5	2748	436	tts/0	S	18:47	0:00	sleep 111
root	26	28.0	2.2	2476	664	tts/0	R	18:48	0:00	ps auxw

1.14.11 [E23] Fake Init Process

```
# chcontext --flag fakeinit ps auxw
```

New security context is 49153

USER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START	TIME	COMMAND
root	1	0.0	0.4	2756	552	?	S	19:09	0:04	init

1.14.12 [E24] Network Isolation

```
# ifconfig lo 127.0.0.1
# ifconfig eth0 10.0.0.2
# ping -c 1 10.0.0.1
```

```
PING 10.0.0.1 (10.0.0.1) from 10.0.0.2 : 56(84) bytes of data.
64 bytes from 10.0.0.1: icmp_seq=0 ttl=64 time=4.814 msec
```

```
# chbind --ip 127.0.0.1 ping -c 1 10.0.0.1
```

```
ipv4root is now 127.0.0.1
connect: Invalid argument
```

1.14.13 [E31] Context File Tagging

```
# mount -o tagxid /dev/hdl/part1 /mnt/
# mkdir /mnt/test
# touch /mnt/test/file
```

```
# lsctx /mnt/test/file

#0 /mnt/test/file

# chcontext --ctx 100 touch /mnt/test/file_100
# lsctx -R /mnt/test

#0 /mnt/test/file
#100 /mnt/test/file_100
```

1.14.14 [E32] XID File Permissions

```
# mount -o tagxid /dev/hdl/part1 /mnt/
# chcontext --ctx 100 mkdir /mnt/100

New security context is 100
# chcontext --ctx 200 touch /mnt/100/file

New security context is 200
touch: /mnt/100/file: Permission denied
```

1.14.15 [E33] Context Flags

```
# chcontext --flag ^24 cat /proc/mounts
New security context is 49154
```

1.14.16 [E34] Context Capabilities

```
# chcontext --flag ^24 cat /proc/mounts
New security context is 49154
```

1.14.17 [E41] Context Accounting

```
# cat /proc/virtual/1001/cacct

UNSPEC:          0/0                0/0                0/0
UNIX:            25/1626             25/1626            0/0
INET:            12/531                48/1653            55/1625
INET6:           0/0                0/0                0/0
```

OTHER: 0/0 0/0 0/0

1.14.18 [E42] Context Limits

```
# cat /proc/virtual/1001/limit

PROC:            16/-1
VM:             14313/-1
VML:            0/-1
RSS:            5519/-1
FILES:          140/-1
```

1.14.19 [E43] Uptime Virtualization

```
# uptime

20:31:11 up 21 min, load average: 0.20, 0.05, 0.01

# chcontext --ctx 100 --flag ^17 bash -c "sleep 100; uptime"

New security context is 100
20:33:13 up 1 min, load average: 0.02, 0.03, 0.00
```

1.14.20 [E44] ProcFS Guest Security

```
# setattr --hide /proc/*
# chcontext --ctx 100 ls /proc

New security context is 100
527 self
```

1.14.21 [E45] ProcFS Host Security

```
# setattr --hide --~admin /proc/*
# ls /proc

1 10 12 13 16 2 3 4 5 532 6 7 8 9 self
```

1.14.22 [XX] Not Included Yet

1.15 Help and References

Linux-VServer is a Community Project. This means that all development and documentation is done with and by the community, and let me say it's a good and friendly community.

Most Documentation for Linux-VServer is available on the web site <http://linux-vserver.org>, which is a Wiki, so you can add your stuff there and improve the site this way.

There is also an IRC channel on irc.oftc.net called #vserver, where you always will receive a warm welcome ...

U1 QEMU CPU Emulator

<http://fabrice.bellard.free.fr/qemu/>

U2 Bochs IA-32 Emulator

<http://bochs.sourceforge.net/>

U3 User-mode Linux Kernel

<http://user-mode-linux.sourceforge.net/>

U4 The Xen virtual machine monitor

<http://www.cl.cam.ac.uk/Research/SRG/netos/xen/>

U5 POSIX.1e Reference

<http://wt.xpilot.org/publications/posix.1e/>

U6 hping tool

<http://www.hping.org/>

U7 poink tool

<http://www.gnu.org/directory/security/system/poink.html>