Linux

- Copy a GPT Partition Table to Another Disk
- Query Hardware Info
- Isolate CPUs from Kernel Scheduler
- KVM Virtual Machines
    - Bridge Zero Copy Transmit
    - PCI Passthrough
    - QEMU Device Properties
    - SR-IOV
    - Mount QCOW2
    - Direct Boot Kernel
    - Serial Only
    - EFI
- LXC
    - LXC GPU Access
    - LXC NIC Passthrough
    - netfilter/iptable logging
    - LXC USB Passthrough
- Passwords
- Serial Console
- Systemd
- Linux Unified Key Setup (LUKS)
- Reboot
- Network Storage
- Building the Kernel
    - Strip Debug

Copy a GPT Partition Table to Another Disk

Command Syntax

To clone GPT partition table command syntax are as following.

```
sgdisk -R <New_Disk> <Existing_Disk>
```

> Be sure to take note of the order of the disks. It looks like many commands with a `<from> <to>` ordering but actually `New_Disk` is an argument to the `-R` parameter.

source

Query Hardware Info

dmidecode

```
# dmidecode --list-types
bios
system
baseboard
chassis
processor
memory
cache
connector
slot
```

```
# dmidecode -t memory
```

lshw

```
# lshw -class memory
```

Isolate CPUs from Kernel Scheduler

Disable CPU(s)

Sysfs

```
# echo 0 > /sys/devices/system/cpu/cpu4/online
```

> When disabling a CPU this way any processes already assigned to this core will keep working but no new work will be assigned.

Kernel Parameter

> `isolcpus` — Isolate CPUs from the kernel scheduler.
>
> Synopsis `isolcpus= cpu_number [, cpu_number ,...]`
>
> Description Remove the specified CPUs, as defined by the cpu_number values, from the general kernel SMP balancing and scheduler algroithms. The only way to move a process onto or off an "isolated" CPU is via the CPU affinity syscalls. cpu_number begins at 0, so the maximum value is 1 less than the number of CPUs on the system.
>
> This option is the preferred way to isolate CPUs. The alternative, manually setting the CPU mask of all tasks in the system, can cause problems and suboptimal load balancer performance.

Use Isolated CPU(s)

`taskset [options] -p [mask] pid`

> taskset is used to set or retrieve the CPU affinity of a running process given its PID or to launch a new COMMAND with a given CPU affinity. CPU affinity is a scheduler property that "bonds" a process to a given set of CPUs on the system. The Linux scheduler will honor the given CPU affinity and the process will not run on any other CPUs. Note that the Linux scheduler also supports natural CPU affinity: the scheduler attempts to keep processes on the same CPU as long as practical for performance reasons. Therefore, forcing a specific CPU affinity is useful only in certain applications.

KVM Virtual Machines

Bridge Zero Copy Transmit

> Zero copy transmit mode is effective on large packet sizes. It typically reduces the host CPU overhead by up to 15% when transmitting large packets between a guest network and an external network, without affecting throughput.

Source: Red Hat - Network Tuning Techniques

```
# /etc/modprobe.d/vhost-net.conf
+ options vhost_net experimental_zcopytx=1
```

PCI Passthrough

Ensure IOMMU Is Activated

> First step of this process is to make sure that your hardware is even capable of this type of virtualization. You need to have a motherboard, CPU, and BIOS that has an IOMMU controller and supports Intel-VT-x and Intel-VT-d or AMD-v and AMD-vi. Some motherboards use different terminology for these, for example they may list AMD-v as SVM and AMD-vi as IOMMU controller.

Ensure Kernel Modules

Debian

```
# /etc/modules
# /etc/modules: kernel modules to load at boot time.
#
# This file contains the names of kernel modules that should be loaded
# at boot time, one per line. Lines beginning with "#" are ignored.
+ vfio_pci
+ vfio
+ vfio_iommu_type1
+ vfio_virqfd
```

Bind `vfio-pci` Driver to Devices

Now you can bind the `vfio-pci` driver to your devices at startup so they can be passed through to a VM. There are two ways of doing this, the first way is quick and easy but forces you to blacklist an entire driver which would stop you from being able to use that driver for another device that you aren't passing through. The second way is a little more complciated but allows you to target individual devices without blacklisting an entire driver.

1) Blacklist Drivers

By running `lspci -knn` you can easily find out which drivers are being used for a device so you know what driver to blacklist in addition to their *<vendor>:<device>* identifier. Armed with both of these we can blacklist the drivers we don't want being used and let the *vfio-pci* driver know which device(s) to bind to.

Below is an example of blacklisting the driver `i915` (Intel iGPU driver) so I can pass through my iGPU to a virtual machine. The driver is blacklisted so it won't load and the device identified by *<vendor>:<device>* is added as a parameter to the `vfio-pci` driver so it knows which device to bind with.

```
# /etc/modprobe.d/blacklist.conf
+ blacklist i915
```

```
# /etc/modprobe.d/vfio.conf
+ options vfio-pci ids=8086:3e92 disable_vga=1
```

2) Alias Devices

Using `lspci -knn` it is easy to find a devices B/D/F identifier and its *<vendor>:<device>* identifier. Then we can find its *modalias* by running `cat /sys/bus/pci/devices/<B/D/F>/modalias`. Armed with both of these we can let the *vfio-pci* module know which devices to bind to.

```
# /etc/modprobe.d/vfio.conf
+ # Intel UHD 630 (8086:3e92)
+ alias pci:v00008086d00003E92sv00001458sd0000D000bc03sc80i00 vfio-pci
+
+ options vfio-pci ids=8086:3e92 disable_vga=1
```

Rebuild `initramfs`

Debian

```
update-initramfs -u
```

Update Bootloader

Update Kernel Parameters

Grub2

```
# /etc/default/grub
- GRUB_CMDLINE_LINUX_DEFAULT="quiet"
+ GRUB_CMDLINE_LINUX_DEFAULT="quiet intel_iommu=igfx_off iommu=pt video=efifb:off"
```

Systemd

```
# /etc/kernel/cmdline
- root=ZFS=rpool/ROOT/pve-1 boot=zfs
+ root=ZFS=rpool/ROOT/pve-1 boot=zfs intel_iommu=igfx_off iommu=pt video=efifb:off
```

Rebuild Bootloader Options

Grub

```
update-grub
```

systemd-boot

```
bootctl update
```

Proxmox

```
pve-efiboot-tool refresh
```

QEMU Device Properties

Example: Rename Device

```
```

Example: Move MSI-X

> The QEMU vfio-pci device option is x-msix-relocation= which allows specifying the bar to use for the MSI-X tables, ex. bar0...bar5. Since this device uses a 64bit bar0, we can either extend that BAR or choose another, excluding bar1, which is consumed by the upper half of bar0.

To set these properties you can edit the VM configuration and add an `args` parameter.

```
args: -set device.hostpci1.x-msix-relocation=bar2
```

SR-IOV

Ensure IOMMU Is Activated

> First step of this process is to make sure that your hardware is even capable of this type of virtualization. You need to have a motherboard, CPU, and BIOS that has an IOMMU controller and supports Intel-VT-x and Intel-VT-d or AMD-v and AMD-vi. Some motherboards use different terminology for these, for example they may list AMD-v as SVM and AMD-vi as IOMMU controller.

Update Bootloader

Update Kernel Parameters

****NOTE**** Be sure to replace `intel_iommu=on` with `amd_iommu=on` if you're running on AMD instead of Intel.

Grub2

```
# /etc/default/grub
- GRUB_CMDLINE_LINUX_DEFAULT="quiet"
+ GRUB_CMDLINE_LINUX_DEFAULT="quiet intel_iommu=on iommu=pt
```

Systemd

```
# /etc/kernel/cmdline
- root=ZFS=rpool/ROOT/pve-1 boot=zfs
+ root=ZFS=rpool/ROOT/pve-1 boot=zfs intel_iommu=on iommu=pt
```

Rebuild Bootloader Options

Grub

```
update-grub
```

systemd-boot

```
bootctl update
```

Proxmox

```
pve-efiboot-tool refresh
```

Enable Virtual Functions

Find the link name you want to add virtual function to using `ip link`. In this scenario we're going to say we want to add 4 virtual functions to link `eth2`. You can find the maximum number of virtual function possible by reading the `sriov_totalvfs` from sysfs...

```
cat /sys/class/net/enp10s0f0/device/sriov_totalvfs
7
```

To enable virtual functions you just `echo` the number you want to `sriov_numvfs` in sysfs...

```
echo 4 > /sys/class/net/enp10s0f0/device/sriov_numvfs
```

Make Persistent

Sysfs is a virtual file system in Linux kernel 2.5+ that provides a tree of system devices. This package provides the program 'systool' to query it: it can list devices by bus, class, and topology.

In addition this package ships a configuration file /etc/sysfs.conf which allows one to conveniently set sysfs attributes at system bootup (in the init script etc/init.d/sysfsutils).

```
apt install sysfsutils
```

Configure sysfsutils

To make these changes persistent, you need to update `/etc/sysfs.conf` so that it gets set on startup.

```
echo "class/net/eth2/device/sriov_numvfs = 4" >> /etc/sysfs.conf
```

Mount QCOW2

Load Kernel module

```
modprobe nbd
```

Connect the image to NBD (Network Block Device) device and then mount that device/partition

```
qemu-nbd --connect=/dev/nbd0 /var/lib/vz/images/100/vm-100-disk-1.qcow2
mount /dev/nbd0p1 /mnt/somepoint/
```

When done unmount, disconnect, and if necessary unload the Kernel module.

```
umount /mnt/somepoint/
qemu-nbd --disconnect /dev/nbd0
rmmod nbd
```

Direct Boot Kernel

Provide path to Kernel and optionally initrd

```
qemu-system-aarch64 ... -kernel /boot/vmlinuz-6.9.0-rc6+ -initrd /boot/initrd.img-6.9.0-rc6+
```

Serial Only

AMD64

```
qemu-system-x86_64 ... -nographic -append "root=/dev/vda rw console=ttyS0" -hda rootfs.img
```

ARM64

```
qemu-system-aarch64 ... -nographic -append "root=/dev/vda rw console=ttyAMA0" -hda rootfs.img
```

> Some emulated consoles will need a speed appended like console=ttyAMA0,115200

EFI

To use OVMF/AAVMF for EFI add these parameters to `qemu-system-*`. Normally you can find `OVMF_CODE.fd` and `OVMF_VARS.fd` (or variants of them) in `/usr/share`

```
-drive if=pflash,format=raw,readonly,file=OVMF_CODE-pure-efi.fd
-drive if=pflash,format=raw,file=OVMF_VARS.fd
```

LXC

LXC GPU Access

Giving a LXC guest GPU access allows you to use a GPU in a guest while it is still available for use in the host machine. This is a big advantage over virtual machines where only a single host or guest can have access to a GPU at one time. Even better, multiple LXC guests can share a GPU with the host at the same time.

> The information on this page is written for a host running Proxmox but should be easy to adapt to any machine running LXC/LXD.

> Since a device is being shared between two systems there are almost certainly some security implications and I haven't been able to determine what degree of security you're giving up to share a GPU.

Determine Device Major/Minor Numbers

To allow a container access to the device you'll have to know the devices major/minor numbers. This can be found easily enough by running `ls -l` in `/dev/`. As an example to pass through the integated UHD 630 GPU from an Core i7 8700k you would first list the devices where are created under `/dev/dri`.

```
root@blackbox:~# ls -l /dev/dri
total 0
drwxr-xr-x 2 root root        80 May 12 21:54 by-path
crw-rw---- 1 root video  226,   0 May 12 21:54 card0
crw-rw---- 1 root render 226, 128 May 12 21:54 renderD128
```

From that you can see the major device number is `226` and the minors are `0` and `128`.

Provide LXC Access

In the configuration file you'd then add lines to allow the LXC guest access to that device and then also bind mount the devices from the host into the guest. In the example above since both devices share the same major number it is possible to use a shorthand notation of `226:*` to represent all minor numbers with major number `226`.

```
# /etc/pve/lxc/*.conf
+ lxc.cgroup.devices.allow: c 226:* rwm
+ lxc.mount.entry: /dev/dri/card0 dev/dri/card0 none bind,optional,create=file,mode=0666
+ lxc.mount.entry: /dev/dri/renderD128 dev/dri/renderD128 none bind,optional,create=file
```

Allow `unprivileged` Containers Access

In the example above we saw that `card0` and `renderD128` are both owned by `root` and have their groups set to `video` and `render`. Because the "unprivilged" part of LXC unprivileged container works by mapping the UIDs (user IDs) and GIDs (group IDs) in the LXC guest namespace to an unused range of IDs on host, it is necessary to create a custom mapping for that namespace that maps those groups in the LXC guest namespace to the host groups while leaving the rest unchanged so you don't lose the added security of running an unprivilged container.

First you need to give root permission to map the group IDs. You can look in `/etc/group` to find the GIDs of those groups, but in this example `video` = `44` and `render` = `108` on our host system. You should add the following lines that allow `root` to map those groups to a new GID.

```
# /etc/subgid
+ root:44:1
+ root:108:1
```

Then you'll need to create the ID mappings. Since you're just dealing with group mappings the UID mapping can be performed in a single line as shown on the first line addition below. It can be read as "remap `65,536` of the LXC guest namespace UIDs from `0` through `65,536` to a range in the host starting at `100,000`." You can tell this relates to UIDs because of the `u` denoting users. It wasn't necessary to edit `/etc/subuid` because that file already gives root permission to perform this mapping.

You have to do the same thing for groups which is the same concept but slightly more verbose. In this example when looking at `/etc/group` in the LXC guest it shows that `video` and `render` have GIDs of `44` and `106`. Although you'll use `g` to denote GIDs everything else is the same except it is necessary to ensure the custom mappings cover the whole range of GIDs so it requires more lines. The only tricky part is the second to last line that shows mapping the LXC guest namespace GID for `render` (`106`) to the host GID for `render` (`108`) because the groups have different GIDs.

```
# /etc/pve/lxc/*.conf
  lxc.cgroup.devices.allow: c 226:* rwm
  lxc.mount.entry: /dev/dri/card0 dev/dri/card0 none bind,optional,create=file,mode=0666
  lxc.mount.entry: /dev/dri/renderD128 dev/dri/renderD128 none bind,optional,create=file
+ lxc.idmap: u 0 100000 65536
+ lxc.idmap: g 0 100000 44
+ lxc.idmap: g 44 44 1
```

```
+ lxc.idmap: g 45 100045 61
+ lxc.idmap: g 106 108 1
+ lxc.idmap: g 107 100107 65429
```

Beaues it can get confusing to read I just wanted show each line with some comments...

```
+ lxc.idmap: u 0 100000 65536    // map UIDs 0-65536 (LXC namespace) to 100000-165535 (host namespace)
+ lxc.idmap: g 0 100000 44       // map GIDs 0-43 (LXC namspace) to 100000-100043 (host namespace)
+ lxc.idmap: g 44 44 1           // map GID  44 to be the same in both namespaces
+ lxc.idmap: g 45 100045 61      // map GIDs 45-105 (LXC namspace) to 100045-100105 (host namespace)
+ lxc.idmap: g 106 108 1         // map GID  106 (LXC namspace) to 108 (host namespace)
+ lxc.idmap: g 107 100107 65429  // map GIDs 107-65536 (LXC namspace) to 100107-165536 (host
namespace)
```

Add `root` to Groups

Because `root`'s UID and GID in the LXC guest's namespace isn't mapped to `root` on the host you'll have to add any users in the LXC guest to the groups `video` and `render` to have access the devices. As an example to give `root` in our LXC guest's namespace access to the devices you would simply add `root` to the `video` and `render` group.

```
usermod --append --groups video,render root
```

Potential Alernative

lxc.mount.entry - static uid/gid in LXC guest

Resources

Proxmox: Unprivileged LXC containers

LXC

LXC NIC Passthrough

On the rare occation you have a good reason to forgo the small overhead of an *veth* (Virtual Ethernet) device connected to an ethernet bridge it is possible to pass a physical network interface directly to a LXC host.

To pass a physical device you just need to provide `lxc.net.[index].type` and `lxc.net.[index].link` parameters in the LXC config. You may optionally provide a name for the link as well with `lxc.net.[index].name`. Just be sure your index value is unique among all network interfaces fot the LXC container including those Proxmox may add if you running your LXC hosts on Proxmox.

```
lxc.net.0.type: phys
lxc.net.0.link: enp1s0
# optional
lxc.net.0.name: eth0
```

netfilter/iptable logging

> Logging from network namespaces other than init has been disabled since kernel 3.10 in order to prevent host kernel log flooding from inside a container.

Source: lxc-users.linuxcontainers.narkive.com

There are two ways to get logging working on guests running in Namespaces. The first is to simply enable it on even though it is off by default due to the security concerns mentioned above. The second *and better* way is to use User space logging which doesn't carry the same restrictions because it doesn't interact with Kernel space in the same way. Besides the User space logging method being the best security practice, anytime it is possible to modify the host machine less it is better in my opinion.

Method 1: Userspace Logging (on guest)

Install `ulogd2`

```
apt install ulogd2
```

Replace `LOG` in any `iptable/netfilter` rules with `NFLOG`

```
- -A INPUT -j LOG
+ -A INPUT -j NFLOG
```

Source: lxadm.com

Method 2: Enable Logging In Namespaces (on host)

> Logging from network namespaces other than init has been disabled since kernel 3.10 in order to prevent host kernel log flooding from inside a container.
>
> If you have kernel >= 4.11 or one with commit 2851940ffee3 ("netfilter: allow logging from non-init namespaces") backported, you can enable netfilter logging from other network namespaces by...

```
sysctl net.netfilter.nf_log_all_netns=1
```

Source: lxc-users.linuxcontainers.narkive.com

This will enable all netfilter (the `nf` part in `nf_log_all_netns`) logging from namespaces until the next reboot. It can also be enabled persistently using one of the following methods...

Option 1: Always On with `sysctl.conf`

Add a single line to `sysctl.conf` so the setting gets applied at boot.

```
echo "net.netfilter.nf_log_all_netns = 1" >> /etc/sysctl.conf
```

Option 2: On Demand with Snippets (for Proxmox only)

Add a bash script to use as a `snippet`.

```
# /var/lib/vz/snippets/nf_log_all_netns.sh
+ #!/bin/bash
+
+ case $2 in
+   pre-start)
+     echo "[pre-start]"
+     echo -e "\tEnabling netfilter namespace logging."
+     echo -e "\t$(sysctl net.netfilter.nf_log_all_netns=1)"
+     ;;
+   pre-stop)
```

```
+     echo "[pre-stop]"
+     echo -e "\tDisabling netfilter namespace logging."
+     echo -e "\t$(sysctl net.netfilter.nf_log_all_netns=0)"
+     ;;
+ esac
```

Then add the "*hookscript*" to that container. If your container ID was `100` it would look like

```
$ pct set 100 -hookscript local:snippets/nf_log_all_netns.sh
```

LXC USB Passthrough

Passing through a USB device with LXC allows your LXC guest access to a physical USB device plugged into the host system.

> The information on this page is written for a host running Proxmox but should be easy to adapt to any machine running LXC/LXD.

Locate Bus/Device

```
root@vault:~# lsusb
Bus 002 Device 001: ID 1d6b:0003 Linux Foundation 3.0 root hub
Bus 001 Device 003: ID 13d3:3273 IMC Networks 802.11 n/g/b Wireless LAN USB Mini-Card
Bus 001 Device 004: ID 10c4:8a2a Silicon Labs HubZ Smart Home Controller
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
```

Determine Device Major/Minor Numbers

```
root@vault:~# ls -l /dev/bus/usb/001/004
crw-rw-r-- 1 root root 189, 3 Oct  3 17:17 /dev/bus/usb/001/004
```

From that you can see the major device number is `189` and the minor is `3`.

Provide LXC Access

In the configuration file you'd then add lines to allow the LXC guest access to that device and then also bind mount the devices from the host into the guest. In the example above since both devices share the same major number it is possible to use a shorthand notation of `189:*` to represent all minor numbers with major number `189`.

```
# /etc/pve/lxc/*.conf
+ lxc.cgroup.devices.allow: c 189:* rwm
+ lxc.mount.entry: /dev/bus/usb/001/020 dev/bus/usb/001/020 none bind,optional,create=file,mode=664
```

Allow `unprivileged` Containers Access

>

> incomplete

Resources

USB Passthrough to an LXC (Proxmox)

Passwords

Generate Random Password

With `pwgen` (generate 1 password, length 16, with a least a number and uppercase character)

```
pwgen -cns 16 1
```

Encrypt Password

With `openssl` (encrypt password from `password.txt` using SHA-512 and random salt)

```
openssl passwd -in password.txt -6
```

Paramter	Description
-salt *string*	use specified salt instead of random
-crypt	encrypt with *crypt* algorithm (default)
-1	encrypt with *MD5* algorithm
-5	encrypt with *SHA-256-crypt* algorithm
-6	encrypt with *SHA-512-crypt* algorithm

Serial Console

Output to Serial Console

Make sure the kernel is started with the following parameter...

```
console=ttyS0,115200
```

Change Size (rows/cols)

Often the expected size of the TTY session isn't what you would want and feels constrained. You can change a bunch of setting using the `stty` command. Below will change the number of columns because that is what I most often feel I need to change...

```
stty cols 140
```

Dual Output

It is possible to have the kernel write to both the standard pseudo-terminal (`tty0`) and the serial console (`ttyS0`) by adding the following kernel parameters...

```
console=ttyS0,9600 console=tty0
```

View Console

It is possible to view serial console output using the `screen` command. With a USB-to-Serial adapter plugged in you may see a device called something like `/dev/tty.usbserial-AG0JL5ZB` that will act as the tty device.

```
screen /dev/tty.usbserial-AG0JL5ZB 115200,cs8,ixon
```

Parameters explained from `man screen`

Parameter	Description
`<baud_rate>`	This affects transmission as well as receive speed (usually 300, 1200, 9600 or 19200)
`cs8` or `cs7`	Specify the transmission of eight (or seven) bits per byte
`ixon` or `-ixon`	Enables (or disables) software flow-control (CTRL-S/CTRL-Q) for sending data
`ixoff` or `-ixon`	Enables (or disables) software flow-control for receiving data
`istrip` or `-istrip`	Clear (or keep) the eight bit in each received byte

Systemd

Introduction

systemd is a software suite that provides an array of system components for Linux operating systems. Its main aim is to unify service configuration and behavior across Linux distributions; systemd's primary component is a "system and service manager"—an init system used to bootstrap user space and manage user processes.

Documentation

- systemd.unit
- systemd.service

Common Parameters

Unit

Option	Description
`Description`	A short description of the unit.
`Documentation`	A list of URIs referencing documentation.
`Before`, `After`	The order in which units are started.
`Requires`	If this unit gets activated, the units listed here will be activated as well. If one of the other units gets deactivated or fails, this unit will be deactivated.
`Wants`	Configures weaker dependencies than `Requires`. If any of the listed units does not start successfully, it has no impact on the unit activation. This is the recommended way to establish custom unit dependencies.
`Conflicts`	If a unit has a `Conflicts` setting on another unit, starting the former will stop the latter and vice versa.

Get a complete list of parameters by running `man systemd.unit`

Install

Option	Description
`Alias`	A space-separated list of additional names for the unit. Most `systemctl` commands, excluding `systemctl enable`, can use aliases instead of the actual unit name.
`RequiredBy`, `WantedBy`	The current service will be started when the listed services are started. See the description of `Wants` and `Requires` in the `[Unit]` section for details.
`Also`	Specifies a list of units to be enabled or disabled along with this unit when a user runs `systemctl enable` or `systemctl disable`.

Get a complete list of parameters by running `man systemd.unit`

Service

Option	Description
`Type`	Configures the process start-up type. One of: • `simple` (default) – starts the service immediately. It is expected that the main process of the service is defined in `ExecStart`. • `forking` – considers the service started up once the process forks and the parent has exited. • `oneshot` – similar to `simple`, but it is expected that the process has to exit before systemd starts follow-up units (useful for scripts that do a single job and then exit). You may want to set `RemainAfterExit=yes` as well so that systemd still considers the service as active after the process has exited. • `dbus` – similar to `simple`, but considers the service started up when the main process gains a D-Bus name. • `notify` – similar to `simple`, but considers the service started up only after it sends a special signal to systemd. • `idle` – similar to `simple`, but the actual execution of the service binary is delayed until all jobs are finished.

Option	Description
`ExecStart`	Commands with arguments to execute when the service is started. `Type=oneshot` enables specifying multiple custom commands that are then executed sequentially. `ExecStartPre` and `ExecStartPost` specify custom commands to be executed before and after `ExecStart`.
`ExecStop`	Commands to execute to stop the service started via `ExecStart`.
`ExecReload`	Commands to execute to trigger a configuration reload in the service.
`Restart`	With this option enabled, the service shall be restarted when the service process exits, is killed, or a timeout is reached with the exception of a normal stop by the `systemctl stop` command.
`RemainAfterExit`	If set to `True`, the service is considered active even when all its processes exited. Useful with `Type=oneshot`. Default value is `False`.

Get a complete list of parameters by running `man systemd.service`

Example

```
[Unit]
Description=The NGINX HTTP and reverse proxy server
After=syslog.target network.target remote-fs.target nss-lookup.target

[Service]
Type=forking
PIDFile=/run/nginx.pid
ExecStartPre=/usr/sbin/nginx -t
ExecStart=/usr/sbin/nginx
ExecReload=/bin/kill -s HUP $MAINPID
ExecStop=/bin/kill -s QUIT $MAINPID
PrivateTmp=true

[Install]
WantedBy=multi-user.target
```

```
[Unit]
Description=The Apache HTTP Server
```

```
After=network.target remote-fs.target nss-lookup.target

[Service]
Type=notify
EnvironmentFile=/etc/sysconfig/httpd
ExecStart=/usr/sbin/httpd $OPTIONS -DFOREGROUND
ExecReload=/usr/sbin/httpd $OPTIONS -k graceful
ExecStop=/bin/kill -WINCH ${MAINPID}
KillSignal=SIGCONT
PrivateTmp=true

[Install]
WantedBy=multi-user.target
```

```
[Unit]
Description=Redis persistent key-value database
After=network.target

[Service]
ExecStart=/usr/bin/redis-server /etc/redis.conf --daemonize no
ExecStop=/usr/bin/redis-shutdown
User=redis
Group=redis

[Install]
WantedBy=multi-user.target
```

source: shellhacks.com

Linux Unified Key Setup (LUKS)

I messed up editing this page and some of the information is missing and in the wrong order.

All the examples below assume wanting to setup a btrfs pool on two disks `/dev/sdX` and `/dev/sdY` that will be used just for additional storage.

Prepare Disks

> Before encrypting a drive, it is recommended to perform a secure erase of the disk by overwriting the entire drive with random data. To prevent cryptographic attacks or unwanted file recovery, this data is ideally indistinguishable from data later written by dm-crypt.

[Source](https://wiki.archlinux.org/title/Dm-crypt/Drive_preparation)

There are multiple ways to prepare a disk and some other potentially better ones listed on the page linked to above. Because I want to wipe my disks as quickly as possible and they were both the same size I am using a slightly more complicated method. This method creates the equivalent of filling the disks with the output from `/dev/urandom` but does so faster by using the output of encrypting `/dev/zero` and writing that to the disks instead. I save even more time by using `tee` and `process substitution` to redirect the output to both drives at once. Just for good measure I am using `pv` to measure the speed at which I am writing and to track my progress.

```
PASS=$(tr -cd '[:alnum:]' < /dev/urandom | head -c128)
openssl enc -aes-256-ctr -pass pass:"$PASS" -nosalt < /dev/zero | dd ibs=4K | pv | tee >(dd obs=64K oflag=direct of=/dev/sdX) | dd obs=64K oflag=direct of=/dev/sdY
```

Partition

Although LUKS can be layered on top of redundant storage (btrfs -or- mdadm + dm-integrity) for my usages it almost always makes sense to layer those things on top of LUKS instead. My goal is just to have an encrypted filesystem for storage of data so I only need to create one partition on each disk.

```
sgdisk --clear --new=0:0:0 /dev/sdX
sgdisk --clear --new=0:0:0 /dev/sdY
```

Encrypt

Setup LUKS with passphrase encrypted drives.

```
cryptsetup luksFormat /dev/sdX1 cryptbtrpool_1
cryptsetup luksFormat /dev/sdY1 cryptbtrpool_2
```

Create encryption key.

```
dd if=/dev/urandom bs=512 count=4 of=/etc/keyfile
```

Add keyfile as optional decryption key.

```
cryptsetup luksAddKey /dev/sdX1 /etc/keyfile
cryptsetup luksAddKey /dev/sdY1 /etc/keyfile
```

Unlock Devices

```
cryptsetup open /dev/sdX1 cryptbtrpool_1 --key-file=/etc/keyfile
cryptsetup open /dev/sdY1 cryptbtrpool_2 --key-file=/etc/keyfile
```

Create btrfs Pool

> You can use LUKS devices like any other block device and format them any way you want. However below I am combining them into a RAID1 btrfs filesystem that spans both disks and utilizes the underlying LUKS devices for encryption since btrfs doesn't support this natively.

```
mkfs.btrfs --data raid1 --metadata raid1 --label btrpool /dev/mapper/cryptbtrpool_1 /dev/mapper/cryptbtrpool_2
```

Add to `crypttab`

It is best practice to reference drives in `/etc/fstab` or `/etc/crypttab` using something more constant than just the dev name like `/dev/sdX1`. I reference the drives by the UUID of the LUKS partition but another good option is to reference the drives by their "disk-id" found under `/dev/disk/by-id/...`.

I can find the UUID for the LUKS partitions by using `blkid` and `grep` to filter the output.

```
blkid | grep LUKS
/dev/sdX1: UUID="99fc46af-1048-4c50-bc38-2085aee78579" TYPE="crypto_LUKS" PARTLABEL="Linux filesystem" PARTUUID="8cbdf3b0-7ba0-4b7b-8639-15ea3029c72e"
/dev/sdY1: UUID="507033de-5eb5-4baf-8875-6595fbb260af" TYPE="crypto_LUKS" PARTLABEL="Linux filesystem" PARTUUID="14d8331c-9a82-4e5d-8ea8-4d1a6d8025fe"
```

Now with those UUIDs I can use them in `/etc/crypttab` to automatically open my LUKS partition during boot.

```
# <target name> <source device>         <key file>      <options>
+ cryptbtrpool_1  UUID=507033de-5eb5-4baf-8875-6595fbb260af       /etc/keyfile
+ cryptbtrpool_2  UUID=99fc46af-1048-4c50-bc38-2085aee78579       /etc/keyfile
```

Add to `fstab`

Entries in `/etc/crypttab` will all have completed by the time entries in `/etc/fstab` are attempted. So knowing that the LUKS devices will have been automatically opened I can then mount that filesystem as I would any other block device. Since the btrfs filesystem above was labeled `btrpool` it is possible to mount subvolumes using a combination of that label and the names of any subvolumes that were created.

```
# /etc/fstab
# <file system> <mount point> <type> <options> <dump> <pass>
+ LABEL=btrpool   /storage/btrpool        btrfs   x-mount.mkdir=0755,defaults,subvol=@,compress=zstd      0 0
+ LABEL=btrpool   /storage/btrpool/services       btrfs   defaults,subvol=@services,compress=zstd,X-mount.mkdir=0755      0 0
+ LABEL=btrpool   /storage/btrpool/media  btrfs   defaults,subvol=@media,compress=zstd,X-mount.mkdir=0755 0 0
```

Reboot

Really Force Reboot

I've had to do this when the ZFS kernel module has a problem that was preventing shutdown/reboot commands from completing because they try and do so in a tidy way. For those situations there is the following...

> When the "reboot" or "shutdown" commands are executed daemons are gracefully stopped and storage volumes unmounted. This is usually accomplished via scripts in the /etc/init.d directory which will wait for each daemon to shut down gracefully before proceeding on to the next one. This is where a situation can develop where your Linux server fails to shutdown cleanly leaving you unable to administer the system until it is inspected locally. This is obviously not ideal so the answer is to force a reboot on the system where you can guarantee that the system will power cycle and come back up. The method will not unmount file systems nor sync delayed disk writes, so use this at your own discretion.

```
echo 1 > /proc/sys/kernel/sysrq
```

Then to reboot the machine simply enter the following:

```
echo b > /proc/sysrq-trigger
```

Network Storage

iSCSI

> Internet Small Computer Systems Interface or iSCSI (/aɪˈskʌzi/ i eye-SKUZ-ee) is an Internet Protocol-based storage networking standard for linking data storage facilities. iSCSI provides block-level access to storage devices by carrying SCSI commands over a TCP/IP network. iSCSI facilitates data transfers over intranets and to manage storage over long distances. It can be used to transmit data over local area networks (LANs), wide area networks (WANs), or the Internet and can enable location-independent data storage and retrieval.
>
> The protocol allows clients (called initiators) to send SCSI commands (CDBs) to storage devices (targets) on remote servers. It is a storage area network (SAN) protocol, allowing organizations to consolidate storage into storage arrays while providing clients (such as database and web servers) with the illusion of locally attached SCSI disks. It mainly competes with Fibre Channel, but unlike traditional Fibre Channel which usually requires dedicated cabling, iSCSI can be run over long distances using existing network infrastructure. iSCSI was pioneered by IBM and Cisco in 1998 and submitted as a draft standard in March 2000.
>
> source

Terminology/Concepts

Setup Host/Target

Install Linux target framework (tgt)

```
apt install tgt
systemctl start tgt
```

Create a new target. Be sure to replace the `TARGET_NAME` with an appropriate name. See iSCSI Addressing

```
TARGET_NAME=iqn.2023-09.home.mini-tgt-1
cat <<EOF > /etc/tgt/conf.d/$TARGET_NAME.conf
<target $TARGET_NAME>
        direct-store /dev/disk/by-id/ata-some-disk-1
        direct-store /dev/disk/by-id/ata-some-disk-2
        initiator-address 172.16.4.2
</target>
EOF
```

Setup Initiator

tbd

Building the Kernel

Strip Debug

When Kernel modules are built they often contain debug information. This creates modules with substantially larger sizes. To remove this debug information just set the environment variable when running `make/make install`

```
INSTALL_MOD_STRIP=1
```

(source)