Encypted Btrfs Root with Opt-in State on NixOS

2020-06-29

category: tech

tags: NixOS Nix

grahamc's "Erase your darlings" blog post is an amazing example of what a snapshotting filesystems (zfs) combined with an immutable, infrastructue-as-code OS (NixOS) can achieve. To summarize the post, grahamc demonstrates how to erase the root partition at boot while opting in to state by getting NixOS to symlink stuff to a dedicated partition. This restores the machine to a clean state on every boot, preserving the "new computer smell".

I believe the main selling point of this concept of **opt-in state** is that it makes it dead simple to keep track of ephemeral machine state (everything not explicitly specified by your NixOS configuration) and enforces elimination of Configuration Drift. While the benefits of this are clear for servers, this also works pretty well with workstations and laptops, where you gradually accumulate junk in `/etc` and `/var` which you never can be completely confident in deleting.[1]

Here are some notes on how to reproduce the setup with an encrypted[2] btrfs root, along with a few tips for a nicer laptop experience. The instructions for encrypted btrfs root are heavily based on this blog post.

Making a Live USB

The laptop I'm currently using is a Dell XPS-13 2-in-1 (7390) with a fair number of issues running on Linux, some of which interferes with boot. Fortunately, most of these have been fixed in newer kernels, but the default installation ISO ships an older kernel version, so we need a custom ISO. Building an ISO with a custom configuration for NixOS is shockingly simple; following the instructions on the wiki:

```
        # iso.nix
        { config, pkgs, ... }:
        {
          imports = [
            # installation-cd-graphical-plasma5-new-kernel.nix uses pkgs.linuxPackages_latest
            # instead of the default kernel.
            <nixpkgs/nixos/modules/installer/cd-dvd/installation-cd-graphical-plasma5-new-
kernel.nix>
            <nixpkgs/nixos/modules/installer/cd-dvd/channel.nix>
          ];

          hardware.enableAllFirmware = true;
          nixpkgs.config.allowUnfree = true;

          environment.systemPackages = with pkgs; [
            wget
            vim
            git
            tmux
            gparted
            nix-prefetch-scripts
```

```
          ];
        }
```

The image can be built with

```
nix-build '<nixpkgs/nixos>' -A config.system.build.isoImage -I nixos-config=iso.nix
```

Then, we write the ISO to a USB stick like so:

```
sudo dd if=./result/iso/nixos-20.03....-x86_64-linux.iso of=/dev/<usb device> bs=1M status=prc
```

NixOS Installation

Once we've booted into a graphical session, we need to partition the disk. We'll refer to the whole disk as `$DISK` (`/dev/nvme0n1` in my case), and we need three partitions. The EFI partition, swap[3], and the rest of the disk for btrfs to use, which we'll respecively refer to as `"$DISK"p1`, `"$DISK"p2`, and `"$DISK"p3`.

Btrfs doesn't natively support encryption, so we'll be using dm-crypt to transparently encrypt the partition, which would be available at `/dev/mapper/enc` after running these commands:

```
cryptsetup --verify-passphrase -v luksFormat "$DISK"p3
cryptsetup open "$DISK"p3 enc
```

We can then format each partition as needed:

```
mkfs.vfat -n boot "$DISK"p1
mkswap "$DISK"p2
swapon "$DISK"p2
mkfs.btrfs /dev/mapper/enc
```

Now we have a btrfs volume, we need to decide on how to structure our subvolumes. We want to split our data into a number of subvolumes to keep track of a few things:

- root: The subvolume for `/`, which will be cleared on every boot.
- home: The subvolume for `/home`, which should be backed up.
- nix: The subvolume for `/nix`, which needs to be persistent but is not worth backing up, as it's trivial to reconstruct.
- persist: The subvolume for `/persist`, containing system state which should be persistent across reboots and possibly backed up.
- log: The subvolume for `/var/log`. I'm not so interested in backing up logs but I want them to be preserved across reboots, so I'm dedicating a subvolume to logs rather than using the persist subvolume.

Somewhat arbitrarily, we'll go with the "Flat" layout as described in the btrfs wiki, and create our subvolumes accordingly.

```
mount -t btrfs /dev/mapper/enc /mnt

# We first create the subvolumes outlined above:
btrfs subvolume create /mnt/root
btrfs subvolume create /mnt/home
btrfs subvolume create /mnt/nix
btrfs subvolume create /mnt/persist
btrfs subvolume create /mnt/log

# We then take an empty *readonly* snapshot of the root subvolume,
# which we'll eventually rollback to on every boot.
btrfs subvolume snapshot -r /mnt/root /mnt/root-blank
```

```
umount /mnt
```

Once we've created the subvolumes, we mount them with the options that we want. Here, we're using Zstandard compression along with the `noatime` option.

```
mount -o subvol=root,compress=zstd,noatime /dev/mapper/enc /mnt

mkdir /mnt/home
mount -o subvol=home,compress=zstd,noatime /dev/mapper/enc /mnt/home

mkdir /mnt/nix
mount -o subvol=nix,compress=zstd,noatime /dev/mapper/enc /mnt/nix

mkdir /mnt/persist
mount -o subvol=persist,compress=zstd,noatime /dev/mapper/enc /mnt/persist

mkdir -p /mnt/var/log
mount -o subvol=log,compress=zstd,noatime /dev/mapper/enc /mnt/var/log

# don't forget this!
mkdir /mnt/boot
mount "$DISK"p1 /mnt/boot
```

Then, let NixOS figure out the config.

```
nixos-generate-config --root /mnt
```

This should result with `/mnt/etc/nixos/hardware-configuration.nix` looking something like this:

```
        # Do not modify this file!  It was generated by 'nixos-generate-config'
        # and may be overwritten by future invocations.  Please make changes
        # to /etc/nixos/configuration.nix instead.
        { config, lib, pkgs, ... }:

        {
          imports =
            [ <nixpkgs/nixos/modules/installer/scan/not-detected.nix>
            ];

          boot.initrd.availableKernelModules = [ "xhci_pci" "nvme" "usb_storage" "sd_mod"
"rtsx_pci_sdmmc" ];
          boot.initrd.kernelModules = [ ];
          boot.kernelModules = [ "kvm-intel" ];
          boot.extraModulePackages = [ ];

          fileSystems."/" =
            { device = "/dev/disk/by-uuid/f73c53b7-ae6c-4240-89c3-511ad918edcc";
              fsType = "btrfs";
              options = [ "subvol=root" "compress=zstd" "noatime" ];
            };

          boot.initrd.luks.devices."enc".device = "/dev/disk/by-uuid/050db9bf-0741-4150-8cf8-
d6ec12735d4c";

          fileSystems."/home" =
            { device = "/dev/disk/by-uuid/f73c53b7-ae6c-4240-89c3-511ad918edcc";
              fsType = "btrfs";
              options = [ "subvol=home" "compress=zstd" "noatime" ];
            };

          fileSystems."/nix" =
            { device = "/dev/disk/by-uuid/f73c53b7-ae6c-4240-89c3-511ad918edcc";
              fsType = "btrfs";
              options = [ "subvol=nix" "compress=zstd" "noatime" ];
```

```
    };

  fileSystems."/var/log" =
    { device = "/dev/disk/by-uuid/f73c53b7-ae6c-4240-89c3-511ad918edcc";
      fsType = "btrfs";
      options = [ "subvol=log" "compress=zstd" "noatime" ];
    };

  fileSystems."/persist" =
    { device = "/dev/disk/by-uuid/f73c53b7-ae6c-4240-89c3-511ad918edcc";
      fsType = "btrfs";
      options = [ "subvol=persist" "compress=zstd" "noatime" ];
    };

  fileSystems."/boot" =
    { device = "/dev/disk/by-uuid/8CE7-3C76";
      fsType = "vfat";
    };

  swapDevices =
    [ { device = "/dev/disk/by-uuid/5b1b6659-14ab-497f-a788-5518c25e7ec8"; }
    ];

  nix.maxJobs = lib.mkDefault 8;
  powerManagement.cpuFreqGovernor = lib.mkDefault "powersave";
  # High-DPI console
  console.font = lib.mkDefault "${pkgs.terminus_font}/share/consolefonts/ter-
u28n.psf.gz";
}
```

Make sure that this is what you want, and adjust options as necessary. Note that in order to correctly persist `/var/log`, the log subvolume needs to be mounted early enough in the boot process. To do this, we need to add `neededForBoot = true;` so the entry will look like this:

```
  fileSystems."/var/log" =
    { device = "/dev/disk/by-uuid/f73c53b7-ae6c-4240-89c3-511ad918edcc";
      fsType = "btrfs";
      options = [ "subvol=log" "compress=zstd" "noatime" ];
      neededForBoot = true;
    };
```

Although it's possible to customize `/mnt/etc/nixos/configuration.nix` at this point to set up all the things you need in one fell swoop, I recommend starting out with a reletively minimal config to make sure everything works ok. I went with something like this, with a user called `delta`:

```
{ config, pkgs, ... }:
{
  imports =
    [ # Include the results of the hardware scan.
      ./hardware-configuration.nix
    ];

  boot.kernelPackages = pkgs.linuxPackages_latest;
  boot.supportedFilesystems = [ "btrfs" ];
  hardware.enableAllFirmware = true;
  nixpkgs.config.allowUnfree = true;

  # Use the systemd-boot EFI boot loader.
  boot.loader.systemd-boot.enable = true;
  boot.loader.efi.canTouchEfiVariables = true;

  networking.hostName = "apollo"; # Define your hostname.
  networking.networkmanager.enable = true;
```

```
  # Enable the X11 windowing system.
  services.xserver.enable = true;

  # Enable the KDE Desktop Environment.
  services.xserver.displayManager.sddm.enable = true;
  services.xserver.desktopManager.plasma5.enable = true;

  # Define a user account. Don't forget to set a password with ‘passwd’.
  users.users.delta = {
    isNormalUser = true;
    extraGroups = [ "wheel" ]; # Enable ‘sudo’ for the user.
  };

  system.stateVersion = "20.03";
}
```

Take a deep breath.

```
nixos-install
reboot
```

If all goes well, we'll be prompted for the passphrase for `$DISK` entered earlier, then we'll see the greeter for the KDE Desktop Environment. Swith to another tty with `Ctrl+Alt+F1`, login as root, `passwd delta` to set your password, and switch back to KDE with `Ctrl+Alt+F7`. Once you're logged in, you can continue to tweak your NixOS configuration as you want. However, I generally recommend keeping enabled services at a minimum, and setting up opt-in state first.

Darling Erasure

Now that we're comfortable in our desktop environment of choice (mine is XMonad), we can move onto the opt-in state setup. First, we need to find out what state exists in the first place. Seeing what has changed since we took the blank snapshot seems like a good way to do this.

Taking a diff between the root subvolume and the root-blank subvolume (in btrfs, snapshots are just subvolumes) can be done with a script based off of the answers to this serverfault question.

```
#!/usr/bin/env bash
# fs-diff.sh
set -euo pipefail

OLD_TRANSID=$(sudo btrfs subvolume find-new /mnt/root-blank 9999999)
OLD_TRANSID=${OLD_TRANSID#transid marker was }

sudo btrfs subvolume find-new "/mnt/root" "$OLD_TRANSID" |
sed '$d' |
cut -f17- -d' ' |
sort |
uniq |
while read path; do
  path="/$path"
  if [ -L "$path" ]; then
    : # The path is a symbolic link, so is probably handled by NixOS already
  elif [ -d "$path" ]; then
    : # The path is a directory, ignore
  else
    echo "$path"
  fi
done
```

Then, all it takes to find out which files now exist in the root subvolume is:

```
sudo mkdir /mnt
sudo mount -o subvol=/ /dev/mapper/enc /mnt
./fs-diff.sh
```

This may show a surprisingly small list of files, or possible something fairly lengthy, depending on your configuration. We'll first tackle NetworkManager, so we don't have to re-type passwords to Wi-Fi access points after every reboot. While grahamc's original blog post suggests that simply persisting `/etc/NetworkManager/system-connections` by moving it to somewhere in `/persist` and creating a symlink is enough, this was not enough to get it to work on my XMonad setup. I ended up with something like this, symlinking a few files in `/var/lib/NetworkManager` as well.

```
  environment.etc = {
    "NetworkManager/system-connections".source = "/persist/etc/NetworkManager/system-connectic
  };
  systemd.tmpfiles.rules = [
    "L /var/lib/NetworkManager/secret_key - - - - /persist/var/lib/NetworkManager/secret_key"
    "L /var/lib/NetworkManager/seen-bssids - - - - /persist/var/lib/NetworkManager/seen-bssids
    "L /var/lib/NetworkManager/timestamps - - - - /persist/var/lib/NetworkManager/timestamps"
  ];
```

Now you might have noticed that the NixOS configuration itself lives in `/etc/nixos/`, which will be deleted if left there. After adding a few things, I ended up with a configuration like this.

```
 environment.etc = {
    nixos.source = "/persist/etc/nixos";
    "NetworkManager/system-connections".source = "/persist/etc/NetworkManager/system-connectic
    adjtime.source = "/persist/etc/adjtime";
    NIXOS.source = "/persist/etc/NIXOS";
    machine-id.source = "/persist/etc/machine-id";
  };
  systemd.tmpfiles.rules = [
    "L /var/lib/NetworkManager/secret_key - - - - /persist/var/lib/NetworkManager/secret_key"
    "L /var/lib/NetworkManager/seen-bssids - - - - /persist/var/lib/NetworkManager/seen-bssids
    "L /var/lib/NetworkManager/timestamps - - - - /persist/var/lib/NetworkManager/timestamps"
  ];
  security.sudo.extraConfig = ''
    # rollback results in sudo lectures after each reboot
    Defaults lecture = never
  '';
```

Rolling back the root subvolume is a little bit involved when compared to zfs, but can be achieved with this config.

```
  # Note `lib.mkBefore` is used instead of `lib.mkAfter` here.
  boot.initrd.postDeviceCommands = pkgs.lib.mkBefore ''
    mkdir -p /mnt

    # We first mount the btrfs root to /mnt
    # so we can manipulate btrfs subvolumes.
    mount -o subvol=/ /dev/mapper/enc /mnt

    # While we're tempted to just delete /root and create
    # a new snapshot from /root-blank, /root is already
    # populated at this point with a number of subvolumes,
    # which makes `btrfs subvolume delete` fail.
    # So, we remove them first.
    #
    # /root contains subvolumes:
    # - /root/var/lib/portables
    # - /root/var/lib/machines
    #
    # I suspect these are related to systemd-nspawn, but
    # since I don't use it I'm not 100% sure.
    # Anyhow, deleting these subvolumes hasn't resulted
```

```
    # in any issues so far, except for fairly
    # benign-looking errors from systemd-tmpfiles.
    btrfs subvolume list -o /mnt/root |
    cut -f9 -d' ' |
    while read subvolume; do
      echo "deleting /$subvolume subvolume..."
      btrfs subvolume delete "/mnt/$subvolume"
    done &&
    echo "deleting /root subvolume..." &&
    btrfs subvolume delete /mnt/root

    echo "restoring blank /root subvolume..."
    btrfs subvolume snapshot /mnt/root-blank /mnt/root

    # Once we're done rolling back to a blank snapshot,
    # we can unmount /mnt and continue on the boot process.
    umount /mnt
  '';
```

While NixOS will take care of creating the specified symlinks, we need to move the relevant file and directories to where the symlinks are pointing at after running `sudo nixos-rebuild boot` and before rebooting.

```
sudo nixos-rebuild boot

sudo mkdir -p /persist/etc/NetworkManager
sudo cp -r {,/persist}/etc/NetworkManager/system-connections
sudo mkdir -p /persist/var/lib/NetworkManager
sudo cp /var/lib/NetworkManager/{secret_key,seen-bssids,timestamps} /persist/var/lib/NetworkMa

sudo cp {,/persist}/etc/nixos
sudo cp {,/persist}/etc/adjtime
sudo cp {,/persist}/etc/NIXOS
```

Before rebooting, make sure that your user credentials are appropriately handled. Be especially careful[4] when setting `users.mutableUsers` to false and using `users.extraUsers.<name?>.passwordFile`, as these settings are some of the few in NixOS which can lock you out across NixOS configurations and require non-trivial recovery work or a reinstall. If you want declerative user management, I recommend using `users.extraUsers.<name?>.hashedPasswords`, but this has it's own downsides as well.[5]

Take another deep breath.

```
reboot
```

If something goes wrong and `/mnt/root` isn't deleted, `btrfs subvolume snapshot /mnt/root-blank /mnt/root` will just create a snapshot under `/mnt/root`, so a quick hack to check if rolling back failed without consulting `journalctl -b` is to see if `/mnt/root/root-blank` exists.[6]

Adding NixOS Services Case Study (Docker and LXD)

As much as Nix and NixOS are attractive for everyday use, sometimes the time it takes to get some language or package running on NixOS just doesn't seem worth it. That's when container runtimes like Docker and LXD can help. These tools can act as an escape hatch to get some software working quickly on your machine.

Here, we'll go through the workflow for getting NixOS services to work with opt-in state, with Docker and LXD as examples.

First, let's get Docker and LXD running to inspect what kind of state they have. Thanks to NixOS, this is a just a few lines of configuration.

```
  virtualisation = {
     docker.enable = true;
    lxd = {
      enable = true;
      recommendedSysctlSettings = true;
    };
  };
```

```
sudo nixos-rebuild switch
```

This will install, set up, and start both Docker and LXD on our machine. With `fs-diff.sh` we can see a few relevant files and directories show up.

```
/etc/docker/key.json
...
/var/lib/docker/...
/var/lib/lxd/...
```

Some quick googling tells us that `/etc/docker/key.json` is generated on every boot, so it seems like we don't need to keep this around. On the other hand, `/var/lib/docker` and `/var/lib/lxd` seem important, so let's adjust our config accordingly.

```
  systemd.tmpfiles.rules = [
    "L /var/lib/NetworkManager/secret_key - - - - /persist/var/lib/NetworkManager/secret_key"
    "L /var/lib/NetworkManager/seen-bssids - - - - /persist/var/lib/NetworkManager/seen-bssids
    "L /var/lib/NetworkManager/timestamps - - - - /persist/var/lib/NetworkManager/timestamps"
    "L /var/lib/lxd - - - - /persist/var/lib/lxd"
    "L /var/lib/docker - - - - /persist/var/lib/docker"
  ];
```

Now, stop the two services and copy over the directories.

```
sudo mkdir -p /persist/var/lib/

sudo systemctl stop lxd
sudo cp -r {,/persist}/var/lib/lxd

sudo systemctl stop docker
sudo cp -r {,/persist}/var/lib/docker

sudo nixos-rebuild boot
reboot
```

If all goes well, running the `fs-diff.sh` after reboot shouldn't show persisted directories `/var/lib/lxd` and `/var/lib/docker` since they should be symlinks which are created during the boot process.

Docker should work without any problems at this point, but we LXD needs some additional configuration. LXD requires a storage pool to operate, so we create a subvolume for LXD, and mount it in `/persist` .

```
sudo mount -o subvol=/ /mnt
sudo btrfs subvolume create /mnt/lxd
sudo umount /mnt
sudo mkdir /persist/lxd
sudo mount -o subvol=lxd /dev/mapper/enc /persist/lxd
```

Once the subvolume is ready, we run `lxd init` and answer the questions in the following manner.

```
$ lxd init
Would you like to use LXD clustering? (yes/no) [default=no]: no
```

```
Do you want to configure a new storage pool? (yes/no) [default=yes]:
Name of the new storage pool [default=default]:
Name of the storage backend to use (btrfs, dir, lvm) [default=btrfs]:
Would you like to create a new btrfs subvolume under /var/lib/lxd? (yes/no) [default=yes]: no
Create a new BTRFS pool? (yes/no) [default=yes]: no
Name of the existing BTRFS pool or dataset: /persist/lxd
Would you like to connect to a MAAS server? (yes/no) [default=no]:
Would you like to create a new local network bridge? (yes/no) [default=yes]:
What should the new bridge be called? [default=lxdbr0]:
What IPv4 address should be used? (CIDR subnet notation, “auto” or “none”) [default=auto]:
What IPv6 address should be used? (CIDR subnet notation, “auto” or “none”) [default=auto]:
Would you like LXD to be available over the network? (yes/no) [default=no]:
Would you like stale cached images to be updated automatically? (yes/no) [default=yes]
Would you like a YAML "lxd init" preseed to be printed? (yes/no) [default=no]:
```

Remember to add the relevant information to `/etc/nixos/hardware-configuration.nix` so NixOS will mount the subvolume where LXD expects (i.e. `/persist/lxd`).

```
fileSystems."/persist/lxd" =
  { device = "/dev/disk/by-uuid/f73c53b7-ae6c-4240-89c3-511ad918edcc";
    fsType = "btrfs";
    options = [ "subvol=lxd" "compress=zstd" "noatime" ];
  };
```

EDIT 2020-01-26: Added persistence for `/etc/machine-id`, which fixes an issue where journalctl fails to find logs from past boots, among various others. Thanks j-hui for pointing this out!

Thanks to cannorin and __pandaman64__ for comments and suggestions.

---

1. When using a Windows or macOS laptop, I find myself reinstalling the OS every so often to restore the machine to a clean state. Why go through this trouble if you can get your OS to do this on every boot?↩

2. Sadly, we stop short of FDE and settle for only encrypting the btrfs volume, as encrypting `/boot` seems much more complicated than I’m willing to experiment with. It’s unfortunate that desktop Linux security severely lags behind smartphones, where FDE is the norm rather than the exception, for example.↩

3. Note that I’m creating a swap partition despite having 32GB of RAM. Contrary to popular belief, you should still create swap partitions on systems with “enough RAM”. See this blog post for details: In defence of swap: common misconceptions↩

4. You may need to add `neededForBoot = true;` to `/persist`, but I haven’t verified this first-hand.↩

5. Using `hashedPasswords` has two drawbacks off the top of my head:
   - Since your configuration is kept in the Nix store, other users can read your hashed password and attempt to crack it. Note this does not happen when `users.mutableUsers = false;` since `/etc/shadow` is only root-readable.
   - Putting your configuration.nix in a public repository has similar problems. I feel this is a bigger problem, since you can no longer just `git clone https://github.com/user/dotfiles-repo` which may somewhat complicate your initial setup process.

   ↩

6. Something like `[ -d /root-blank ] && notify-send -u critical "opt-in state" "rollback failed"` would be nice to run after logging in.↩
