Encrypted Root and ZFS on NixOS

Encrypt data at rest with LUKS, OpenZFS, and NixOS.

12/21/2024 · 8 min read

homelab nixos security

This post builds on configuration in Using Nix Flakes to Configure Systems.

When we’re setting up a new machine, one of the first things we need to do is partition drives and install file systems. By default, none of the data stored on drives is encrypted. This means anyone with physical access to a drive can read its contents.

This may not be as big of a deal for a homelab. However, any sensitive information stored on a drive, like passwords or private keys, could be used to gain access into more critical systems should the device fall into an attacker’s control.

Disk Encryption

On Linux, disk encryption is generally performed by the Linux Unified Key Setup, or LUKS. It works by generating an encryption key that can only be unlocked via some secret, like a password. The disk or partition is then encrypted using that encryption key when writing data, or decrypted when reading. When the system boots, the secret must be provided, after which the drive is unlocked and data can be read freely until the system shuts down or loses power.

OpenZFS on Linux also has the ability to natively perform encryption. However, it isn’t able to be unlocked by a TPM2 module, or at least not easily (*foreshadowing*).

Wiping Root on Boot

This is less of a security feature, and more of a feature to make sure we’re declaratively configuring as much as possible. You might be wondering how it’s possible to wipe `/` but still have a functioning system. The answer is the `/nix/store`, and a tool called Impermanence. Instead of storing files across the filesystem, we can designate a filesystem for `/persist`. Files that need to persist between boots will live here, and the files or directories will be symlinked at boot, just like `/nix/store` does for configuration files and programs.

This can also *help* prevent against attackers gaining a persistent foothold in your system, but it shouldn't be relied upon.

To wipe our root filesystem at boot, we'll use OpenZFS. OpenZFS operates on volumes of one or more disks, can natively implement redundancy features akin to RAID, or mirroring, and provides fast backups via snapshots. Snapshots are particularly useful to us as that is what will allow us to quickly reset to a clean slate on every boot.

This is heavily inspired by Erase your darlings by Graham Christensen

Reinstalling NixOS

To set all of this up, we'll need to reinstall the NixOS configuration we installed in the last post, this time with some tweaks. To get started, boot into the minimal installation ISO.

Creating partitions

We only need two partitions, one for `/boot`, and one for everything else. We won't be using swap as Kubernetes doesn't play nicely with it. First, identify the boot drive under `/dev/disk/by-id`.

```
ls -la /dev/disk/by-id

export DISK='/dev/disk/by-id/my-boot-drive'
```

Now we can partition the drive. Run `sudo cgdisk $DISK` and delete any partitions that may be present. Write the partition table. Now, create two partitions:

1. EFI: From first aligned sector (just press enter), 1G in size. Partition type is `ef00`. Name it whatever you want

2. root: next aligned sector (press enter), consume remainder of drive (press enter). Partition type is `8300` (press enter). Name it whatever you want.

Write the partition table then quit `cgdisk`.

Encrypting the root partition

Now we can setup LUKS on the large second partition on our drive.

```
printf "YOUR_PASSWORD" | sudo cryptsetup luksFormat --type luks2 "${DISK}-part2
printf "YOUR_PASSWORD" | sudo cryptsetup luksOpen "${DISK}-part2" luks-rpool -
```

Just like that, we've setup encryption for the root partition, and unlocked the device.

Creating the ZFS pool and datasets

```
zpool create \
    -o ashift=12 \
    -o autotrim=on \
    -R /mnt \
    -O acltype=posixacl \
    -O canmount=off \
    -O dnodesize=auto \
    -O normalization=formD \
    -O relatime=on \
    -O xattr=sa \
    -O mountpoint=none \
    rpool \
    /dev/mapper/luks-rpool

# rpool/local/* is generated from nixos or wiped entirely
zfs create -o canmount=noauto -o mountpoint=legacy rpool/local/root

# create ZFS snapshot that we'll rollback to on boot
zfs snapshot rpool/local/root@blank

zfs create -o mountpoint=legacy rpool/local/nix

# rpool/safe/* is backed up regularly
zfs create -o mountpoint=legacy rpool/safe/home
zfs create -o mountpoint=legacy rpool/safe/persist

# mount datasets and boot drives
mount -o X-mount.mkdir -t zfs rpool/local/root /mnt
mount -o X-mount.mkdir -t zfs rpool/local/nix /mnt/nix
mount -o X-mount.mkdir -t zfs rpool/safe/home /mnt/home
mount -o X-mount.mkdir -t zfs rpool/safe/persist /mnt/persist
mount -t vfat \
  -o fmask=0077,dmask=0077,iocharset=iso8859-1,X-mount.mkdir \
  "${DISK}-part1" /mnt/boot
```

Phew, that was a lot to type out... Thankfully we're almost done. Continue with the installation up to where the NixOS configuration is generated.

OpenZFS requires the hostId of the machine to be set so that a pool isn't imported on the wrong machine accidentally.

```
head -c 8 /etc/machine-id >> /etc/nixos/configuration.nix
```

Open `/etc/nixos/configuration.nix` in your editor and add the following lines.

```
networking.hostId = "partial machine id";

# enable SSH like previously, so we can rebuild remotely later
services.openssh.enable = true;
services.openssh.settings.PermitRootLogin = "yes";
```

We need to add a bit of configuration to `/etc/nixos/hardware-configuration.nix` to tell NixOS about the LUKS device. First, we need to grab another identifier for the root partition. Up until now, we've been using `/dev/disk/by-id/`, but LUKS can't always find that at boot, so we'll need the `/dev/disk/by-uuid/` identifier that points to the same partition (/dev/sdxN, or /dev/nvmeXnYpZ).

```
echo /dev/disk/by-uuid/my-device-uuid >> /etc/nixos/hardware-configuration.nix
```

This copies the identifier into the hardware configuration. Open it in an editor and add these lines.

```
boot.initrd.luks.devices = {
  luks-rpool.device = "/dev/disk/by-uuid/my-device-uuid";
};
```

We also need to mark `/persist` as a necessary partition for booting.

```
fileSystems."/persist" = {
  # ...
  neededForBoot = true;
};
```

Wrapping up the installation

Now, continue the rest of the installation as usual. After the system is installed and rebooted, you should be prompted for a password to unlock the root partition. Once the password is entered, the boot process should continue like normal. Repeat the process from Using Nix Flakes to Configure Systems to copy the `/etc/nixos/hardware-configuration.nix` back to your local device. Also, don't forget to add `networking.hostId` to your host's `default.nix`.

Setup Impermanence

Back on the local machine, we'll need to add some configuration to install and setup Impermanence. In `flake.nix`, add:

```
inputs = {
  # ...

  # wipe drive on boot
  impermanence.url = "github:nix-community/impermanence";
};

outputs = {
  # ...
  impermanence,
  ...
} @ inputs: let
  # ...
in {
  # ...
  nixosConfigurations = {
    kube-host-1 = lib.nixosSystem {
      system = "x86_64-linux";
      modules = [
        impermanence.nixosModules.impermanence
        ./nix/hosts/kube-host-1
      ];
      specialArgs = {inherit inputs outputs;};
    };
  };
};
```

In `nix/modules/default.nix`, add:

```
# ...
environment.persistence."/persist" = {
  enable = true;
  directories = [
    "/var/log"
    "/var/lib/nixos"
    "/var/lib/systemd/coredump"
```

```
      ];
    };
    # ...
```

Finally, let's update the remote system configuration.

```
nixos-rebuild --flake .#kube-host-1 --target-host root@$IP switch
```

We can verify that impermanence is setup. You should see two directories `lib` and `log`.

```
ssh root@$IP ls -la /persist/var
```

Rollback ZFS Snapshot on Boot

The last piece of the puzzle here is to actually configure ZFS to rollback to the snapshot at boot. We can also set some ZFS configuration best-practices here like trim and automatically scrubbing.

To do this, we'll need to make some new files and changes to our configuration. First, in `nix/modules/default.nix`, add

```
{
  # ...
}: {
  imports = [./server];
  # ...
}
```

Now, let's create `nix/modules/server/default.nix`. In the future, we'll need to set some small amount of configuration per host to influence these common settings. We can do that via the Nix module system. This allows us to define options, as well as the config that's produced from those options.

```
{
  lib,
  config,
  pkgs,
  ...
}: let
  cfg = config.rs-homelab.server;
```

```
in {
  imports = [
    ./zfs.nix
  ];

  options = {
    rs-homelab.server.enable = lib.mkEnableOption "sets up default server confi
  };

  config = lib.mkIf cfg.enable {
    rs-homelab.server.zfs.enable = lib.mkDefault true;
  };
}
```

Here, I've defined an option called `rs-homelab.server.enable`. If enabled, we'll automatically enable `rs-homelab.server.zfs.enable` by default. Let's enable `rs-homelab.server` for our `kube-host-1` host machine. In `nix/hosts/kube-host-1/default.nix`:

```
  # ...
  rs-homelab = {
    server.enable = true;
  };
  # ...
```

We're also importing a `zfs.nix` file. Let's create that now at `nix/modules/server/zfs.nix`.

```
{
  lib,
  config,
  pkgs,
  ...
}: let
  cfg = config.rs-homelab.server.zfs;
in {
  options = {
    rs-homelab.server.zfs.enable = lib.mkEnableOption "Enables common zfs setti
  };

  config = lib.mkIf cfg.enable {
    boot.initrd.systemd.enable = true;
    boot.initrd.systemd.services.rollback = {
      description = "Rollback root filesystem to a pristine state";
      wantedBy = ["initrd.target"];
      after = ["zfs-import-rpool.service"];
      before = ["sysroot.mount"];
      path = with pkgs; [zfs];
      unitConfig.DefaultDependencies = "no";
      serviceConfig.Type = "oneshot";
      script = ''
        zfs rollback -r rpool/local/root@blank && echo " >> >> Rollback Complet
      '';
    };

    services.zfs = {
      trim.enable = true;
      autoScrub.enable = true;
    };
  };
}
```

Here we can see the systemd service that runs after zfs imports the pool, but before we mount the root filesystem. When the service runs, it rolls back to the blank snapshot we took when we were setting up partitions.

If you want to know what will be removed after a boot, you can run the following zfs command on the remote system (i.e. via SSH).

```
zfs diff rpool/local/root@blank
```

If you see files you want to keep around, add them to the `` `environment.persistence."/persist".directories` `` option. Check out the impermanence docs if you want to learn more about what it can do. Of course, don't forget to apply the configuration to the homelab machine for the ZFS rollback to take effect.

Conclusion

Quite a lot was accomplished here. Our disk is no longer easily read by attackers with physical access, and we're enforcing declarative configuration on ourselves. There's a bit more to do here. For one, typing in your password every time the machine reboots is a real pain in the butt. Next up, we'll fix that and find more ways to tighten physical security.

This post is one of many about running a homelab. To view more, click on a tag at the top of the page to see more related posts.