Paranoid NixOS Setup

Published on 07/18/2021, 4172 words, 16 minutes to read

Most of the time you can get away with a fairly simple security posture on NixOS. Don't run services as root, separate each service into its own systemd units, don't run packages you don't trust the heritage of and most importantly don't give random people shell access with passwordless sudo.

Sometimes however, you have good reasons to want to lock everything down as much as humanly possible. This could happen when you want to create production servers for something security-critical such as a bastion host. In this post I'm going to show you a defense-in-depth model for making a NixOS server that is a bit more paranoid than usual, as well as explanations of all the moving parts.

High-level Ideas

At a high-level I'm assuming the following things about this setup:

- It should be very difficult to get in as a passive attacker
- But the defense doesn't stop at "just hope they don't get in"
- It should be annoying for attackers to get a user-level shell
- But ensure they'll be able to anyways if they're dedicated enough
- It should be difficult for attackers to run their own code on the system
- But assume that it could happen and make evidence of that very loud
- It should be aggravating for attackers to access the package manager on the system
- But ensure that they can't do anything very easily even if they can access the package manager itself

Some additional goals:

- Make the system only manageable by a central management system such as morph or nixops
- Only make SSH visible over a VPN of some kind, such as Tailscale or another WireGuard setup
- Mount the root filesystem on a tmpfs
- Have explicitly defined persistent folders
- Mark everything as `noexec` except for the mount that `/nix/store` is on
- Don't make the system too difficult to use in the process

Cadey

Disclaimer: I am a Tailscale employee. Tailscale did not review this post for accuracy or content, though this setup is based on conversations I've had with a coworker at Tailscale.

Along the way we'll be making a system that I'm naming `meeka`. We'll put its configuration in a folder named `meeka`:

```
# hosts/meeka/configuation.nix
{ ... }:

{
  networking.hostName = "meeka";
  services.openssh.enable = true;
}
```

Low-hanging Fruit

There are some easy things we can get out of the way. One of the biggest ways that people get in is to make services visible to attack in the first place.

The Firewall

Let's get one of the lowest-hanging fruits out of the way: the firewall. Most of the background radiation of the internet is in the form of automated probes to development ports and SSH traffic. NixOS actually includes a firewall by default! You can see more information on how to configure it here, but here's a good collection of values to use by default:

```
# hosts/meeka/firewall.nix
{ ... }:

{
  networking.firewall.enable = true;
}
```

VPN for Access

Generally, it's probably okay to use SSH over the unprotected internet for accessing your machines. However, this is all about maximum paranoia, so we're going to use a VPN to get into the machine. Tailscale is a fairly direct thing to set up in NixOS:

```
# hosts/meeka/tailscale.nix
{ ... }:

{
  services.tailscale.enable = true;

  # Tell the firewall to implicitly trust packets routed over Tailscale:
  networking.firewall.trustedInterfaces = [ "tailscale0" ];
}
```

When you boot into the server, you can log in like normal using the `tailscale up` command. You can probably isolate down the server using ACLs if you want to make sure things are a bit more paranoid.

It may be good to set up a second way to get into the machine, just in case. I personally try to leave at least 3 ways into my servers, but the super paranoid production-facing servers should probably only be able to be connected to over a VPN of some kind.

If you want to see more about how to set up WireGuard on NixOS, see here for more information.

Locking Down the Hatches

Now that we're getting out of the easy stuff, let's go to the more defense in depth stuff. Here we're going to talk about separation of concerns and all those other fun things.

Each Service Gets its own User Account

I am going to use the word "service" annoyingly vague here. In this world, a "service" is a human-oriented view of "computer does the thing I want it to do". This website you're reading this post on could be one service, and it should have a separate account from other services. See here for more information on how to set this up.

Lock Down Services Within Systemd

systemd is a suite of tools that NixOS uses to manage a huge chunk of the system. It is kinda complicated and very large in scope, however this also means that you get access to a lot of convenient security management features. One of them is the `Protect*` unit options in `systemd.exec(5)`, which can be used to lock down permissions to the resource and system call level. Let's cover some of my favorites that you can slipstream into services:

Also take a look at `systemd-analyze security yourservicename.service`, that will give you a lot more things to search through the systemd documentation for.

`ProtectHome`/`ProtectSystem`

These options allow you to change how systemd presents critical system files and `/home` to a given process. You can use this to remove the ability for a service to modify system files or peek into user's home directories, even as root. This allows you to put a lot more limits on a service's power.

`NoNewPrivileges`

If this is set, child processes of this service cannot gain more privileges period. Even if the child process is a suid binary.

Mara

A suid binary is a binary that has the suid flag set. This makes the Linux kernel change the active user field of that binary to the owner of the binary when you run it. This is a huge part of how the magic behind sudo and ping works.

`ProtectKernel{Logs,Modules,Tuneables}`

These ones are fairly simple so I'm gonna use some bullet trees for them:

- `ProtectKernelLogs`: If set to true, the service cannot access the kernel message buffer that you get by running `dmesg` or reading from `/proc/kmsg`.
- `ProtectKernelModules`: If set to true, the service cannot load or unload kernel modules.
- `ProtectKernelTunables`: If set to true, various twiddly bits in `/proc` and `/sys` that let you control tunable values in the kernel will be made read-only. Most of the time these values are set early in the system boot process and never twiddled with again, so it's reasonable to deny a service (and its child processes) access to these

Mara

Why should I bother making all of these changes to my services though? Isn't it overkill to have a webapp running as a service user get denied access to even look at the kernel log?

Cadey

To be honest, it can look like paranoid overkill, but this isn't just for the service itself. This is for defense in *depth*, which means that you want to make sure that things are reasonably secure even if an attacker manages to get code execution on one of your services. These settings prevent the service's view of the system from having too much detail, which can make the attacking process more annoying. Remember that the goal here isn't to make the system attack-proof, nothing is. The goal is to annoy the attacker enough that they give up. This is not perfect and probably will fall apart if your enemy is the Mossad, but it's at least an attempt to lock things down just in case the attackers aren't sending their "A" game. You may also want to look into `InaccessiblePaths` to block away other folders that you deem "forbidden" as facts and circumstances demand.

Lock Down Nix Access

Nix is the package manager for NixOS. Nix can be invoked by users. Nix lets users access things like compilers and scripting languages. These can be used to run exploit tools. This can be understandably problematic from a security standpoint.

NixOS has an option called `nix.allowedUsers` that lets you specify which users or groups are allowed to do anything with the Nix daemon, and by extension the Nix package manager. For a fairly standard setup, you can probably get away with the following which allows everyone that can `sudo` to access the Nix daemon:

```
# configuration/meeka/nix.nix
{ ... }:

{
  nix.allowedUsers = [ "@wheel" ];
}
```

However if you want to prevent everyone but root, you can use a configuration like this:

```
# configuration/meeka/nix.nix
{ ... }:

{
  nix.allowedUsers = [ "root" ];
}
```

You may also want to block access to the NixOS cache CDN with an external firewall rule if you really don't trust things. You can block it by blocking the fastly IP range `151.101.0.0/16`.

Mara

I'd suggest doing this firewall change on the level above the NixOS machine itself, just in case the machine gets owned and then they ditch your firewall rules in an effort to aid in exfiltration.

Making the System Amnesiac

Most of these steps go way deep down the security rabbit hole. A lot of these are focused on limiting access to persistent storage so that persistence is opted into, not opted out of. These steps will essentially mount the root filesystem on a tmpfs that is cleared out on every reboot, with persistent data written to a subfolder in `/nix` that a symlink/bindmount farm is linked to. Most of these steps will require you to reprovision your NixOS machines and may require you to build your own custom images for cloud providers. Your experience and mileage may vary.

These steps will be based on the excellent work done in these posts/projects:

- impermanence

- NixOS ❄: tmpfs as root
- Erase your darlings

Partitioning/Setup

Normally the NixOS partition setup looks a bit like this:

- `/boot` for either BIOS boot or EFI files
- 2x ram for swap (swap is not a panacea, however it can sometimes give you valuable time to debug a problem)
- `/` for everything else

Mara

It's worth noting that technically NixOS works fine if you make only one big filesystem and put `/boot` on there directly, but this may only pan out for BIOS booting systems.

Given that `/` is going to become an in-memory tmpfs, we can instead move the partitioning to look like this:

- `/boot` for either BIOS boot or EFI files
- 2x ram for swap
- `/nix` for everything else

Assuming you are installing NixOS from scratch in a VM to test this part out, the partitioning setup commands could look something like this:

```
dev=/dev/vda # replace me with the actual device
parted ${dev} -- mklabel msdos
parted ${dev} -- mkpart primary ext4 1M 512M
parted ${dev} -- set 1 boot on
parted ${dev} -- mkpart primary ext4 512MiB 100%
mkfs.ext4 -L boot ${dev}1
mkfs.ext4 -L nix ${dev}2
```

Mara

Wait, ext4? I thought you were a zfs stan?

Mara

I normally am, however in this case it's probably better to keep the scary production servers as boring and vanilla as possible, especially when doing a more weird setup like this.

The exact size of your /boot partition may vary based on facts and circumstances, however in practice I've found 512 MB to be a not-terrible default.

Make your "root mount" with a tmpfs:

```
mount -t tmpfs none /mnt
```

Then you need to create the persistent folders on `/nix/persist`. I've found these defaults to be not-horrible:

```
mkdir -p /mnt/{boot,nix,etc/{nixos,ssh},var/{lib,log},srv}
```

Mara

We use `/srv` as the home for our services. Adjust this as your facts and circumstances demand.

Then mount those two partitions to your tmpfs:

```
mount ${dev}1 /mnt/boot
mount ${dev}2 /mnt/nix
```

And create matching folders in `/mnt/nix/persist`:

```
mkdir -p /mnt/nix/persist/{etc/{nixos,ssh},var/{lib,log},srv}
```

Then finally create some bind mounts to tie everything together for the meantime. These bindmounts will be handled by impermanence in the future, however for now the quick and dirty method will suffice:

```
mount -o bind /mnt/nix/persist/etc/nixos /mnt/etc/nixos
mount -o bind /mnt/nix/persist/var/log /mnt/var/log
```

Then generate a base config with `nixos-generate-config`:

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

And open `/etc/nixos/hardware-configuration.nix` to edit the settings for the tmpfs mount on `/`. At a high level you'll need to change this:

```
fileSystems."/" = {
  device = "none";
  fsType = "tmpfs";
  options = [ "defaults" "mode=755" ];
};
```

to this:

```
fileSystems."/" = {
  device = "none";
  fsType = "tmpfs";
  options = [ "defaults" "size=2G" "mode=755" ];
};
```

This will limit `/` to taking up 2 GB of storage at most. This will mostly contain temporary files and the like, but you should adjust this as makes sense given the amount of ram your systems have. I personally think that 512 MB could make sense depending on what you are doing.

Using Impermanence

Now we get to add impermanence to the mix to handle making all of those pesky bind mounts for us on boot. One of the easiest ways you can add its module to the nix search path is to set the `NIX_PATH` environment variable like this:

```
export NIX_PATH=nixpkgs=channel:nixos-21.05:impermanence=https://github.com/nix-community/impe
```

This will set the import path `<impermanence>` to point to the git repository for impermanence. Depending on your security needs you may want to mirror the impermanence

git repo, but keep in mind it needs to point to a tarball for Nix to understand what to do with it.

Once you have that added, you can add the impermanence configuration to your `/etc/nixos/configuration.nix`:

```
  environment.persistence."/nix/persist" = {
    directories = [
      "/etc/nixos" # nixos system config files, can be considered optional
      "/srv"       # service data
      "/var/lib"   # system service persistent data
      "/var/log"   # the place that journald dumps it logs to
    ];
  };
```

Finally you'll want to set these configuration lines for files in `/etc/sshd`. I've tried doing it directly in `environment.persistence.<name>.directories` directly but it seems to make `sshd.service` unable to generate its host keys, which is slightly important for sshd to work at all. These lines will point the files to the right places:

```
  environment.etc."ssh/ssh_host_rsa_key".source
    = "/nix/persist/etc/ssh/ssh_host_rsa_key";
  environment.etc."ssh/ssh_host_rsa_key.pub".source
    = "/nix/persist/etc/ssh/ssh_host_rsa_key.pub";
  environment.etc."ssh/ssh_host_ed25519_key".source
    = "/nix/persist/etc/ssh/ssh_host_ed25519_key";
  environment.etc."ssh/ssh_host_ed25519_key.pub".source
    = "/nix/persist/etc/ssh/ssh_host_ed25519_key.pub";
```

The machine ID may be important too if you want to read logs locally after you reboot, or if you have any services that expect the machine ID to not change.

```
  environment.etc."machine-id".source
    = "/nix/persist/etc/machine-id";
```

From here you can continue with `nixos-install` like normal (though you may want to add `--no-root-passwd` if you added a default root password to your config for bootstrap reasons only). However if you want to be lazy you can read below where I show you how to automatically create an ISO that does all this for you.

Repeatable Base Image with an ISO

Using the setup I mentioned in a past post, you can create an automatic install ISO that will take a blank disk to a state where you can SSH into it and configure it further using a tool like morph. Take a look at this folder in my nixos-configs repo for more information. Most of the magic is done with the `build` script. It's basically the last few sections of this article turned into nix files. If you build it yourself you'll want to take care with the line that looks like this:

```
users.users.root.initialPassword = "hunter2";
users.users.root.openssh.authorizedKeys.keyFiles = [ (fetchKeys "Xe") ];
```

This sets the root password to `hunter2` (a reasonably secure default for bootstrapping systems only, holy crap do not use this in production) so you can log in with the console and the list of SSH keys from here. Replace `Xe` with your GitHub username. This is not the most deterministic, but if GitHub is down you probably have bigger problems. It's also a decent crutch to help you bootstrap things. If this bothers you you can set authorized keys as normal:

```
users.users.root.openssh.authorizedKeys.keys = [
  "ssh-yolo swag420blazeit"
];
```

You can turn this into an EC2 image with something like packer.

Audit Tracing

The Linux kernel has some fancy auditing powers that are criminally under-used.

Mara

Isn't that because the audit subsystem has the ergonomics of driving a submarine down a road?

Well, yes but until I learn how to summon the right kinds of daemons, I can start with this audit rule to log every single time a program is attempted to be run:

```
# hosts/meeka/auditd.nix
{ ... }:
{
  security.auditd.enable = true;
  security.audit.enable = true;
  security.audit.rules = [
    "-a exit,always -F arch=b64 -S execve"
  ];
}
```

You can monitor these logs with `journalctl -f`. If you don't see any audit logs show up, ssh in from another window and run some commands like `ls`. You should see a flurry of them show up.

Send All Logs Off-Machine

You should really treat all system-local logs as radioactive. They are liabilities and in some cases can present problematic situations when faced with questionable interpretations of things like the GDPR. Not to mention attackers will be tempted to wipe all record of their attacks from them. I don't really have a suggestion for the best practice here, but I'm sure that people smarter than me have come up with good suggestions in my place. Either way, get them off the system as fast as possible.

You should probably have some process scraping the audit logs to check for programs outside of `/nix/store` being executed. That can sometimes point to signs of a break-in.

Optional Steps

Normally a lot of these suggestions are aimed at not totally interfering with normal usability so that in case you need to debug things you can do so with surgical precision. However, depending on your level of paranoia you may want to go a step further and disable some things that most may consider to be a "core part of basic usability". Just be aware that these things may make debugging an errant system difficult.

Rip Out `sudo`

sudo is a commonly used tool that allows users to assume superuser powers for a short amount of time. The things they do with `sudo` are logged to the system, but this project

has been known to occasionally have security issues.

Mara

Isn't that because it's written in C and C is inherently unsafe even though hordes of "experts" decry otherwise?

Cadey

Don't say that, you'll incite the horde.

NixOS lets us rip that out if we want to:

```
# hosts/meeka/sudo.nix
{ ... }:

{
  security.sudo.enable = false;
}
```

If you want to keep it around but instead limit its use to users that are in the `wheel` group, you can instead opt for something like this:

```
# hosts/meeka/sudo.nix
{ ... }:

{
  security.sudo.execWheelOnly = true;
}
```

Rip Out Default Packages

By default NixOS comes with a few packages like nano, perl and rsync to help you get started using it. These are great and all, but can be slightly incredibly problematic from a security standpoint. Rip them out like this:

```
# hosts/meeka/no-defaults.nix
{ lib, ... }:

{
  environment.defaultPackages = lib.mkForce [];
}
```

Mara

The `lib.mkForce` function forcibly overrides the contents of that value to what you give as an argument. This is useful for saying "no, heck you, I want it to be set to this no matter what anyone else says". This can be a useful hammer when correcting the security model of NixOS services when you have a good reason to.

Disable sshd Features

sshd is great. You can use it to log into systems, proxy traffic and more. sshd is also horrible because you can proxy traffic and more, turning a machine into an unexpected jumpbox for attackers. This is not ideal for machines that you don't expect to be jumpboxes. Disable this feature and some more (such as X11 forwarding, SSH agent forwarding and stream-local forwarding) like this:

```
# configuration/meeka/sshd.nix
{ ... }:

{
  services.openssh = {
    passwordAuthentication = false;
    allowSFTP = false; # Don't set this if you need sftp
    challengeResponseAuthentication = false;
    extraConfig = ''
      AllowTcpForwarding yes
      X11Forwarding no
      AllowAgentForwarding no
      AllowStreamLocalForwarding no
      AuthenticationMethods publickey
    '';
  };
}
```

Mark All Partitions but `/nix/store` as `noexec`

This is the most paranoid of the ideas in this post. The idea is that if you lock down the package manager so random services can't install software and you also make it impossible for them to write and run executable files outside of `/nix/store`, it becomes very difficult to exploit kernel bugs to get root. Add this with the other systemd isolation features that disable access to device nodes and twiddly system flags and you have a defense in

depth setup that will make an attacker's life hard. They will have to get code execution in your services to do any damage.

Keep in mind that doing this will likely break the heck out of Nix when it needs to build things. In my testing it's been fine, however I am not an expert in these things. Something else to keep in mind is that you should configure your services to be denied access to `/nix/persist` and instead only allow them access to individual paths in the bind mounts on `/`, just in case they do that to try and sneak an executable through. This will not stop them from making a shell script and running it with `bash ./foo.sh`, but it will make it annoying to run things like C executables, which is much more important in this case.

For this you can set the following NixOS options:

```
# hosts/meeka/noexec.nix
{ ... }:

{
  fileSystems."/".options = [ "noexec" ];
  fileSystems."/etc/nixos".options = [ "noexec" ];
  fileSystems."/srv".options = [ "noexec" ];
  fileSystems."/var/log".options = [ "noexec" ];
}
```

This will make `/nix/store` (or symlinks to files in `/nix/store`) the only binaries that are allowed to be executed. This is a rather extreme step, but it should fairly sufficiently prevent any attacker from getting very far with exploits written in languages like C (which also means that it prevents bitcoin miner bots from running).

PCI Compliance Tip

PCI Compliance requires you to have an antivirus program installed on every server. It doesn't say anything about the program *running*, but just it being installed is enough. Get one step closer to PCI compliance with this one neat trick:

```
# hosts/meeka/pci-compliance-pass.nix
{ pkgs, ... }:

{
```

```
    environment.systemPackages = with pkgs; [ clamav ];
}
```

Mara

...doesn't that defeat the spirit of the thing?

Cadey

To be honest, if you get to the end of those post and have an "all yes config" of this setup, installing an antivirus program to satisfy requirements that were primarily written for windows servers is probably one of the easiest steps you can take.

All in all, this entire setup will let you get a rather paranoid configuration that will reject everything outside of the golden path of what you told the machines to do. It will take some work to get to here (as well as being willing to experiment with a few virtual machines to test this process a few times before feeling safe enough to put this into production), but the end result should be a decently secure setup.

Obligatory warning: don't put this directly into production unless you know what you are doing, or at least can claim you know what you are doing with enough certainty to make servers difficult to debug. Have a way to "break the glass" and go back to a less noexec setup if you need to, it will save your ass.

Mara

Oh, also be sure to import all of those random `.nix` files if you want to use it in one cohesive system config. That may be a slight bit entirely essential. ^_^

Share

Facts and circumstances may have changed since publication. Please contact me before jumping to conclusions if something seems wrong or unclear.

Tags: paranoid, noexec
