BeOne stuttgart

...UND ANDERE GRÜNDE, WIESO SICH EIN BLICK AUF NIX/NIXOS LOHNT

Schlanke Container mit Nix

Dr. Stefan Schlott (BeOne Stuttgart GmbH)

ABOUT.TXT

Stefan Schlott, BeOne Stuttgart GmbH

Java-Entwickler, statische-Sprachen-Fan, Linux-Jünger

Seit jeher begeistert für Security und Privacy

Herr der Container / Übersetzer Team ⇔ Security

stefan.schlott@beone-stuttgart.de

~~@_skyr~~ @skyr@chaos.social

REPRODUCIBILITY

Was und warum

REPRODUCIBLE BUILDS

Identischer Inhalt des Builds

vs.

Binäridentisches Buildergebnis

IDENTISCHER INHALT FÜR...

Nachstellen von Bugs

Identischer Inhalt auf Entwickler- und Buildsystem

Erstellen von Patches/Hotfixes

→ „einfach Container archivieren“ schwierig

BINÄRIDENTISCHES BUILDERGEBNIS FÜR...

Vertrauensbildende Maßnahme: Angebotene Binärdownloads durch manuelles Bauen verifizieren

Caching-Mechanismen, die auf Hash der Daten basieren

→ Heute Fokus auf „identischen Inhalt“

SUPPLY CHAIN ATTACKS

Angriffe über die die Lieferkette

Quelle: Tagesschau

SHAI HULUD

Bild: Computer-Rendered Artificial Picture

XZ UTILS BACKDOOR

Die **XZ Utils Backdoor** war eine schwerwiegende Backdoor in der Open-Source-Datenkompressionssoftware XZ Utils, die im März 2024 entdeckt wurde. Die Sicherheitslücke wurde von einem Account unter dem Namen „Jia Tan“ in die Versionen 5.6.0 und 5.6.1 der liblzma-Bibliothek eingeschleust. Sie ermöglichte es Angreifern, über eine manipulierte SSH-Authentifizierung Root-Zugriff auf betroffene Linux-Systeme zu erhalten. Die Schwachstelle wurde von dem Softwareentwickler Andres Freund aufgedeckt, der eine ungewöhnlich hohe CPU-Auslastung bei SSH-Verbindungen bemerkte.[3]

Quelle: Wikipedia

MASSNAHMEN

Deaktivieren von Pre-/Post-Install-Skripten

Isolation der Build- und Ausführungsumgebung

Verzögertes Update (Release muss erst X Tage alt sein)

Versions-Pinning, um Einschleusen über Patch-Releases zu verhindern

LOCKFILES!

z.B. Python/Poetry: pyproject.toml:

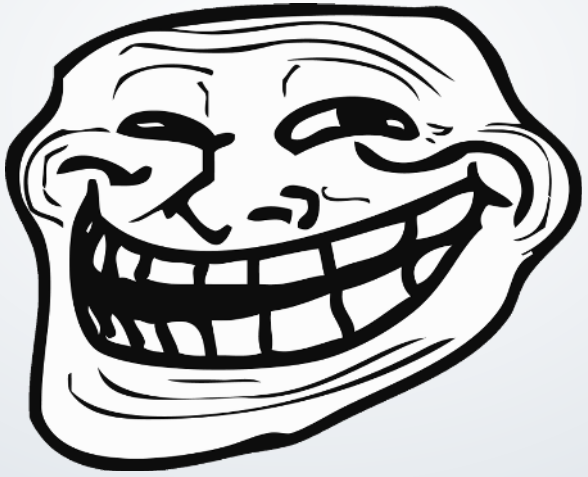
```
dependencies = [
    "click (>=8.3.0,<9.0.0)",
]
```

poetry.lock:

```
[[package]]
name = "click"
version = "8.3.3"
(...)
files = [
    {file = "click-8.3.3-py3-none-any.whl", hash = "sha256:a2bf4..."},
    {file = "click-8.3.3.tar.gz", hash = "sha256:39832..."},
]
```

...UND CONTAINER?

```
FROM debian:trixie-slim
RUN apt-get update && apt-get -y upgrade && rm -rf /var/lib/apt/lists/*
```



NIX / NIXOS

Von nix kommt Nix (tatsächlich so gemeint)

NIX?

Nix, die Sprache

Nix, der Paketmanager

NixOS, die Distribution

NIX (SPRACHE)

Pur funktional

Lazy evaluation

Pfade: First-class citizen (eigener Typ)

NIX (PAKETMANAGER)

Datenquelle: Packages-Collection(s) (typisch: git-Repos)

Packages: Compile-Anweisung + Dependencies

Binärpakete via Caches

Ablage aller Daten im „Nix Store“

Kann standalone bzw. ergänzend zum Paketmanager der Distribution verwendet werden

PAIN POINTS

Dokumentation / Lernkurve

Abweichung von LSB (Linux Standard Base),
insbes. FHS (Filesystem Hierarchy Standard)

```
#!/bin/bash
```

schlägt fehl, es muss heißen:

```
#!/usr/bin/env bash
```

FLAKES

Dependency Locking plus Commandline-Tooling

Offiziell: Unstable, wird aber überall verwendet

Mögliche Alternative (hier out of scope): npins

CONTAINERBAU MIT NIX

Das Nix-Buildsystem (mit Vorteilen und Hürden)

ERSTER CONTAINER

```
{
  inputs = {
    nixpkgs.url = "github:NixOS/nixpkgs/nixos-26.05";
  };
  outputs = { self, nixpkgs, ... }: {
    packages.x86_64-linux.nginx = let
      pkgs = nixpkgs.legacyPackages.x86_64-linux;
    in pkgs.dockerTools.buildLayeredImage {
      name = "nix-nginx";
      tag = "latest";
      contents = [ pkgs.fakeNss ];  # info for user nobody
      config = {
        User = "nobody";
        Entrypoint = [ "${pkgs.nginx}/bin/nginx" ];
      };
    };
  };
}
```

Bauen mit `nix build .#nginx`

HINWEISE

Inputs: Mehrere möglich; unterschiedliche Quellen, auch Referenz über git-Hash möglich

Erster Start erzeugt Lockfile mit Zustand der Inputs

Durch Verwendung des nginx-Pakets: Abhängigkeit klar, wird deshalb mit eingepackt

Achtung Pitfall: In git-Repo, aber Datei noch nicht eingecheckt → wird von nix ignoriert

MINIMALER INHALT!

Container enthält nix-Store-Auszug mit *exakt* den benötigten Abhängigkeiten

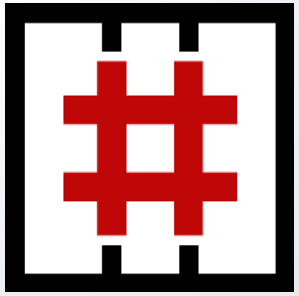
→ kein Paketmanager (und dessen Abhängigkeiten)

→ meist keine Shell

LIVING OF THE LAND ATTACKS

Nach Eindringen in System: Angriffe mit „Bordmitteln“ fortsetzen

Beispiel-Sammlungen: LOLBAS (Windows), GTFOBins (Linux)



VERGLEICH ZU DISTROLESS

Distroless: Google-Projekt, welches minimale, schlanke Container bereitstellt

Basis: Debian-Pakete

Bazel Build: Läd Debian-Pakete und entpackt diese; rekursiv wiederholen für Abhängigkeiten

→ kein Pinning auf konkreten Stand, nur Debian Release

→ potentiell Trouble wenn eigenes Programm Native Code enthält

EIGENES PROGRAMM PAKETIEREN

`mkDerivation`: Basiskonstrukt für neues Paket

Quellen: Herunterladen, gegen hinterlegte Checksumme prüfen

Deklaration Pakete für Buildvorgang (`nativeBuildInputs`) und Abhängigkeiten bei der Ausführung (`buildInputs`)

Eigentlicher Buildvorgang: Offline!
Nur Zugriff auf deklarierte Pakete!

PROBLEM OFFLINE VS DEPENDENCYAUFLÖSUNG

Typisches Buildsystem: Holt fehlende Dependencies beim Buildvorgang

Buildvorgang offline...

→ Blick in Build Helpers Documentation lohnt!

BASE-IMAGE MIT NIX

Runtime-Image mit nix
Anwendung „klassisch“ bauen
damit finalen Container erstellen

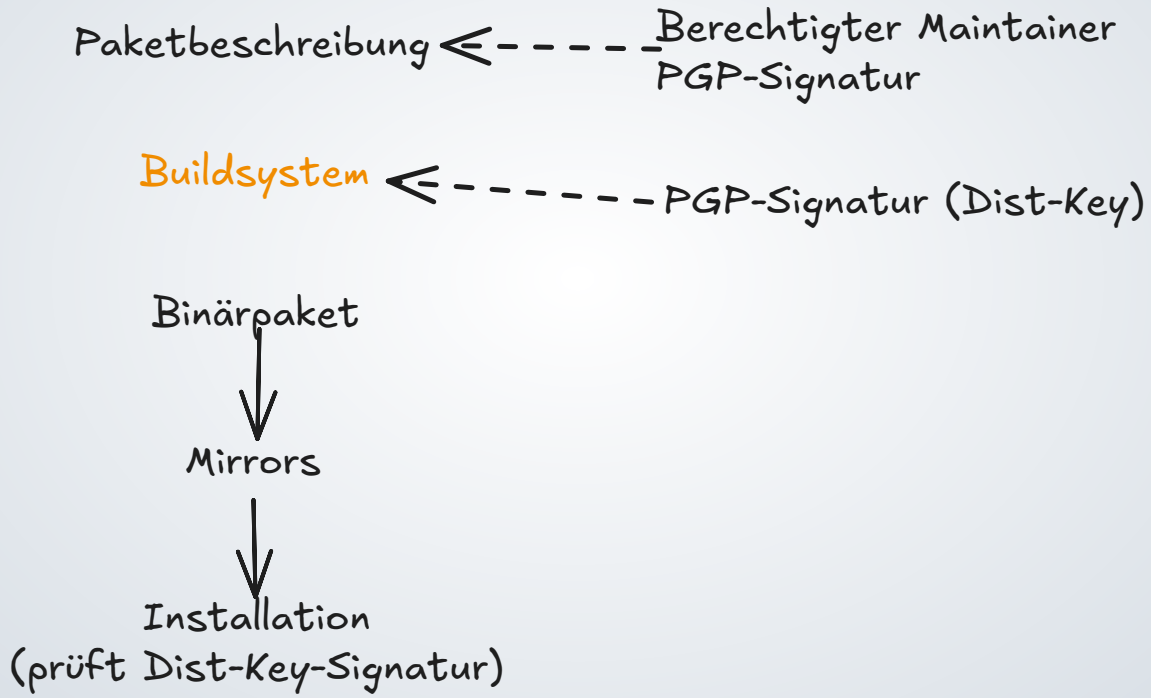
→ ähnlich Distroless, aber mit exaktem Pinning

→ imho gangbarer Kompromiss

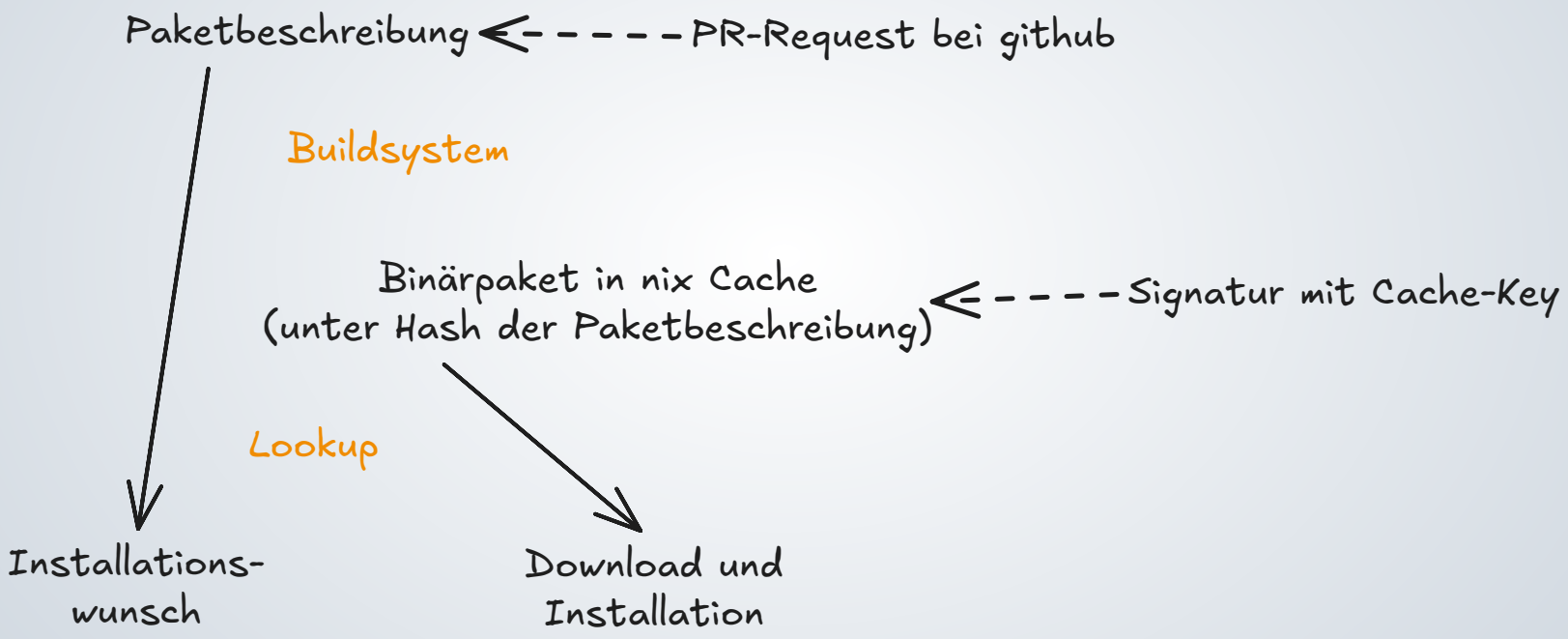
SECURITY-BETRACHTUNG

Supply-Chain-Attacks: Einfacher als bei Debian?

DEBIAN


Paketbeschreibung
Berechtigter Maintainer
PGP-Signatur
Buildsystem
PGP-Signatur (Dist-Key)
Binärpaket
Mirrors
Installation
(prüft Dist-Key-Signatur)

NIXPKGS


Paketbeschreibung
PR-Request bei github
Buildsystem
Binärpaket in nix Cache
(unter Hash der Paketbeschreibung)
Signatur mit Cache-Key
Lookup
Installations-
wunsch
Download und
Installation

EINSCHÄTZUNG

Prinzipiell bei beiden dieselben Einstiegspunkte: Maintainer/ Paketbeschreibung, Buildsystem, Cache/Repo-Key

Maintainer-Prozess bei Debian strikt reguliert (Bürgen, ...)

nixpkgs: github-zentrisch, alles steht und fällt mit dem Akzeptieren eines PRs. Prozessdoku?

WEITERE USECASES IM ENTWICKLER-UMFELD

EINHEITLICHE ENTWICKLERTOOLS

Developer Shell: Bestimmte Pakete (in gepinnter Version) verfügbar machen

devenv.sh: „Bundles“ für Sprachen / komplexe Setups (z.B. Android-Development)

devbox: Ähnlich devenv, aber JSON

→ Kombination mit direnv: ♥

...oder Container mit Tools und Nutzung mit devcontainers

TOOL-PFADE IN IDE

```
{ pkgs ? import <nixpkgs> {}}:

pkgs.mkShellNoCC {
  packages = [
    pkgs.nodejs_24
    pkgs.jdk21_headless
  ];

  shellHook = ''
    mkdir -p .local
    ln -sf "${pkgs.jdk21_headless}/lib/openjdk" .local/jdk
    ln -sf "${pkgs.nodejs_24}" .local/node
  '';
}
```

...und `.local/bin/jdk` als JDK in IDE einstellen

ISOLATION MIT MICROVM

MicroVM: Flake zum einfachen Definieren einer VM

Nix-Store wird read-only in VM gemountet: Minimaler Platzbedarf für VM-Image

Ideal für Isolation von z.B. AI-Agents

→ gibt dafür elaborierte Setups wie permafrost

FAZIT

FAZIT

Nix: Umgebung mit hoher Priorität auf Reproducibility

Lernkurve, „spröder Charme“, mitunter nicht „die“ eine Linie

Aber auch: Vielzahl der Ansätze Zeichen für Potential der Idee

Kann umfassende Lösung für identische Umgebungen sein
sowohl Containerbau als auch Entwicklungsumgebungen

NIX-REFERENZEN

nix.dev

Zero to Nix

wiki.nixos.org (*nicht* nixos.wiki)