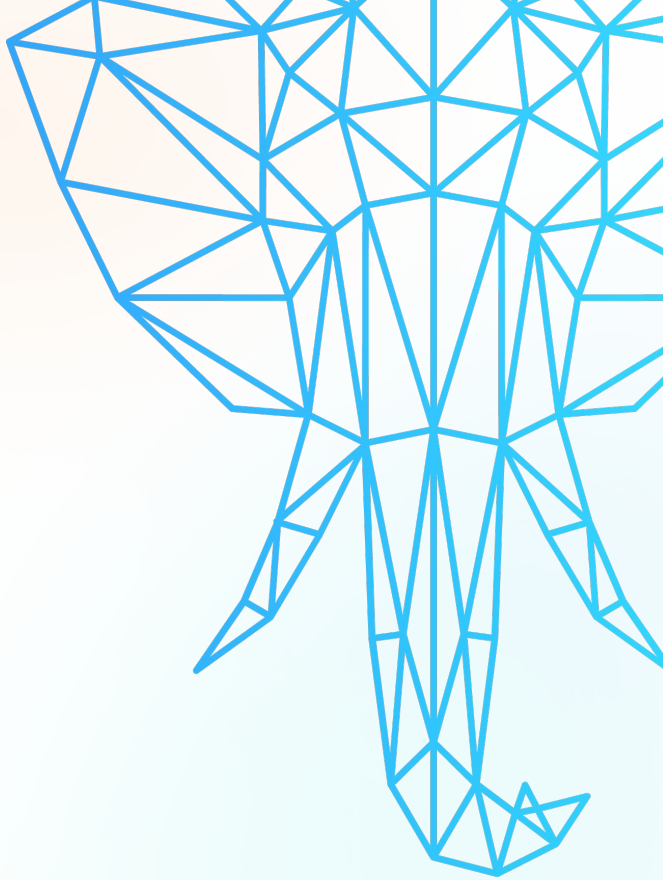
Reproducible Postgres



Javier Maestro
@jjmaestro

Álvaro Hernández
@ahachete

` whoami `

- Infrastructure Software Engineer with 20+ years of experience
- Worked at hyperscalers like Facebook and Tuenti Technologies with distributed systems, real-time data, reliability engineering, disaster recovery, and incident management.

Javier Maestro

<jjmaestro@ieee.org>

2jotas.com

` whoami `

- Founder & CEO, OnGres
- 20+ years Postgres user and DBA
- Mostly doing R&D to create new, innovative software on Postgres
- More than 140 tech talks, most about Postgres
- Founder and President of the NPO *Fundación PostgreSQL*
- AWS Data Hero

Alvaro Hernandez
‹aht@ongres.com›

aht.es

Re-thinking Postgres Distributions

Open source and supply-chain attacks

You use open source software, right?

Yes, for security reasons and to prevent vendor lock in.

Do you compile it from source?

No, I use binary packages.

Who builds those binary packages? How do you ensure they provide from the OSS software you think and no attacks are injected during the process?

Open source and supply chain attacks

CVE-2024-3094 Detail

MODIFIED

This vulnerability has been modified since it was last analyzed by the NVD. It is awaiting reanalysis which may result in further changes to the information provided.

Description

Malicious code was discovered in the upstream tarballs of xz, starting with version 5.6.0. Through a series of complex obfuscations, the liblzma build process extracts a prebuilt object file from a disguised test file existing in the source code, which is then used to modify specific functions in the liblzma code. This results in a modified liblzma library that can be used by any software linked against this library, intercepting and modifying the data interaction with this library.

https://nvd.nist.gov/vuln/detail/CVE-2024-3094


Reproducible
Builds
https://reproducible-builds.org
Reproducible builds are a set of software development practices that create an independently-verifiable path from source to binary code. (more)

Reproducible builds

If a binary is built twice* and the resulting binaries are not byte-for-byte identical, **the build is not reproducible**.

** the devil is in the details…*

Reproducible builds

Without reproducible builds:

- You have little guarantee of how the binary was built (can't reproduce).
- You can't troubleshoot on dev/test environments with the very same binary (since they may be different).
- Provisioning is much harder and caching degrades (many more binaries).

Hermetic builds

"When given the same input source code and product configuration, a hermetic build system always returns the same output by isolating the build from changes to the host system"

https://bazel.build/basics/hermeticity

Hermetic builds

Hermetic builds lead to (but don't guarantee):

- Reproducibility
- Protection from environment poisoning
- The ability to create self-contained (or static) packages

Breaking reproducibility/hermeticity

- System-dependent embeddings in the binary
  - Timestamps
  - `RPATH`
  - `GNU_BUILD_ID`
  - strings / debug info with build paths, config flags…
  - code generation (`flex` and its `#line` directive)
- Different versions of dependencies and/or tools

But Debian is reproducible, isn't it?

"*Most packages built in sid today are reproducible…*

under a fixed, predefined, build-path *and environment*"

https://wiki.debian.org/ReproducibleBuilds

Postgres source code: packaged on a “golden server”

Tarball construction

In principle this could be done anywhere, but again there's a concern about reproducibility, since the results may vary depending on installed bison, flex, docbook, etc versions. Current practice is to always do this as **pgsql** on **borka.postgresql.org**, so it can only be done by people who have a login there. In detail:

```
ssh borka.postgresql.org
sudo -u pgsql -i

mk-release-bundle commit-hash [ commit-hash ... ]
```

https://wiki.postgresql.org/wiki/Release_process

Monogres

The Postgres monorepo

Reproducible Postgres

Monogres: goal

Create the **Postgres monorepo**

A centralized repository where
Postgres and *all* of its **extensions**
are indexed, built and packaged

Monogres: an Open Source, upstream distro

- Monogres will be **Open Source** with Apache 2.0 License.
- An **upstream distribution** that other **downstream distributions** can re-use and re-package.
- Both a **binary** and (potentially) a **source** distribution

Monogres: cardinality

- 5 `major` versions
- All `minor` versions of every `major`
- 5 "option sets" (`barebones`, `minimal`, `regular`, `full`, `debug`)
- All extensions (1K+) with multiple versions
- All extensions compiled against `major.minor` versions to avoid potential ABI issues

Monogres: high cardinality

4 major-minor per year x (5y + 4y + … + 1y) x (

5 Postgres option sets `(barebones, minimal, regular, full, debug)`

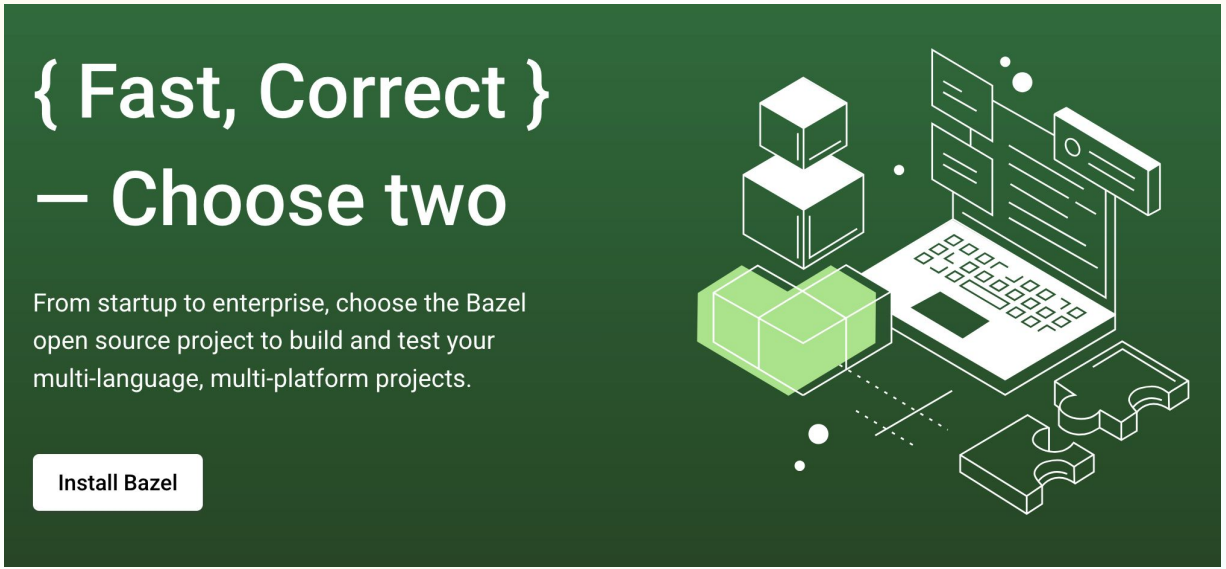
+ (1K extensions x ~10 extension versions)

) x 2 architectures `(amd64, arm64)`

= 4 x 15 x (5 + 10K) x 2 ≅ 1.2M

1M+ packages (and more!)

{Monogres, Bazel} — Choose two



A **mature** (10y), **open-source**, **build and testing tool** created by Google and the Bazel community

https://bazel.build

Bazel: remote builds

bazelbuild/remote-apis: remote execution, caching, …

(1) is becoming the de-facto **standard**

(2) with **industry support**

(3) and **no vendor lock-in**

(1) **Bazel**, Buck2, BuildStream, Pants, Please, Buildbox

(2) Aspect, BuildBuddy, Engflow, NativeLink

(3) BuildBarn, BuildBuddy, BuildFarm, BuildGrid, NativeLink

Bazel: extensible, polyglot

- It's fast, reliable, hermetic, incremental, parallelized and **extensible**
- It has a **high-level build language** with deterministic evaluation and hermetic execution (Starlark)
- **Polyglot**: supports multiple languages, platforms, and architectures (**ideal for extensions!**)

Bazel: hermeticity, sandboxing

- Bazel constructs a work directory for each target (the `execroot/`).
- It contains all input files and serves as the container for any generated outputs.
- When possible, Bazel uses an OS mechanism to constrain the action within the `execroot/` (e.g. `containers` on Linux and `sandbox-exec` on Mac)

Bazel: community, ecosystem

Third-party extensions that bring awesome functionality with little effort:

- toolchains (GCC, LLVM, Zig…)
- rules_pkg: packaging tar, zip, deb, rpm
- rules_oci: building OCI images
- BCR: Bazel Central Registry (discoverability)

Bazel: pain points

- Abstraction comes with developer complexity, especially when debugging.
- The hermeticity and reproducibility aspects still lack a simple and easy sandbox integration.
- In the end, the easy path is to initially use container images which partially defeat the purpose and complicate the reproducibility.

Monogres code tour

```
load(":cfg.bzl", "CFG")
load(":pg_build.bzl", "pg_build_all")

pg_build_all(
    name = CFG.name,
    cfg = CFG,
)
```

```
def _new(name, versions, option_sets, repo_name):
    """
    Creates a config `struct` containing build targets for multiple PostgreSQL versions.
    """
    targets = []
    default_target = None

    for version in versions:
        for option_set in option_sets:
            target = _target(name, version, option_set, repo_name)

            if version == DEFAULT_VERSION and option_set == DEFAULT_OPTION_SET:
                default_target = target

            targets.append(target)

    return struct(
        name = name,
        targets = targets,
        default = default_target,
    )

CFG = _new(
    name = "postgres",
    versions = VERSIONS,
    option_sets = OPTION_SETS,
    repo_name = REPO_NAME,
)
```

Reproducible Postgres

```
load("@pg_src//:repo.bzl", "DEFAULT_VERSION", "METADATA", "REPO_NAME", "VERSIONS")
load(":build_options.bzl", "DEFAULT_OPTION_SET", "OPTION_SETS", "build_options")

def _target(name, version, option_set, repo_name):
    """
    Creates a struct representing a Postgres build target.
    """
    if version not in VERSIONS:
        fail("Postgres version %s is not available in pg_src" % version)

    build_options_metadata = METADATA.get("build_options", {})

    return struct(
        name = "~".join((name, version, option_set)),
        version = version,
        option_set = option_set,
        pg_src = "@%s//%s" % (repo_name, version),
        build_options = build_options(version, option_set, build_options_metadata),
    )
```

Reproducible Postgres

```
151 def pg_build_all(name, cfg):
152     """
153     Defines Bazel targets for building all configured PostgreSQL versions.
154 
155     This macro instantiates `pg_build` for every version listed in the Postgres
156     config struct, and creates aliases for the default version.
157     """
158     for target in cfg.targets:
159         pg_build(
160             name = target.name,
161             pg_src = target.pg_src,
162             build_options = target.build_options,
163         )
164 
```

```
120
121 def pg_build(name, pg_src, build_options):
122     """
123     Generates a Bazel target to build PostgreSQL with the Meson build system.
124
125     This rule configures the environment and invokes the rules_foreign_cc
126     `meson` rule, using preconfigured options, toolchains, etc.
127     """
128     options, auto_features = build_options
129
130     meson(
131         name = name,
132         build_data = BUILD_DATA,
133         env = ENV | ENV_MESON,
134         lib_source = pg_src,
135         options = options | MESON_TOOL_OPTIONS,
136         out_binaries = PG_BINARIES,
137         out_data_dirs = OUT_DATA_DIRS,
138         setup_args = [
139             "--auto-features=%s" % auto_features,
140         ],
141         toolchains = TOOLCHAINS,
142         visibility = ["//visibility:public"],
143     )
144
145     native.filegroup(
146         name = "{}--logs".format(name),
147         srcs = [name],
148         output_group = "Meson_logs",
149     )
```

```
66
67 # postgres/
68 download_archives = use_repo_rule("@repo_utils//download/archives:defs.bzl", "download_archives")
69
70 download_archives(
71     name = "pg_src",
72     index = "//postgres:repo.json",
73     patches = {
74         "//postgres/patches:fix-propagate-M4-env-variable-to-flex-subprocess.patch": "*",
75     },
76 )
77
```

Reproducible Postgres

```
{
  "version": 1,
  "sources": {
    "gh": {
      "filename": "{tag}",
      "strip_prefix": "postgres-{tag}",
      "url": "https://github.com/postgres/postgres/archive/refs/tags/{filename}.tar.gz"
    }
  },
  "versions": {
    "17.0": {
      "tag": "REL_17_0",
      "sha256": "9a4b01944f9749e90e28b58e3c8556d900b68e3eef02ee509284d5312831787d"
    },
    "16.0": {
      "tag": "REL_16_0",
      "sha256": "f3ffaa5cbeefd3a6d426423e1001d01e543841946e5b13d4c8ebcad4434f2be8"
    }
  },
  "metadata": {
    "build_options": {
      "injection_points": {
        "compatible": "⩾17.0"
      }
    }
  }
}
```


Reproducible Postgres

What’s next

What's next

- Publish as open source
- Monobot: an automatic crawler that will generate repo.json
- Add more extensions
  - So far we have all contrib and some PGXS extensions
- Support multiple `glibc`
- Support multiple forks

  (Babelfish, IvorySQL, OrioleDB, OpenHalo, PgEdge, …)

github.com/monogres

Reproducible Postgres