How NixOS could have detected the XZ supply-chain attack for the benefit of all thanks to reproducible-builds

Julien Malka

▶ **To cite this version:**

Julien Malka. How NixOS could have detected the XZ supply-chain attack for the benefit of all thanks to reproducible-builds. 2025. ⟨hal-05326226⟩

HAL Id: hal-05326226

https://hal.science/hal-05326226v1

Preprint submitted on 22 Oct 2025

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

HAL Authorization

How NixOS could have detected the XZ supply-chain attack for the benefit of all thanks to reproducible-builds

Julien Malka, *Télécom Paris, Institut Polytechnique de Paris, Palaiseau, France*

Abstract—*In March 2024, a sophisticated backdoor was discovered in `xz`, a core compression library in Linux distributions, covertly inserted over three years by a malicious maintainer, Jia Tan. The attack, which enabled remote code execution via `ssh`, was only uncovered by chance when Andres Freund investigated a minor performance issue. This incident highlights the vulnerability of the open-source supply chain and the effort attackers are willing to invest in gaining trust and access. In this article, I analyze the backdoor's mechanics and explore how bitwise build reproducibility could have helped detect it.*

In March 2024, a backdoor was discovered in `xz`, a (de)-compression software that is regularly used at the core of Linux distributions to unpack source tarballs of packaged software. The backdoor had been covertly inserted by a malicious maintainer under the pseudonym of *Jia Tan* over a period of three years. This event deeply stunned the open source community as the attack was both of **massive impact** (it allowed *remote code execution* on all affected machines that had `ssh` installed) and **extremely difficult to detect**. In fact, it was only thanks to the diligence of Andres Freund — a Postgres developer working at Microsoft — that the catastrophe was avoided: while investigating a seemingly unrelated 500ms performance regression in `ssh` that he was experiencing on several *Debian unstable* machines, he was able to trace it back to the `liblzma` library, identify the backdoor and document it.

While it was already established that the open source supply chain was often the target of malicious actors, what is stunning is the amount of energy invested by *Jia Tan* to gain the trust of the maintainer of the `xz` project, acquire push access to the repository and then among other perfectly legitimate contributions insert — piece by piece — the code for a very sophisticated and obfuscated backdoor. This should be a wake up call for the OSS community. We should consider the open source supply chain a high value target for powerful threat actors, and collectively find countermeasures against such attacks.

XXXX-XXX © 2021 IEEE
Digital Object Identifier 10.1109/XXX.0000.0000000

In this article, I'll discuss the inner workings of the `xz` backdoor and how I think we could have mechanically detected it thanks to build reproducibility.

How does the attack work

The main intent of the backdoor is to allow for *remote code execution* on the target by hijacking the `ssh` program. To do that, it replaces the behavior of some of `ssh`'s functions (most importantly the `RSA_public_decrypt` one) in order to allow an attacker to execute arbitrary commands on a victim's machine when some specific RSA key is used to log in. Two main pieces are combined to install and activate the backdoor:

1) **A script to de-obfuscate and install a malicious object file as part of the `xz` build process.** Interestingly the backdoor was not comprehensively contained in the source code for `xz`. Instead, the malicious components were only contained in tarballs built and signed by the malicious maintainer *Jia Tan* and published alongside releases `5.6.0` and `5.6.1` of `xz`. This time the additional release tarball contained slight and disguised modifications to extract a malicious object file from the `.xz` files used as data for some test contained in the repository.
2) **A procedure to hook the `RSA_public_decrypt` function.** The backdoor

uses the *ifunc* mechanism of `glibc` to modify the address of the `RSA_public_function` when `ssh` is loaded, in case `ssh` links against `liblzma` through `libsystemd`.

1. A script to de-obfuscate and install a malicious object file as part of the `xz` build process

As explained above, the malicious object file is stored directly in the `xz` git repository, hidden in some test files. The project being a decompression software, test cases include `.xz` files to be decompressed, making it possible to hide some machine code into fake test files; The backdoor is not active in the code contained in the git repository, it is only included by building `xz` from the tarball released by the project, which has a few differences with the actual contents of the repository, most importantly in the `m4/build-to-host.m4` file.

The changes may look benign to the naive eyes and well commented, they are actually hiding a chain of commands that decrypts/deobfuscates several fake `.xz` test files to ultimately produce two files:

1) a shell script that is run during the build of `xz` ;
2) a malicious binary object file.

There is an excellent analysis from Russ Cox [1] that explains in detail how these two malicious resources are produced during the build process, and I advise any interested reader to find all relevant details there.

The shell script run during the build has two main purposes:

1) Verifying that the conditions to execute the backdoor are met on the builder (the backdoor targets specific Linux distributions, needs specific features of the glibc activated, needs ssh installed, etc) ;
2) Modifying the (legitimate) `liblzma_la-crc64_fast.o` to use the `_get_cpuid` symbol defined in the backdoor object file.

2. A procedure to hook the `RSA_public_decrypt` function

So how does a backdoor in the `xz` executable have any effect on `ssh`? To understand that, we have to take a little detour in the realm of dynamic loaders and dynamically linked programs. Whenever a program depends on a library, there are two ways that library can be linked into the final executable:

- statically, in that case the library is embedded into the final executable, hence increasing its size ;
- dynamically, in which case it is the role of the dynamic loader (`ld-linux.so` in Linux) to find that shared library when the program starts and load it in memory.

When a program is compiled using dynamic linking, the addresses of the symbols belonging to dynamically linked libraries cannot be provided at compilation time: their position in memory is not know ahead of time! Instead, a reference to the *Global Offset Table* (or *GOT*) is inserted. When the program is started, the actual addresses are filled in the GOT by the dynamic linker.

The `xz` backdoor uses a functionality of the *glibc* called *ifunc* to force execution of code during dynamic loading time: *ifunc* is designed to allow selection between several implementations of the same function at dynamic loading time.

Listing 1. Minimal ifunc example (C)

```
#include <stdio.h>

// Declaration of ifunc resolver function
int (*resolve_add(void))(int, int);

// First version of the add function
int add_v1(int a, int b) {
printf("Using add_v1\n");
return a + b;
}

// Second version of the add function
int add_v2(int a, int b) {
printf("Using add_v2\n");
return a + b;
}

// Resolver function that chooses the correct
     version of the function
int (*resolve_add(void))(int, int) {
// You can implement any runtime check here.
// In that case we check if the system is 64bit
if (sizeof(void*) == 8) {
return add_v2;
} else {
return add_v1;
}
}

// Define the ifunc attribute for the add function
int add(int a, int b)
     __attribute__((ifunc("resolve_add")));

int main() {
int result = add(10, 20);
printf("Result: %d\n", result);
return 0;
}
```

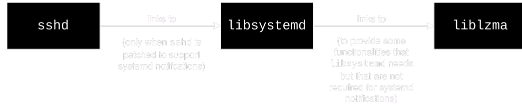

FIGURE 1. Dependency chain between `sshd` and `liblzma`.

In the above example, the *ifunc* attribute surrounding the `add` function indicates that the version that will be executed will be determined at dynamic loading time by running the `resolve_add` function. In that case, the `resolve_add` function returns `add_v1` or `add_v2` depending if the running system is a 64 bit system or not – and as such is completely harmless – but this technique is used by the `xz` backdoor to run some malicious code at dynamic loading time.

But dynamic loading of which program? Well, of `ssh`! In some Linux distributions (Debian and Fedora for example), `ssh` is patched to support `systemd` notifications and for this purpose, links with `libsystemd`, that in turn links with `liblzma`. In those distribution `sshd` hence has a transitive dependency on `liblzma`.

This is how the backdoor works: whenever `sshd` is executed, the dynamic loader loads `libsystemd` and then `liblzma`. With the backdoor installed, and leveraging the *ifunc* functionality as explained above, the backdoor is able to run arbitrary code when *liblzma* is being loaded. Indeed, as you remember from the previous section, the backdoor script modifies one of the legitimate `xz` object files: it actually modifies the resolver of one of the functions that uses *ifunc* to call its own malicious `_get_cpuid` symbol. When called, this function meddles with the GOT (that is not yet read-only at this time of execution) to modify the address of the `RSA_public_decrypt` function, replacing it by a malicious one! That's it, at this point `sshd` uses the malicious `RSA_public_decrypt` function that gives RCE privileges to the attacker.

Once again, there exist more precise reports on exactly how the hooking happen [2].

Avoiding the `xz` catastrophe in the future

What should our takeaways be from this near-miss and what should we do to minimize the risks of such an attack happening again in the future? There is a lot to be said about the social issues at play here and how we can build better resilience in the OSS ecosystem against malicious entities taking over really fundamental OSS projects, but in this piece I'll only address the technical aspects of the question.

People are often convinced that OSS is more trustworthy than closed-source software because the code can be audited by practitioners and security professionals in order to detect vulnerabilities or backdoors. In this instance, this procedure has been made difficult by the fact that part of the code activating the backdoor was not included in the sources available within the git repository but was instead present in the maintainer-provided tarball. While this was used to hide the backdoor out of sight of most investigating eyes, this is also an opportunity for us to improve our software supply chain security processes.

1. Building software from trusted sources

One immediate observation that we can make in reaction to this supply chain incident is that it was only effective because a lot of distributions were using the maintainer provided tarball to build `xz` instead of the raw source code supplied by the git forge (in this case, GitHub). This reliance on release tarballs has plenty of historical and practical reasons:

- the tarball workflow predates the existence of `git` and was used in the earliest Linux distributions;
- tarballs are self-contained archives that encapsulate the exact state of the source code intended for release while git repositories can be altered, creating the need for a snapshot of the code;
- tarballs can contain intermediary artifacts (for example manpages) used to lighten the build process, or configure scripts to target specific hardware, etc;
- tarballs allow the source code to be compressed which is useful for space efficiency.

This being said, these reasons do not weigh enough in my opinion to justify the security risks they create. In all places where it is technically feasible, we should build software from sources authenticated by the most trustworthy party. For example, if a project is developed on GitHub, an archive is automatically generated by GitHub for each release. The risk of a compromise of that release archive is far lower than the risk of a malicious maintainer distributing unfaithful tarballs, as it would require compromising the GitHub infrastructure (and at this point the problem is much more serious). This reasoning can be extended in all cases where the development is happening on a platform operated by a trusted third party like Codeberg/SourceHut/Gitlab, etc.

2. When building from source is possible...

NixOS is a distribution built on the functional package management model, that is to say every package is encoded as an expression written in Nix, a functional programming language. A Nix expression for a software project is usually a function mapping all the project dependencies to a "build recipe" that can be later executed to build the package. I am a NixOS developer and I was surprised when the backdoor was revealed to see that the malicious version of `xz` had ended up being distributed to our users. While there is no policy about this, there is a culture among NixOS maintainers of using the source archive automatically generated by GitHub (that are simply snapshots of the source code) when available through the `fetchFromGitHub` function. In the simplified example of the `xz` package below, you can see that the sources for the package are actually extracted from the manually uploaded *malicious* maintainer provided tarball through another source fetcher: `fetchurl`.

Listing 2. Simplified Nix expression for the XZ package

```
{ lib, stdenv, fetchurl
, enableStatic ? stdenv.hostPlatform.isStatic
}:


stdenv.mkDerivation rec {
pname = "xz";
version = "5.6.0";


src = fetchurl {
url = "github.com/xz/releases/v5.6.0.tar.xz";
hash = "sha256-...=";
};
...
}
```

To understand why, we must first talk about the bootstrap of `nixpkgs`. The concept of a bootstrap is the idea that one could rebuild all of the packages in `nixpkgs` from a small set of seed binaries. This is an important security property because it means that there are no other external tools that one must trust in order to trust the toolchain that is used to build the software distribution. What we call the "bootstrap" in the context of a software distribution like `nixpkgs`, is all the steps needed to build the basic compilation environment to be used by other packages, called `stdenv` in nixpkgs. Building `stdenv` is not an easy task; how does one build `gcc` when one doesn't even have a C compiler? The answer is that you start from a very small binary that does nothing fancy but is enough to build `hex`, a minimalist assembler, which in turn can build a more complex assembler, and this until we are able to build more complex software and finally a modern C compiler. The bootstraping story of Nix/Guix is an incredibly interesting topic, that I will not cover extensively here, but I strongly advise reading blog posts from the Guix community, that are on the bleeding edge (they have introduced a 357-byte bootstrap [3] that is being adapted for nixpkgs).

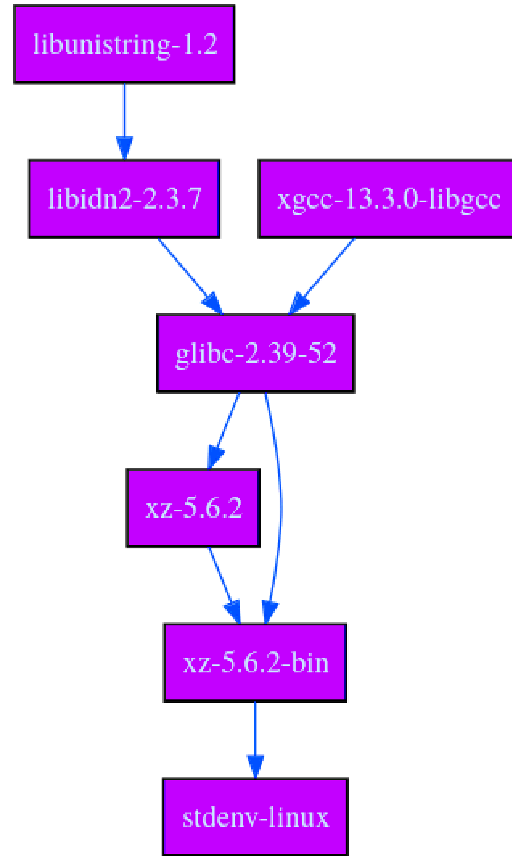

FIGURE 2. Runtime dependency graph showing `stdenv` depending on `xz`.

What does all that has to do with `xz` though? Well, `xz` is included in the nixpkgs bootstrap (see Figure 2).

We can see that `stdenv` depends at runtime on `xz`, so it is indeed built during the bootstrap stage. To understand a bit more why this is the case, Figure 3 is a graph of the software in `stdenv` that depends on `xz` at buildtime.

We can see that several packages depend on `xz`. Let's take `coreutils` for example and try to understand why it depends on `xz` by reading its derivation file, which is the intermediary representation of the build process obtained by evaluating the Nix expression for `coreutils`:

The `inputDrvs` field in Figure 4 corresponds to all the other packages or expressions that the

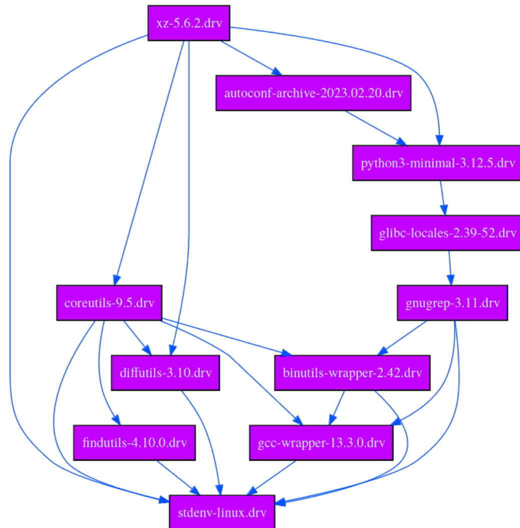

FIGURE 3. Build-time graph of packages in `stdenv` that depend on `xz`.

`coreutils` build process depends on. We see that in particular it depends on two components:

- `...12mrim1av6-xz-5.6.2.drv`, which is `xz` itself;
- `...4vzg-coreutils-9.5.tar.xz.drv` which is a source archive for `coreutils`! As it is a `.xz` archive, we need `xz` to unpack it and that is where the dependency comes from!

The same reasoning applies to the other three direct dependencies that we could see in the graph earlier.

`xz` being built as part of the bootstrap means it doesn't have access to all the facilities normal packages in nixpkgs can rely on. In particular it can only access packages that are built /before/ in bootstrap. For example, to build `xz` from sources, we need `autoconf` to generate the configure script. But `autoconf` has a dependency on `xz`! Using the maintainer tarball allows us to break this dependency cycle.

In conclusion, at the point in the `nixpkgs` graph where the `xz` package is built, the GitHub source archive cannot be used and we have to rely on the maintainer provided tarball, and hence, trust it. That does not mean that further verification cannot be implemented in `nixpkgs`, though...

Building trust into untrusted tarballs

To recap, the main reason that made NixOS vulnerable to the xz attack is that it is built as part of the bootstrap phase, at a point where we rely on maintainer-provided tarballs instead of the ones generated by GitHub. This incident shows that we should have specific protections in place, to ensure software built as part of our bootstrap is trustworthy.

By comparing sources

One idea that comes to mind is that it should be easy, as a distribution, to verify that the sources tarballs we are using are indeed identical to the GitHub ones. While this seem like a natural idea, it doesn't really work in practice: it's not that rare that the maintainer provided tarball differs from the sources, and it's often nothing to worry about.

> *Can I just say that I have created curl releases "the xz way" since the 90s: I generate the release tarballs on my machine. It makes the tarball have (generated) files included that are not present in git. It's a feature. But it also makes it harder for observers to figure out if the additional files are fine or not.*

As Daniel Stenberg (the maintainer of `curl`) explains, the release tarball being different than the source is a *feature*: it allows the maintainer to include intermediary artifacts like manpages or configure scripts for example (this is especially useful for distributions that want to get rid of the dependency on `autoconf` to build the program). Of course when we care about software supply chain security, this flexibility that project maintainers have in the way they provide the release assets is actually a liability because it forces us to trust them to do it honestly.

Leveraging bitwise reproducibility

Reproducible builds is a property of a software project that is verified if building it twice in the same conditions yields the exact same (bitwise identical) artifacts. Build reproducibility is not something easy to obtain, as there are all kinds of nondeterminisms that can happen in build processes, and making as many packages as possible reproducible is the purpose of the Reproducible-Builds group. It is also a property recognized as instrumental to increase the trust in the distribution of binary artifacts [4].

There are several ways bitwise reproducibility could be used to build up trust in untrusted maintainer provided tarballs:

FIGURE 4. Recipe for the `coreutils` package in the bootstrap of nixpkgs


```
{
  "/nix/store/57hlz5fnvfgljivf7p18fmcl1yp6d29z-coreutils-9.5.drv": {
    "args": [
      "-e",
      "/nix/store/v6x3cs394jgqfbi0a42pam708flxaphh-default-builder.sh"
    ],
    "builder": "/nix/store/razasrvdg7ckplfmvdxv4ia3wbayr94s-bootstrap-tools/bin/bash",

      ...

    "inputDrvs": {

      ...

      "/nix/store/c0wk92pcxbxi7579xws6bj12mrim1av6-xz-5.6.2.drv": {
        "dynamicOutputs": {},
        "outputs": [
          "bin"
        ]
      },
      "/nix/store/xv4333kfggq3zn065a3pwrj7ddbs4vzg-coreutils-9.5.tar.xz.drv": {
        "dynamicOutputs": {},
        "outputs": [
          "out"
        ]
      }
    },

    ...

    "system": "x86_64-linux"
  }
}
```


Reproducibly building the tarball

A first approach that has been adopted by the postgresql project [6] is to make the tarball generation process reproducible. This allows any user (or a linux distribution) to independently verify that the maintainer provided tarball was honestly generated from the original source code.

With this method, you can keep some advantages of building from tarballs (including the tarball containing some intermediary build artifacts like manpages or configure scripts). However, the drawback of this approach for software supply chain security is that it has to be implemented by upstream project maintainers. This means that adoption of this kind of security feature will probably be slow in the FOSS community, and while it is a good practice to make *everything* reproducible, including the tarball generation process, this is not the most effective way to increase software supply chain security *today*.

Checking for build convergence between various starting assets

a

Assuming `xz` is bitwise reproducible (and that is indeed the case), and that the maintainer provided tarball doesn't contain any modification that impacts the build process, building it from the GitHub tarball or from the maintainer provided tarball *should* produce the same artifacts, right? Based on this idea, my proposal is to build `xz` a second time *after* the bootstrap, this time using the GitHub tarball (which is only possible after the bootstrap). If both builds differ we can suspect that there a suspicion of a supply chain compromise.

Let's see how this could be implemented:

First, we rewrite the `xz` package, this time using the `fetchFromGitHub` function. I create a `after-boostrap.nix` file alongside the original `xz` expression in the `pkgs/tools/compression/xz` directory of nixpkgs:

Listing 3. Nix expression for the `xz-after-bootstrap` package.

```
{
lib,
stdenv,
fetchurl,
enableStatic ? false,
writeScript,
fetchFromGitHub,
testers,
gettext,
autoconf,
libtool,
automake,
perl538Packages,
```

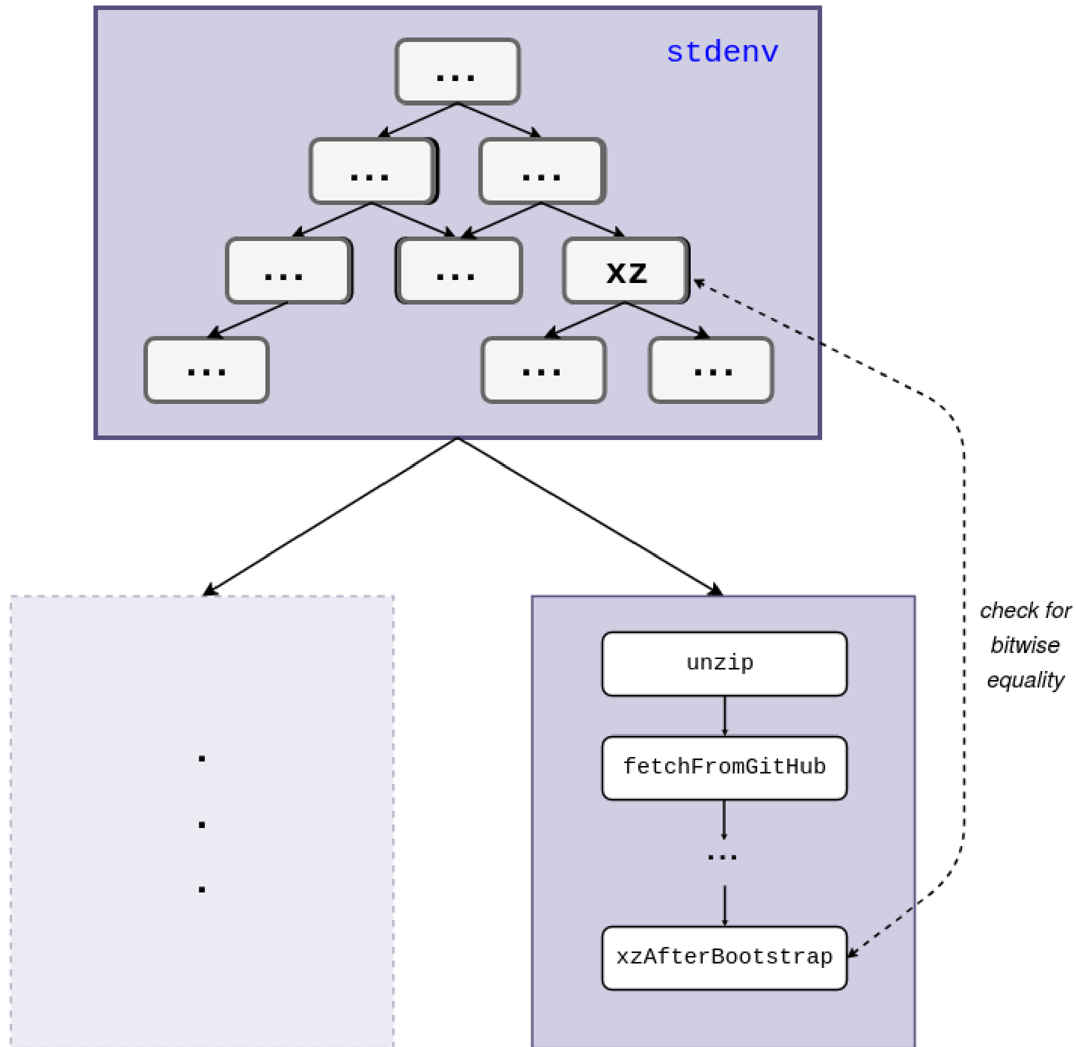

FIGURE 5. Summary of the method proposed to detect vulnerable `xz` source tarballs.

```
    doxygen,
    xz,
  }:

  stdenv.mkDerivation (finalAttrs: {
    pname = "xz";
    version = "5.6.1";

    src = fetchFromGitHub {
      owner = "tukaani-project";
      repo = "xz";
      rev = "v${finalAttrs.version}";
      hash =
     "sha256-alrSXZ0KWVlti6crmdxf/qMdrvZsY5yigcV9j6GIZ6c=";
    };

    strictDeps = true;
    configureFlags = lib.optional enableStatic
     "--disable-shared";
    enableParallelBuilding = true;
    doCheck = true;
    nativeBuildInputs = [
      gettext
      autoconf
      libtool
      automake
      perl538Packages.Po4a
      doxygen
      perl
    ];

    preConfigure = ''
      ./autogen.sh
    '';
  })
```

I removed details here to focus on the most important: the Nix expression is very similar to the actual derivation for `xz`, the only difference (apart from the method to fetch the source) is that we need to use

autoconf to generate configure scripts. When using the maintainer provided tarball these are already pre-generated for us (as Daniel Stenberg was explaining in the toot above) – which is very handy particularly when you are building xz in the bootstrap phase of a distribution and you don't want a dependency on autoconf / automake – but in this instance we have to do it ourselves.

Now that we can build xz from the code archive provided by GitHub, we have to write Nix code to compare both outputs. For that purpose, we register a new phase called compareArtifacts, that runs at the very end of the build process. To make my point, I'll first only compare the liblzma.so file (the one that was modified by the backdoor), but we could easily generalize this phase to all binaries and libraries outputs:

Listing 4. Example implementation of the new compareArtifacts phase.

```
postPhases = [ "compareArtifacts" ];

compareArtifacts = ''
  diff $out/lib/liblzma.so
   ${xz.out}/lib/liblzma.so
  '';
```

Now, building xz-after-bootstrap on master completes successfully, but running on version 5.6.1, outputs the logs on Figure 6, confirming that the proposed method could have helped identify the backdoor.

Addendum 1: Evaluation: reproducibility of stdenv over time

As discussed above, the method I propose assumes the packages we want to build trust in are *bitwise reproducible*. In order to help validate the approach, let's verify that the packages belonging to the stdenv runtime are indeed reproducible. To do that, I have (as part of a wider research project [5]) sampled 17 nixpkgs-unstable revisions from 2017 to 2023 and rebuilt every *non-fixed-output-derivation* (FOD) composing stdenv from these revisions using the nix-build -check command to check for bitwise reproducibility. Here are my findings:

- In every revision xz was bitwise reproducible ;
- In 12 of the 17 revisions there was either one or two packages that were buildable but not reproducible, but those packages are consistent over time: for example gcc has consistently been non reproducible from 2017 to 2021 and bash until 2019.

These findings, while showing that this method cannot be applied to *every* package in stdenv, are encouraging: even if some packages are not bitwise reproducible, they are consistently so, which means that it should be possible to selectively activate it on packages that exhibit good reproducibility in the long term.

Addendum 2: Limitations: the trusting trust issue

The trusting trust issue is a famous thought experiment initiated by Ken Thomson during his Turing award acceptance lecture. The idea is the following: assume there is a backdoor in compilers we use to build our software such that the compiler propagates the backdoor to all new version of itself that it builds, but behaves normally for any other build until some point in time where it backdoors all executables it produces. Moderns compilers often need a previous version of themselves to be compiled so there must be an initial executable that we have to trust to build our software, making this kind of sophisticated attack *theoretically* possible and completely undetectable. Similarly, the method I am proposing here requires to make the assumption that the untrusted xz (the one built during the bootstrap phase) can't indirectly corrupt the build of xz-after-bootstrap to make it look like the produced artifacts are identical. Again, such an attack would probably be extremely complex to craft so the assumption here seems sane.

REFERENCES

1. Cox, The xz attack shell script, https://research.swtch.com/xz-script
2. XZ backdoor story – Initial analysis, https://securelist.com/xz-backdoor-story-part-1/112354/
3. Nieuwenhuizen, Courtes, The Full-Source Bootstrap: Building from source all the way down, https://guix.gnu.org/en/blog/2023/the-full-source-bootstrap-building-from-source-all-the-way-down/
4. Lamb, Zacchiroli, Reproducible Builds: Increasing the Integrity of Software Supply Chains, IEEE Software
5. Eisentraut, The new PostgreSQL 17 make dist, https://peter.eisentraut.org/blog/2024/08/13/the-new-postgresql-17-make-dist
6. Malka, Zacchiroli, Zimmermann, Does Functional Package Management Enable Reproducible Builds at Scale? Yes. MSR'25

Julien Malka is a PhD student at Télécom Paris, working on fucntional package management, reproducibility and software supply chain security.

FIGURE 6. compareBins output showing differing `liblzma.so` paths.

```
$ nix-build -A xz-after-bootstrap
/nix/store/57p62d3m98s2bgma5hcz12b4vv6nhijn-xz-5.6.1
...
...
Running phase: compareBins
Binary files /nix/store/cxz8iq3hx65krsyraill6figp03dk54n-xz-5.6.1/lib/liblzma.so \
and /nix/store/4qp2khyb22hg6a3jiy4hqmasjinfkp2g-xz-5.6.1/lib/liblzma.so differ
```