Trusting-Trust Attack against an Entire Linux Distribution through Binary Manipulation

Julien Malka
julien.malka@telecom-paris.fr
LTCI, Télécom Paris,
Institut Polytechnique de Paris
Palaiseau, France

Aman Sharma
amansha@kth.se
KTH Royal Institute of Technology
Stockholm, Sweden

Martin Monperrus
monperrus@kth.se
KTH Royal Institute of Technology
Stockholm, Sweden

Stefano Zacchiroli
stefano.zacchiroli@telecom-paris.fr
LTCI, Télécom Paris,
Institut Polytechnique de Paris
Palaiseau, France

Théo Zimmermann
theo.zimmermann@telecom-paris.fr
LTCI, Télécom Paris,
Institut Polytechnique de Paris
Palaiseau, France

Abstract

Ken Thompson's trusting-trust attack, in which a compromised compiler backdoors the programs it builds and reproduces the backdoor in subsequent rebuilds of itself, is often regarded as a threat specific to compilers. We show that it is not. We construct a complete trusting-trust attack around GNU `strip`, an ordinary build utility that neither inspects nor generates source code, using only manipulations of finished ELF files. In one bootstrap execution of the NixOS Linux distribution, a single tampered `strip` in the binary seed implants its payload into each later `strip` rebuilt from unmodified source. The infected `strip` in the final standard environment can then implant the payload into binaries of downstream packages, even though that environment has no runtime reference to the bootstrap seed. On a real nixpkgs revision, the attack builds a complete graphical installer without failures and backdoors almost every one of its binaries, enabling arbitrary malicious behavior of the subverted packages.

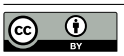
CCS Concepts

• **Security and privacy** → **Software security engineering**; • **Software and its engineering** → *Software creation and management*.

Keywords

trusting trust, software supply chain, bootstrapping, Linux distributions, binary rewriting

ACM Reference Format:
Julien Malka, Aman Sharma, Martin Monperrus, Stefano Zacchiroli, and Théo Zimmermann. 2026. Trusting-Trust Attack against an Entire Linux Distribution through Binary Manipulation. In *Conference on Software Supply Chain Offensive Research and Ecosystem Defenses (SCORED '26), October 06, 2026, Prague, Czech Republic*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3848003.3848012



SCORED '26, Prague, Czech Republic

ACM ISBN 979-8-4007-3009-2/2026/10
https://doi.org/10.1145/3848003.3848012

1 Introduction

"Given enough eyeballs, all bugs are shallow." Eric S. Raymond [20] popularizes this statement as Linus's law in his essay on open source software development. It means that making source code publicly available for review, including by security experts, makes free and open source software less likely to contain bugs, security vulnerabilities, or backdoors. Yet software must first be compiled before it can run, which limits the trust that source availability alone provides. As Ken Thompson showed in his Turing Award lecture [23], some compromises remain invisible to source code review. In Thompson's construction, a malicious compiler recognizes both a security-sensitive program and the source of the compiler itself. It inserts a backdoor into the former and reproduces this logic in the latter. The malicious source can then be removed: recompiling unmodified compiler source with the compromised binary propagates the compromise.

Thompson's attack, dubbed "trusting trust", was more than a thought experiment because he implemented a prototype. However, few documented or studied implementations exist, and the practical persistence of such a backdoor remains unclear. One difficulty is that the malicious compiler must recognize its own source code to produce an altered binary. Significant changes to that source could therefore stop the backdoor from propagating without exposing it.

Although the trusting-trust attack is often regarded as a threat specific to compilers, we demonstrate how its underlying mechanism extends to other build utilities. We instantiate the attack in GNU `strip`, a post-build utility from GNU binutils that removes information from object files and executables, in the context of the NixOS Linux distribution. Because `strip` processes a large fraction of the executables produced by a distribution, compromising it provides an opportunity both to propagate the compromise and to affect downstream software.

We demonstrate an end-to-end attack in which a malicious `strip` binary placed in the initial NixOS bootstrap seed causes the compromise to reproduce in later `strip` binaries rebuilt from unmodified source, and subsequently modifies other executables processed during their build. Unlike Thompson's original construction, our attack operates entirely through ELF-level transformations and neither inspects nor modifies source code. It is therefore insensitive to source-level changes that would prevent a malicious compiler

from recognizing its own source. To our knowledge, this is the first documented end-to-end implementation of a self-propagating trusting-trust attack carried by a non-compiling post-build utility through the bootstrap of a production Linux distribution.

This paper makes the following contributions:

- We show that the trusting-trust attack extends beyond compilers by designing a self-propagating compromise of GNU `strip` based entirely on ELF-level transformations that neither inspect nor modify source code (Section 4).
- We demonstrate that the attack propagates end to end through one NixOS bootstrap execution and persists in the final standard environment even though its output has no runtime reference to the malicious seed (Sections 3 and 4).
- We evaluate the attack by rebuilding a large and representative package set from nixpkgs, measuring its effect on buildability, its reach across downstream executables, and its impact on their runtime behavior (Section 5).

We release the complete implementation and experimental artifacts in our replication package [12].

2 Background

2.1 Bootstrapping Linux Distributions

A *Linux distribution* is a curated collection of software (the kernel, a compiler toolchain and C library, system services, and end-user applications) assembled and shipped together as a complete operating system, with a package manager that installs and updates the individual *packages*. Debian, Fedora, Arch Linux, and the Nix-based NixOS are such examples. A distribution contains *recipes* for building packages, each specifying the upstream source and the exact steps needed to turn it into an installable package.

A Linux distribution needs to be bootstrapped: producing packages requires a compiler, a linker, and a C library. Construction of the entire distribution must therefore begin from a small set of pre-existing binaries called the *bootstrap seed*. *Bootstrapping* is the process that starts from this seed and builds a build toolchain. A distribution is self-hosted when it depends only on binary artifacts it has built itself.

2.2 The Nixpkgs Bootstrap

Nix is a functional package manager, a model in which packages are built from recipes (also called *derivations*) written in a functional programming language. Derivations contain a precise definition of the build environment needed to build the packages and undeclared dependencies are excluded from builds by a sandboxing mechanism [7].

The package collection *nixpkgs* gathers the package recipes from which the Linux distribution *NixOS* is built. Nixpkgs packages are built from a standard environment (or `stdenv`), containing a C toolchain and associated build helpers such as binutils, a shell, and basic Unix tools [18]. The standard environment also provides a *generic builder* that runs each package through a fixed sequence of standard phases [18]. These phases unpack, configure, build, and install the sources. A default post-install phase called *fixup* then normalizes the installed outputs by invoking GNU `strip` on their binaries and adjusting their runtime paths. Constructing `stdenv` poses the familiar chicken-and-egg bootstrap problem described in Section 2.1: those tools are needed to build themselves. To solve it, Nixpkgs relies on a small set of pre-built binaries called the *bootstrap seed* to build the standard environment from source.

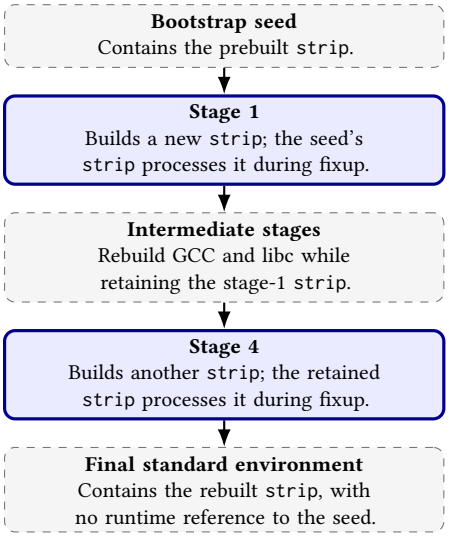


Figure 1: Relevant portion of the evaluated nixpkgs bootstrap. Arrows show the order of the stages. Heavy-bordered boxes mark stages that build `strip` from source as part of binutils; dashed boxes contain or retain an existing copy. Each newly built `strip` is processed by a previous `strip` during fixup.

The seed bundles roughly twenty prebuilt programs, including a C compiler, binutils, the C library, and core utilities such as `bash`, `coreutils`, `tar`, `gzip`, and `patch` [25]. The bootstrap rebuilds the toolchain and basic utilities from source in several stages, each using tools produced by earlier stages. The final `stdenv` has no runtime reference to the bootstrap seed, even though tools derived from the seed were used to build it. Binutils, and with it `strip`, is built more than once to remove these runtime references. The first source-built `strip` still uses the seed's compiler, while a later bootstrap stage builds `strip` with the rebuilt toolchain. Figure 1 shows the relevant portion of this sequence.

2.3 Trusting Trust Attack

The original trusting-trust attack has been described by Thompson [23]. In this attack, a compromised compiler binary builds a new compiler from unmodified source. That source contains no malicious logic. During compilation, however, the compromised binary injects malicious logic into the new binary. Because the injection happens during compilation rather than in the source, inspecting the source code cannot detect it. The injected logic includes a copy of itself. When the newly built compiler later compiles another copy of the compiler, it performs the same injection. The attack propagates this way indefinitely, regardless of how many times the source is audited.

2.4 Linux Internals

2.4.1 The `strip` *utility.* GNU `strip`, part of binutils, removes symbols and other information from object files. `strip` is the tool the generic builder runs in the fixup phase [18]. Thus `strip` is routinely granted write access to executable package outputs.

2.4.2 ELF Format. The Executable and Linkable Format (ELF) is the standard binary format for Linux executables, shared libraries, and object files. It is consumed both by the operating system to load and execute a program and by build tools to transform programs.

Figure 2 presents a running example of a simplified ELF file and how it is mapped in memory by the operating system loader. An ELF file begins with a header that identifies the target architecture and file type and locates two different tables: the *program header table* (located by the `e_phoff` entry) and the *section header table* (located by the `e_shoff` entry). The header also contains `e_entry`, the virtual address at which execution begins [22].

The program header table tells the operating-system loader how to load the executable. Each entry has a type that determines how the loader interprets it. A `PT_LOAD` entry describes a loadable *segment*, specifying its file offset, virtual address, size, alignment, and permissions. Other entry types provide auxiliary information. In particular, `PT_INTERP` specifies the dynamic loader, and `PT_NOTE` identifies notes such as ABI or build metadata. Once the loadable segments are mapped, the kernel transfers control to the `e_entry` address. It does not need the section table to start the program.

Section headers provide a complementary, tool-oriented view of many of the same bytes. Named sections such as `.text`, `.data`, symbol tables, and note sections describe logical units used by linkers, debuggers, and binary utilities. A segment may contain several sections, and not all sections need to be mapped at runtime. GNU `strip` reasons about this representation when deciding which information to retain or discard.

Program headers and section headers serve different consumers, but they must remain coherent with one another.

2.4.3 ELF transformations. The attack described in this paper does not act on the source code of the packages it targets. It manipulates the binary files directly. We call these binary-level edits *ELF transformations* [3].

We use three transformations defined below that add regions and adjust metadata. They do not rewrite existing program code or data.

- *Append* adds new bytes at the end of the file;
- *Repurpose* hijacks one spare metadata slot, a `PT_NOTE` program header the loader does not need for execution, to use as an executable segment instead;
- *Redirect* changes a few ELF-header fields so that the loader and the binary tools notice what was appended.

These transformations preserve the existing program code and data, although repurpose and redirect modify ELF metadata. Figure 3 shows each transformation as a concrete before/after edit to the file.

3 Threat Model

We assume the attacker wants to attack a Linux distribution by targeting its binary bootstrap seed. The attacker's capabilities are as follows. The attacker can replace one executable in the bootstrap seed of the distribution. The attacker cannot modify any distribution recipe or package sources.

This capability models compromise before the seed is included in the distribution: for example, by compromising the system that produces or publishes bootstrap archives, or by making an authorized update to the pinned seed, or with social engineering of the process by which maintainers decide and merge a new seed.

4 The Bootstrapping Attack

4.1 Attack Objective

The objective of the attack is to compromise every downstream user of the distribution, and to keep doing so indefinitely. We design and implement the attack through the `strip` binary, because it touches all binaries in a distribution. Unlike a compiler such as `gcc`, which affects only the programs written in the languages it compiles, `strip` operates on finished ELF binaries directly and therefore reaches the outputs of every language and toolchain.

So, per the trusting trust attack model, a single tampered seed `strip` can both (1) infect any application target and (2) reproduce the compromise in the next built `strip`. The attack is invisible to an auditor who reads the sources or the recipe.

4.2 Attack Overview

Figure 4 summarizes the attack in three stages. *Seed tampering* builds a trojaned seed containing a compromised `strip`. *Propagation* carries that malicious behavior across the NixOS bootstrap to every later `strip`. *Compromise* occurs when a subverted application executable runs on the victim's machine.

Seed tampering (Section 4.3) constructs the trojaned seed. Propagation (Section 4.4) spreads it through the bootstrap. Compromise (Section 4.5) describes the observable effect on the victim.

4.3 Seed Tampering

Our attack instantiates `strip` as a *self-propagating build utility*. An earlier copy of such a utility processes a newly built copy, which later assumes the same role in the bootstrap. Three conditions make this pattern possible, and `strip` satisfies all of them. First (C1), the bootstrap must trust the utility before it builds the utility from source, and `strip` ships in the bootstrap seed. Second (C2), an earlier copy must modify a newly built copy, and `strip` rewrites binaries during the fixup phase. Third (C3), the modified copy must later perform the same operation, and every later stage runs `strip` again for fixup. We call this propagation path from one copy to the next the *successor edge*.

We build the trojaned `strip` by injecting the payload into the ELF file of the genuine `strip` with the ELF transformations of Section 2.4.3.

The injector *appends* the payload at the end of the ELF file. This has no effect on execution at that point, because the region is not mapped in memory, as it is not referenced by the program headers. The injector then *repurposes* a non-functional `PT_NOTE` program header and turns it into a `PT_LOAD` entry that points at the newly added payload, so that the operating system loads it as an executable memory segment. This step assumes the target carries at least one spare `PT_NOTE` header, which holds for the toolchain binaries we

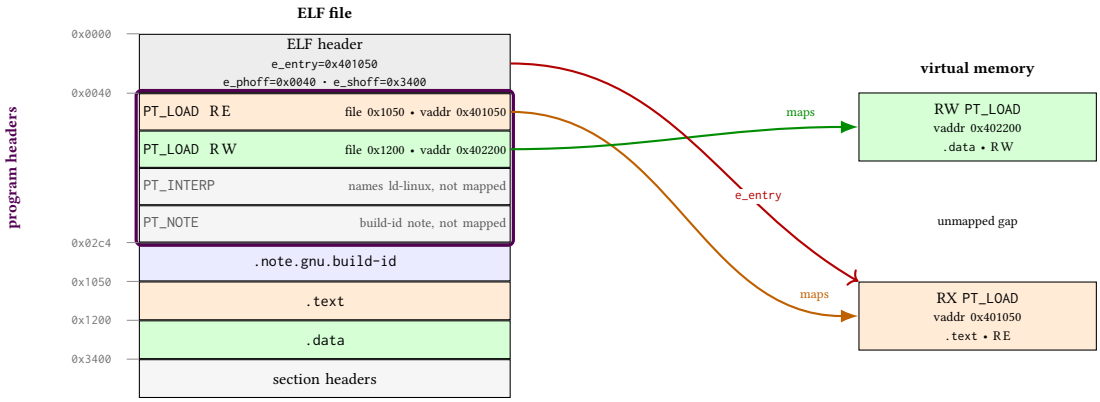
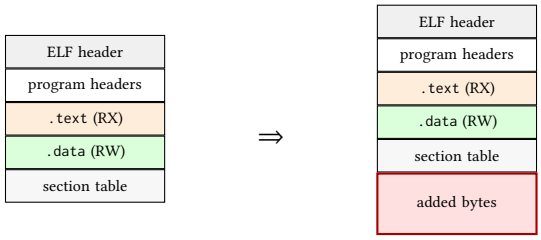
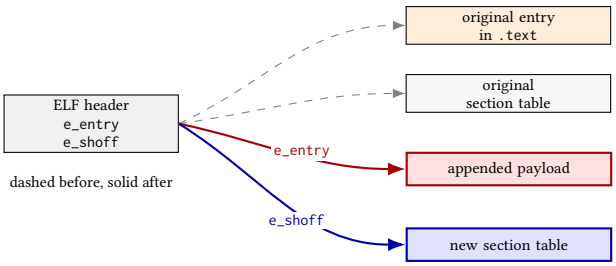


Figure 2: Simplified example of an ELF file and associated memory layout when running it. The program header table has four entries. The two `PT_LOAD` entries create mapped regions. Each `PT_LOAD` identifies its bytes by a file offset and declares the virtual address to map them at. `PT_INTERP` and `PT_NOTE` are metadata and map no new memory. Execution starts at `e_entry` inside the executable segment.



(a) *Append* adds new bytes at the end of the file.

(b) *Repurpose* retypes a spare `PT_NOTE` header as an executable `PT_LOAD` segment.



(c) *Redirect* repoints `e_entry` and `e_shoff` at the appended regions.

Figure 3: The three ELF transformations we define, each shown as a before/after edit to the file in the style of Figure 2.

target because the linker emits a build-id note (`.note.gnu.build-id`) by default. Finally, the injector *redirects* normal execution by modifying the `e_entry` ELF header, so that the kernel jumps directly to the payload when it executes the program. The first stage of Figure 4 shows this redirection, where the entry point that normally reaches the `.text` section is made to point at the injected code instead. The payload also embeds the original `e_entry` so that it can return to normal execution after performing its actions.

The produced ELF file must also stay coherent from the section-table view; otherwise, the nixpkgs post-processing step would discard the appended code. The injector therefore describes the payload as a new `.payload` section. It cannot enlarge the original section table in place, because the memory layout leaves no room for a larger table. It instead builds a new section table at the end of the file. This new table extends the original one with the `.payload` entry. The injector then *redirects* `e_shoff` to the new table, so the new table overrides the original. The old section table stays in the file but is now unreachable. Figure 5 applies all three transformations to the running example of Figure 2, showing the file before and after injection and the resulting memory layout.

4.4 Propagation

The second stage of Figure 4 shows how the payload spreads through the bootstrap along the successor edge. Each infected

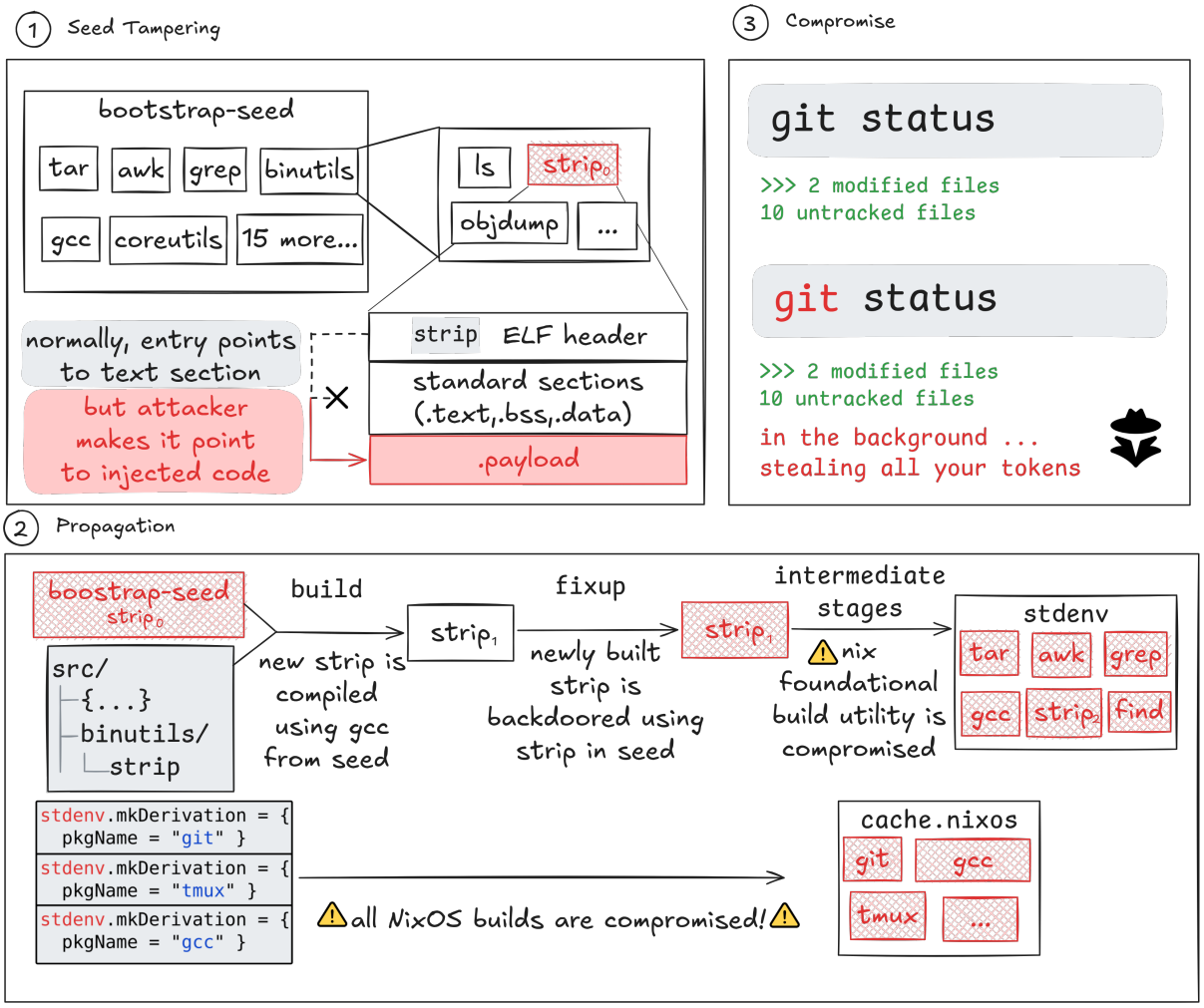


Figure 4: Overview of the attack design and implementation.

`strip` implants the payload into a newly built `strip`, which is then used by a later bootstrap stage.

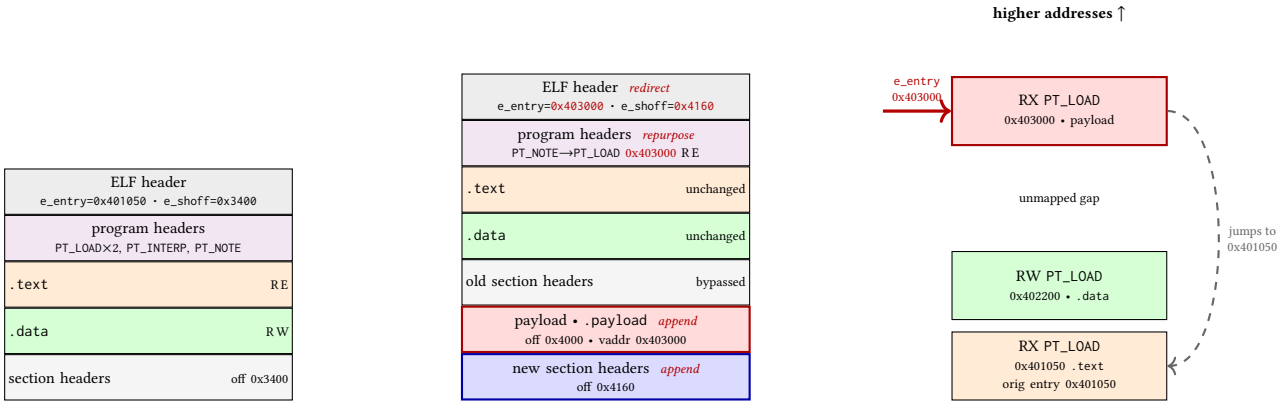
Every package build compiles the package from source and then runs `strip` over the installed binaries during the fixup phase. The upstream source and Nix recipe remain unmodified, but the build environment contains an infected `strip`. Let $\texttt{strip}_i$ denote the copy of `strip` used at bootstrap stage i, starting with the seed's trojaned $\texttt{strip}_0$. When `strip` is rebuilt, the build phase compiles a fresh $\texttt{strip}_1$ from unmodified source, and the fixup phase runs $\texttt{strip}_0$ over the newly built binaries, including $\texttt{strip}_1$ itself. Because $\texttt{strip}_0$ already carries the payload, it runs before `strip`'s own code, detects that it executes as `strip`, and uses the injector of Section 4.3 to implant the payload into $\texttt{strip}_1$ before stripping it. So $\texttt{strip}_1$ is backdoored despite being built from unmodified source, and it implants the payload into $\texttt{strip}_2$ in turn. The chain carries the payload from $\texttt{strip}_0$ to the `strip` of the final standard environment.

The compromised `strip` can also implant the payload into any eligible binary it fixes up. Because each package runs a fixup phase by default, and that phase invokes the compromised `strip`, every binary the distribution produces carries the payload.

4.5 Compromise

The third stage of Figure 4 shows the effect on the end user. The payload runs whenever a backdoored binary starts, and on every host it executes its malicious code.

The payload must attach to something every executable has, so that it infects as many executables as possible, and it must run without any help from the host. It therefore lives at the program's entry point, which the kernel reaches before any language runtime starts. This constraint forces the payload to be self-contained, so we write it as a *freestanding* program [5]. Such a program links against no libraries, avoids libc and the host's dynamic symbols, and calls the kernel directly. We need this because we implant the



(a) The finished file, before injection. **(b) After injection, two ELF header fields change, one program header is repurposed, and two regions are appended. The original program code and data are preserved.** **(c) At run time the payload maps as a new executable segment, runs first, and jumps to the original entry.**

Figure 5: How the injector rewrites the running example of Figure 2. It *redirects* `e_entry` and `e_shoff`, *repurposes* the spare `PT_NOTE` header into an executable `PT_LOAD`, and *appends* the payload (named by a `.payload` section) together with a new section-header table. The original program code and data are preserved. Execution begins in the appended segment and then returns to the host's original entry point.

identical bytes into executables with unrelated dependencies, so the payload can rely on nothing they provide.

Because it runs even before the ordinary `_start` routine, the payload is a careful guest. It saves the registers the kernel handed to the process, aligns the stack as the x86-64 calling convention requires, and later restores the registers that `_start` expects untouched. It *jumps* to the host's original entry point rather than forking or replacing the process, so the host keeps its PID, file descriptors, signal context, initial stack, and startup sequence.

Figure 6 traces the runtime path. ① The kernel enters the process at the implanted entry point, not the host's own. ② The payload saves the kernel's registers and aligns the stack. ③ It checks for a Nix build, emitting the infection marker outside the sandbox and staying silent inside. ④ It checks its own basename and, if it is `strip`, injects the payload into eligible files passed on the command line. ⑤ It restores the saved state and jumps to the host's original entry point.

The payload detects a Nix build by reading the `NIX_BUILD_TOP` environment variable, and it stays silent inside the sandbox so that its behavior never fails a functional test during a build. Outside the sandbox it executes its malicious code with the same permissions as the caller of the subverted executable. In our prototype that behavior is a single fixed infection marker, the string "`You've been pwned!\n`", written to standard output before the host runs. It stands in for any action the payload could take instead, such as stealing the user's credentials or tokens in the background.

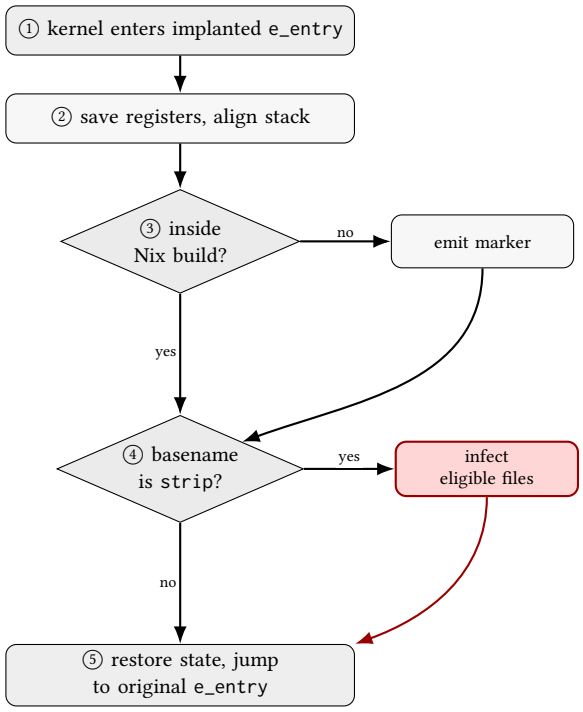


Figure 6: Runtime control flow of an implanted executable. Every host runs the prologue, but only a host named `strip` infects the files passed on its command line. The circled numbers match the walk-through in the text.

5 Experimental Evaluation

We evaluate the attack along three axes. First, does the compromised utility prevent any package from building? Second, which downstream outputs does it reach? Finally, does the payload injection impede the normal run-time behavior of subverted packages?

Table 1: Selection of executables in the trojaned graphical installer closure, by source-language ecosystem. Every listed binary carries the infection marker.

Ecosystem	Example binary	Size	Infected?
C / C++	`bash`	1,327 KB	✓
C / C++	`git`	56,313 KB	✓
C / C++	`sudo`	248 KB	✓
C / C++	`curl`	222 KB	✓
C / C++	`python3.13`	10,928 KB	✓
Python	`pydoc3.13`	110 KB	✓
Python	`idle3.13`	61 KB	✓
Rust	`rsvg-convert`	8,065 KB	✓
Go	`captree`	1,964 KB	✓
Lua	`lua`	328 KB	✓

5.1 Protocol

We use nixpkgs at revision `fef9403a3e4d`, whose bootstrap uses GNU binutils 2.44, and evaluate on `x86_64-linux`.

We construct a compromised `strip` starting from the original one provided by nixpkgs. We then build the graphical ISO installer, which covers a broad range of distribution software. We count the number of packages from that package set that build correctly under our compromised seed. We detect infected packages by running them and checking for console output containing the infection marker "`You've been pwned!\n`". Finally, we run the functional test suite of NixOS for the graphical image, to detect any behavioral degradation of the built packages.

5.2 Results

Buildability. The trojaned seed does not affect the buildability of the package set, and the complete graphical ISO image builds successfully. The runtime closure of the graphical installer, meaning the installer output and all store paths it references recursively, contains 1,199 package outputs and 3,799 user-invokable ELF executables. Its total on-disk size is 6.16 GB and the median package is 0.39 MB.

Reach. Out of 3,799 user-invokable binaries, 3,791 are CLI invokable. Of these, **3,790** print the infection marker when invoked normally. The single binary that does not print the marker is `firefox 147.0.3`. This is fully explained by the fact that the recipe passes `–disable-strip –disable-install-strip` to its build configuration, opting out of the strip phase entirely.

The graphical ISO image covers packages from multiple programming languages, confirming the language independence noted in Section 4.1, since the attack acts on binaries directly. The ten rows in Table 1 present a non-exhaustive view of the diversity of software present in the graphical ISO package set.

Runtime behavior. We run the functional tests of the NixOS distributions associated with the graphical installer. The tests exercise several end-to-end user workflows by running a full NixOS virtual machine in a graphical environment and emulating real workloads like interacting with graphical applications, or installing a NixOS machine from scratch. Again, every binary in the session VM, including `gnome-shell`, `mutter`, `nautilus`, and the GNOME control center, is built by the trojaned `strip`. No test fails, and the infection payload does not break any program covered by the test suite.

6 Discussion

6.1 Limitations

The current injector makes a few assumptions about the shape of the ELF file it works with. It assumes a little-endian ELF file with a spare `PT_NOTE` header and either an `ET_EXEC` binary or an interpreted `ET_DYN` one. Shared libraries, other architectures, static `PIE` executables without an interpreter, and binaries with no suitable note segment fall outside its current scope. These restrictions simplify the research prototype and are not fundamental limitations of the approach.

The implementation presented in this article may not generalize directly to Linux distributions other than NixOS because bootstrap procedures differ and may not use `strip` in the default build process. However, the self-propagating pattern is more general than this implementation and applies to any distribution whose build graph contains the successor edge from Section 4.3.

Our prototype is also deliberately easy to detect because the visible marker, the `.payload` section name, and the basename check provide experimental instrumentation rather than conditions required by the propagation pattern. We also make the attack propagate to *every* package in the distribution to test the extent of its reach. In practice, an attacker could keep the attack mostly dormant and propagate it only to future builds of `strip` until it reaches its ultimate target, possibly years later, which would make detection extremely difficult.

6.2 Other Candidate Carriers

To identify other possible carriers, we examine the bootstrap utilities against the conditions in Section 4.3. A utility must be trusted before it is built from source, modify a newly built copy of itself, and pass that role on to the modified copy.

We inspect the nine default build phases and their shared hooks in the evaluated nixpkgs revision.[1] We find close to forty bootstrap utilities invoked directly by name and check these invocations by tracing a minimal package build. This search excludes commands invoked indirectly through package Makefiles, which we discuss separately below.

Besides `strip`, `patchelf` meets all three conditions. It runs during fixup to adjust runtime library paths in newly built executables. When nixpkgs rebuilds `patchelf`, an earlier copy modifies the new one. A compromised copy could therefore infect its replacement, which would then process later builds.

`install` and `cp` are also plausible carriers. They copy newly built executables into their installation directories, giving them an opportunity to inject a payload. In particular, rebuilding coreutils uses an earlier `install` to copy the newly built `install` binary. The infected replacement could then propagate the payload during later installations. Unlike `strip` and `patchelf`, these utilities depend on the package's installation commands to reach their targets. We

[1] https://github.com/NixOS/nixpkgs/blob/fef9403a3e4d31b0a23f0bacebbec52c248fbb51/pkgs/stdenv/linux/bootstrap-files/x86_64-unknown-linux-gnu.nix

Table 2: Selected bootstrap utilities and the conditions of Section 4.3. C1 requires trust before building the utility from source. C2 requires an earlier copy to modify a newly built copy of itself. C3 requires the modified copy to perform the same operation in later builds. Parenthesized checkmarks indicate that the condition depends on the package's installation commands.

Utility	Phase	C1	C2	C3
`strip`	fixup	✓	✓	✓
`patchelf`	fixup	✓	✓	✓
`install`, `cp`	install	✓	(✓)	(✓)
`make`	build, install	✓	–	–
`sed`, `grep`, `patch`	patch, configure	✓	–	–
`tar`, `gzip`, `xz`	unpack, patch	✓	–	–
`ln`, `rm`, `mkdir`	all	✓	–	–

therefore consider `install` and `cp` plausible carriers, rather than established alternatives to `strip`.

The other utilities in Table 2 do not directly rewrite finished executables during their usual build tasks. For example, `make` runs build commands, `sed` and `patch` edit source text, and `tar` and `gzip` process archives. They therefore lack the mechanism that lets `strip` infect both downstream programs and later copies of itself.

6.3 Alternative Architecture

We built the attack as an entry-point *parasite*, a small payload injected into the genuine `strip`, but the same propagation also works with a *wrapper* architecture. In this version, the seed `strip` is replaced by a program that carries the genuine `strip` as embedded bytes, runs that copy at invocation time, and applies the payload to the files on its command line, rewriting each newly built copy into a fresh wrapper. Because the wrapper runs as an ordinary program rather than a constrained pre-`main` hook, it gives the attacker more control. It can act before, during, and after `strip` runs, rewrite the environment, carry any seed utility regardless of its ELF layout, and even replace `strip` with an unrelated program. Our parasite deliberately performs one synchronous, self-contained action before returning control from a runtime in which libc and the host's normal initialization are unavailable. More elaborate behavior remains possible, but it would require implementing additional runtime functionality directly and would increase compatibility risk.

We nonetheless use the parasite, because it is stealthier and less likely to break a build. First, the parasite leaves the genuine `strip` untouched, so an infected binary still looks like `strip` to anyone who inspects it, whereas the wrapper replaces `strip` with a different program that embeds a whole copy of it and no longer resembles the tool it stands in for (Figure 7). Second, the parasite hands control to the genuine `strip` and inherits its behavior exactly, whereas the wrapper must faithfully reproduce how `strip` is invoked, and any discrepancy risks corrupting an output or failing a package build.

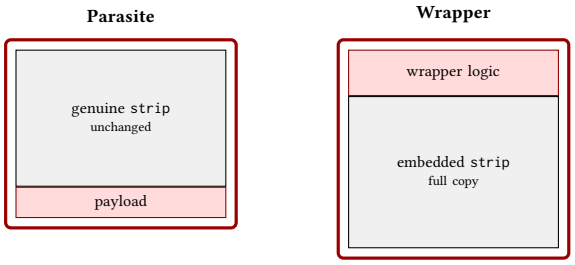


Figure 7: Comparison of the two possible attack architectures. The *parasite* leaves the genuine `strip` in place and appends a tiny entry-point payload. The *wrapper* instead replaces `strip` with a program that carries a full embedded copy of it. Gray marks genuine `strip` code and red marks the attacker's own additions.

Table 3: How Thompson's compiler and our build utility fill the same four roles.

Role	Thompson's compiler [23]	This work
Transformation	compiles source	strips ELF
Recognition	source pattern	host basename
Implantation	code generation	binary rewriting
Successor edge	compiles compiler	strips `strip`

7 Related Work

7.1 The Trusting-Trust Attack and its Generalizations

In his Turing Award lecture, Thompson [23] describes a compiler he modified to insert a backdoor into selected programs and reproduce the attack when compiling itself. The backdoor persists even after the malicious code is removed from the compiler's source. Our attack follows the same principle, but uses `strip` to modify compiled binaries. Whereas the compiler targets programs in the language it compiles, `strip` can infect eligible ELF binaries produced by different languages and toolchains. Table 3 compares the two implementations.

Other work extends Thompson's idea beyond the original compiler. Karger and Schell [9] describe compiler-mediated code insertion in their Multics vulnerability analysis. Spinellis [21] shows that a platform's security policy is untrustworthy once an untrusted agent runs on it, analogous to how an untrusted compiler implies that its source is also untrusted. Bratus et al. [2] argue that the essential ingredient is not a planted compiler bug but the way programs handle input, since any program that reads untrusted input can carry the same attack. Maynor [14] names the idea but does not instantiate the attack, and observes only that the weak link generalizes beyond the compiler to other trusted build tools. Our work supplies the missing instantiation. We show that a post-build utility, rather than a compiler, sustains a complete trusting-trust chain through a real distribution bootstrap. We also give a concrete end-to-end implementation of such an attack.

7.2 Verifying the Source-to-Binary Correspondence

A parallel line of work tries to establish trust in a binary by tying it back to its source. It targets the gap that our attack exploits, the divergence between the trusted source and the shipped binary. Wheeler [26, 27] introduces diverse double-compiling and supplies executable demonstrations, including a deliberately corrupted compiler and experiments with GCC. Diverse double-compiling rebuilds a compiler with an independent second compiler. It then checks that the two results agree. A disagreement exposes a Thompson-style implant that only one of the two toolchains carries. Reproducible builds pursue the same correspondence at distribution scale [10, 13]. They make every build produce bit-for-bit identical output. An independent rebuilder then confirms that a binary matches its source.

Both techniques secure the path from source to binary, but neither inspects the utilities that run after the compiler finishes. Our attack lives in exactly that gap. It hides inside `strip`, which the original build and every rebuild trust alike. Diverse double-compiling diversifies the compiler and not a build utility like `strip`, so the tampered utility sits on both sides of the comparison and cancels out. Reproducible builds confirm that a binary is deterministic and not that its build environment is honest, so a rebuild that reuses the same seed reproduces the same implant bit-for-bit. The recipe, the dependency records, and the source all stay intact, so neither check fires.

7.3 Bootstrappable Builds

The Bootstrappable Builds project [1] addresses the same problem at its root by shrinking the set of opaque binaries a distribution must trust. The full-source bootstrap in GNU Guix [17] reduces the binary seed to a few hundred bytes and rebuilds the whole toolchain from auditable source above it [16]. Camlboot [4] performs an analogous full-source bootstrap for the OCaml toolchain by stacking simple implementations and checking the result with diverse compilation.

7.4 Build-Time and Distribution Supply-Chain Attacks

Documented incidents show attacks against build and distribution pipelines rather than source repositories alone [6, 8, 11]. The SolarWinds compromise inserts its backdoor with SUNSPOT, an implant that watches for the compiler and rewrites a source file during the build, so the shipped binary diverges from the repository [6]. The xz and liblzma backdoor hides its payload in build fixtures and activates it through the release build machinery, again placing malicious logic in the build rather than in reviewable source [8, 11]. Ohm and colleagues catalogue hundreds of malicious open-source packages and organize their injection and trigger techniques into attack trees [19]. Defensive frameworks such as in-toto attest that each build step ran as intended and that artifacts pass unmodified between steps [24]. The Shai-Hulud worm [15] shows that attackers now pursue propagation directly. It steals a maintainer's credentials and republishes trojanized versions of the other npm packages that the victim controls, so one compromise spreads across the registry. This propagation is horizontal and source-level. Each infected package still ships malicious, reviewable source, and rebuilding that package from unmodified upstream source removes the infection. Most other incidents concern a single compromised build or package, and even Shai-Hulud propagates through reviewable source. Our attack is instead self-propagating through the build itself. One tampered seed reproduces the implant in later `strip` binaries built from unmodified source. The final `stdenv` remains infected without retaining a runtime reference to the seed, which neither source review nor a per-artifact attestation of an honestly executed build step detects.

8 Conclusion

We have shown that the trusting-trust attack is not limited to compilers. GNU `strip`, an ordinary build utility that never inspects or generates source, sustains the whole attack using only manipulations of finished ELF files. The implant propagates through one NixOS bootstrap execution, and the `strip` in the final standard environment remains infected without retaining a runtime reference to the malicious seed. On a real nixpkgs revision it reaches almost every binary of a complete graphical installer across a diversity of language ecosystems, and breaks no build or functional test.

References

[1] Bootstrappable Builds. [n. d.]. Bootstrappable Builds. https://www.bootstrappable.org/. Accessed July 18, 2026.

[2] Sergey Bratus, Trey Darley, Michael Locasto, Meredith L. Patterson, Rebecca Bx Shapiro, and Anna Shubina. 2014. Beyond Planted Bugs in "Trusting Trust": The Input-Processing Frontier. *IEEE Security & Privacy* 12, 1 (Jan. 2014), 83–87. https://doi.org/10.1109/MSP.2014.1

[3] Silvio Cesare. 1998. Unix Viruses. Unpublished manuscript.

[4] Nathanaëlle Courant, Julien Lepiller, and Gabriel Scherer. 2022. Debootstrapping without Archeology: Stacked Implementations in Camlboot. *The Art, Science, and Engineering of Programming* 6, 3 (2022), 13:1–13:26. https://doi.org/10.22152/programming-journal.org/2022/6/13

[5] cppreference.com. [n. d.]. Freestanding and hosted implementations. https://en.cppreference.com/w/cpp/freestanding.

[6] CrowdStrike Intelligence Team. 2021. SUNSPOT: An Implant in the Build Process. https://www.crowdstrike.com/blog/sunspot-malware-technical-analysis/. Accessed July 18, 2026.

[7] Eelco Dolstra, Merijn de Jonge, and Eelco Visser. 2004. Nix: A Safe and Policy-Free System for Software Deployment. In *18th Large Installation System Administration Conference (LISA '04)*. USENIX Association. https://www.usenix.org/conference/lisa-04/nix-safe-and-policy-free-system-software-deployment

[8] Andres Freund. 2024. backdoor in upstream xz/liblzma leading to ssh server compromise. https://www.openwall.com/lists/oss-security/2024/03/29/4. oss-security mailing list; CVE-2024-3094.

[9] Paul A. Karger and Roger R. Schell. 1974. *Multics Security Evaluation: Vulnerability Analysis.* Technical Report ESD-TR-74-193, Volume II. Electronic Systems Division, United States Air Force. https://csrc.nist.gov/files/pubs/conference/1998/10/08/proceedings-of-the-21st-nissc-1998/final/docs/early-cs-papers/karg74.pdf

[10] Chris Lamb and Stefano Zacchiroli. 2022. Reproducible Builds: Increasing the Integrity of Software Supply Chains. *IEEE Software* 39, 2 (2022), 62–70. https://doi.org/10.1109/MS.2021.3073045

[11] Julien Malka. 2026. Validating Supply Chain Transformations with Reproducible Builds: Lessons from XZ. (2026). https://hal.science/hal-05326226

[12] Julien Malka, Aman Sharma, Martin Monperrus, Stefano Zacchiroli, and Théo Zimmermann. 2026. Trusting-Trust Attack against an Entire Linux Distribution through Binary Manipulation — Replication Package. https://github.com/chains-project/trusting-trust-attack-strip. Archived in Software Heritage as https://archive.softwareheritage.org/swh:1:rev:4347dd81e18574d6018ff7db6d5abae9706e2bee.

[13] Julien Malka, Stefano Zacchiroli, and Théo Zimmermann. 2025. Does Functional Package Management Enable Reproducible Builds at Scale? Yes.. In *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*. 775–787. https://doi.org/10.1109/MSR66628.2025.00115

[14] David Maynor. 2004. Trust No-one, Not Even Yourself OR The Weak Link Might Be Your Build Tools. Presentation, Black Hat USA 2004. https://blackhat.com/presentations/bh-usa-04/bh-us-04-maynor.pdf

[15] Mike McGuire. 2025. The Shai-Hulud npm Malware Attack. https://www.blackduck.com/blog/npm-malware-attack-shai-hulud-threat.html. Black Duck Blog. Accessed July 20, 2026.

[16] Janneke Nieuwenhuizen. [n. d.]. GNU Mes. https://www.gnu.org/software/mes/. Accessed July 18, 2026.
[17] Janneke Nieuwenhuizen and Ludovic Courtès. 2023. 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/. GNU Guix project blog. Accessed July 18, 2026.
[18] Nixpkgs contributors. 2026. Nixpkgs Reference Manual. https://nixos.org/manual/nixpkgs/stable/. Accessed July 13, 2026.
[19] Marc Ohm, Henrik Plate, Arnold Sykosch, and Michael Meier. 2020. Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA) (Lecture Notes in Computer Science, Vol. 12223)*. Springer, 23–43. https://doi.org/10.1007/978-3-030-52683-2_2
[20] Eric S Raymond. 2001. The Cathedral and the Bazaar. *Readings in cyberethics* (2001), 309–338.
[21] Diomidis Spinellis. 2003. Reflections on trusting trust revisited. *Commun. ACM* 46, 6 (June 2003), 112. https://doi.org/10.1145/777313.777347
[22] System V ABI maintainers. [n. d.]. *System V Application Binary Interface: Executable and Linking Format*. The Linux Foundation. https://refspecs.linuxfoundation.org/elf/gabi4+/contents.html Accessed July 13, 2026.
[23] Ken Thompson. 1984. Reflections on Trusting Trust. *Commun. ACM* 27, 8 (1984), 761–763. https://doi.org/10.1145/358198.358210
[24] Santiago Torres-Arias, Hammad Afzali, Trishank Karthik Kuppusamy, Reza Curtmola, and Justin Cappos. 2019. in-toto: Providing Farm-to-Table Guarantees for Bytes and Bytes in Software Supply Chains. In *28th USENIX Security Symposium (USENIX Security '19)*. USENIX Association, 1393–1410. https://www.usenix.org/conference/usenixsecurity19/presentation/torres-arias
[25] Sergei Trofimovich. 2022. Nixpkgs Bootstrap Intro. https://trofi.github.io/posts/240-nixpkgs-bootstrap-intro.html. Accessed July 19, 2026.
[26] David A. Wheeler. 2005. Countering Trusting Trust through Diverse Double-Compiling. In *Proceedings of the 21st Annual Computer Security Applications Conference*. IEEE Computer Society, 28–40. https://doi.org/10.1109/CSAC.2005.17
[27] David A. Wheeler. 2009. *Fully Countering Trusting Trust through Diverse Double-Compiling*. Ph. D. Dissertation. George Mason University. https://dwheeler.com/trusting-trust/dissertation/html/wheeler-trusting-trust-ddc.html