Half a Second

The Backdoor That Almost Broke the Internet,
and the Invisible Labor Beneath It

Adrian Mastronardi

Half a Second

Half a Second

The Backdoor That Almost Broke the Internet, and the Invisible Labor Beneath It

Adrian Mastronardi

2026

First edition · Version 1.0.2 · 9 July 2026
www.half-second.com

For everyone who makes the open-source world possible, and above all the Lasse Collins among them: the maintainers who give their own hours, mostly unpaid and unseen, so the rest of us can build on what they hold up.

Contents

Preface

The systems that run the modern world are built to feel effortless. A message sends, a payment clears, a server answers a request from the other side of the planet, and the machinery that carries all of it stays out of sight by design. The security journalist Nicole Perlroth, surveying the cyberweapons arms race, named the cost hidden in that smoothness. "We had bought into Silicon Valley's promise of a frictionless society," she wrote, and had "never paused to think that, along the way, we were creating the world's largest attack surface" (2021). Her subject was the build-out of consumer devices and networks. The subject of this book lies one layer beneath hers, in open-source software, the code published for others to inspect, use, and improve that the rest of computing quietly runs on, and in the small, often unpaid, often unseen group of people who keep it running.

In March 2024 that dependency became visible for a moment. A backdoor, a hidden bypass of the normal checks on who is allowed into a system, was discovered inside XZ Utils, the open-source compression project whose `liblzma` library sits in the plumbing of Linux systems around the world. It had been put there over more than two years by a contributor known only as "Jia Tan," through patient, courteous, technically expert work that was almost indistinguishable from genuine help. It was found before it reached the stable releases that most of the world runs, the conservative versions shipped to ordinary users and production systems, because a software engineer at Microsoft named Andres Freund noticed unusual processor use around remote logins, one symptom

of which he could pin to a delay of about half a second, and decided the oddity was worth chasing.

This is an interpretive book, not an investigative one. Nearly everything in it was already public when the writing began, scattered across a security mailing list, the version history of a software project, a maintainer's brief public statements, the work of a handful of researchers who took the backdoor apart, and two years of reporting. That reporting has been excellent and fragmentary, and what has been missing is the shape: the single continuous account that gathers the pieces and asks what they add up to. The contribution here is that synthesis, and an argument that runs through the story rather than sitting on top of it. The book breaks no news. It does not try to identify the person behind "Jia Tan," it does not catalog every other attack of this kind, and it does not hand down a policy prescription. Those are other books, for other writers.

What it does instead is reconstruct, under a rule it does not bend: nothing in these pages is invented. Where the book attributes a thought or a reaction to Freund or to Lasse Collin, it is because one of them has described it in public, and the text says as much. The method is the one Patrick Radden Keefe set down at the front of *Empire of Pain*, his history of the Sackler family behind the opioid crisis: "no details are invented or imagined," and any thought or feeling assigned to a person traces to that person's own words or to someone who knew them (2021). The documentary record for this story is unusually deep. The oss-security mailing list preserved the disclosure and much of the correspondence that led up to it; the project's version history recorded the operation almost commit by commit; the statements of Collin, who had maintained XZ Utils almost single-handedly and without pay, together with the published reverse-engineering and the policy documents that followed, fill in the rest. Together, those public records are the book's spine.

The discipline matters most where the people at the center are quiet. Collin has said little to the press and owes no one more; the operator cannot be approached at all, and may never be named. Keefe, writing about subjects who declined to take part, set the precedent this book leans on. The Sackler family

"did not cooperate," he noted, and yet his account was "substantially built on the family's own words," assembled from the record they left rather than the interviews they withheld (2021). A person can decline to speak and still be portrayed fairly, carefully, and entirely from the record. That is the standard applied here to everyone who did not choose to be in these pages.

The book also belongs to a particular line of narrative nonfiction, and the inheritance is worth naming plainly so that the register reads as borrowed rather than assumed. Its opening move, an ordinary technical worker who trips over something small and follows it down, is the one Clifford Stoll made in 1989 in the book that founded the form, which begins not with a spy but with a baffled scientist: "Me, a wizard?" Stoll wrote. "Until a week ago, I was an astronomer, contentedly designing telescope optics" (1989). Its scale, a single modest artifact pulled apart until it discloses something vast, is the one Kim Zetter worked at in reconstructing the Stuxnet worm, a lone piece of software that "more than a dozen computer security experts around the world spent months deconstructing" before its meaning came clear (2014). This book works at that scale too: one backdoor, one continuous story, and beneath it the labor and the trust that hold up the digital commons.

Stoll also supplied the idea the whole book turns on, and that too is inherited rather than coined. "Networks aren't made of printed circuits, but of people," he wrote, meaning that the real substance of a connected system is the human arrangement of trust around it and not the hardware or the code (1989). The line has only grown more useful since 1989. The "people" now include a relatively small group of maintainers whose work supports infrastructure that trillions of dollars of economic activity assume to be stable, and the operation against XZ Utils got as far as it did by exploiting the trust that decides whose code is allowed to ship. Who ran that operation is still unknown. The book does not resolve it, and treats the blank as a finding rather than a failure: a world that runs on systems it cannot fully see can be reached by adversaries it cannot name, and the anonymity at the center of the operation is itself part of the argument.

A note for the reader who has never worked in software: no prior knowl-

edge is assumed. Every technical term that carries weight is explained in plain language the first time it appears, and the explanations are written so that a reader who already knows them will not feel talked down to. And a note on what the book cannot do. The complete account is not yet possible; parts of the record remain closed, and the attribution remains open. The aim, following the method Keefe describes, is not to claim the last word but to assemble the public record clearly enough that later reporters and researchers can see where the remaining blanks begin (2021). What follows begins where the discovery did: late on an ordinary evening, with a login that was taking longer than it should.

Part I

The Discovery

> *It marks a watershed moment in open source supply chain security, for better or worse.*
>
> — Russ Cox, *Timeline of the xz open source attack*

Chapter 1

Half a Second

It was late, and Andres Freund was running a benchmark. There was nothing unusual in that. Freund is a software engineer who works on databases, one of the longtime contributors to PostgreSQL, the open-source database system that sits underneath a large share of the world's applications, and in the last days of March 2024 he was doing what he often did in his own time: measuring how small changes affected the software's speed, chasing the fractions of a percent that accumulate into something a user can feel.

This is unglamorous work, and it is mostly patience. You change one thing, run the same operation thousands of times, compare the numbers against a baseline, and watch for movement the change alone should not explain. Done well, it is less an experiment than the tuning of an instrument: the engineer comes to know the ordinary sound of the machine so thoroughly that a wrong note announces itself. Most wrong notes are nothing. A benchmark that runs slow has usually tripped over something dull: a cache that never warmed up, a background task stealing cycles, the processor quietly throttling itself to keep from overheating. The discipline is to suspect the boring explanation first and chase it until it either accounts for the anomaly or runs out, and almost always it accounts for it. That habit is what makes a genuine find rare, and it is also what gives one weight when it finally comes. The computer Freund was testing

on ran Debian, one of the oldest and most widely used Linux distributions, in its perpetually updating development edition, the branch its users call by its code name, `sid`. It is the sort of machine a careful engineer keeps close at hand precisely because it runs new software before most people do, which makes it a little faster, a little more current, and a little more likely to behave strangely.

He was not hunting for anything, and that is worth holding onto. He was measuring database performance, not looking for an intruder, and what he found he found sideways. What caught his attention was not dramatic. He would later describe the discovery as a "small slowdown," the kind of reading that sits at the very edge of measurement and is easy to wave off as noise (2024a). On a machine that should have been close to idle, the part of the system that handles remote logins was doing something. Logging in over `ssh`, the standard encrypted channel engineers use to reach a computer across a network, was taking longer than it should, and the program on the far end that answers those logins, the SSH daemon, `sshd`, was burning an unusual amount of processor time to do it (Freund 2024b). In his own telling, what he kept seeing were brief, repeating bursts in which the processor lit up and then subsided: "occasionally there would be these 500 milliseconds . . . of high CPU usage that would then go down again," he said, "and that's just not what I was expecting" (2024). Something was spending effort that nothing had asked it to spend.

For a while the extra processor use looked like nothing in particular. The internet is a noisy place, and any machine exposed to it spends part of every day shrugging off automated attacks rattling the doors; Freund's first reading of the `sshd` activity was that it might be the "typical probing" every public server attracts (2024a). But it did not behave like probing. The effort was coming from the wrong place, at the wrong moment, for no reason he could name, and the longer he looked the less it resolved into anything routine.

Freund does not cast himself as a security researcher, and he has been careful about the distinction. Pressed on it, he has described his own background as only "security adjacent," someone who had run production systems and handled the occasional security problem without belonging to the institutions whose ac-

tual job is to catch attacks like this one (2024a).

He timed it. A login that ought to have finished in about three-tenths of a second was taking closer to eight-tenths, a difference of roughly half a second that had no reason to be there (Freund 2024b). It was the same half-second seen from two directions: the burst of processor activity on each login, and the extra time the login itself took. Around the same moment `valgrind`, a diagnostic tool that watches a running program for memory errors and other misbehavior, was registering complaints from somewhere inside that login path. None of it amounted to an alarm. It was a small cluster of odd readings on one machine, the sort of thing a noisier setup might have hidden, or the same person in a busier week might have explained away and forgotten.

He checked that it was real. The slowdown appeared even when he ran `sshd` in a stripped-down way that did not start it as a server at all, only far enough to do its initial work; that bare invocation took about half a second where it should have taken a hundredth of one (Freund 2024b). Whatever this was, it was not a quirk of his particular configuration or a misread number. It was real, and it was repeatable.

That half-second would become the most famous detail of the whole affair, and in the retelling it hardened into legend: Freund saw the delay, the delay tipped him off, and from there he pulled the thread that unraveled everything. The real sequence is more particular, and Freund has gone out of his way to correct it. The half-second was real, he says, but it arrived second. “I think the myth goes that’s what caused me to look into all of this,” he told one interviewer. “But the 500 milliseconds is actually true. . . . But I only got there after I already had seen the CPUs. . . . So it was like a secondary investigative perspective” (2024). What set him looking was the processor use and the misbehaving logins; the timing was the instrument he reached for to pin the thing down, not the spark. A Microsoft security researcher who later recounted the episode at a conference told it the same way: the half-second was “a kind of a legend,” and what actually triggered the investigation was that Freund “found that the SSH login in some of the testing he was doing was actually failing”

(Roccia 2024, 3:19). The legend is tidier than the truth, which is partly why it spread: a single number, a single instant of noticing, a clean line drawn from symptom to discovery. The real path was messier and more human, a careful person tugging at several loose threads at once until one of them held.

It matters who was at that keyboard, because the easy version of this story turns the catch into a miracle, and it was not one. Freund was not a random user tripping over a random delay. He had contributed to PostgreSQL for more than a decade and had been trusted to write changes directly into its code for years; in November 2020 the project's core team, the small group that steers one of the world's most relied-upon open-source databases, announced that he was joining it, noting that he had been "a contributor to PostgreSQL for over 10 years and a committer for 6," with his work concentrated on "replication, performance & scalability, storage and patch review" (2020). There is a quiet symmetry in that. The person who would catch an attack buried in critical open-source software was himself a steward of critical open-source software, on the maintaining side of the same fragile arrangement this book is about.

Performance and scalability were not a hobby he kept on the side. They were his specialty, the cast of mind that treats an unexplained half-second not as noise to be tuned out but as a question with an answer somewhere underneath it. A person who spends his working life learning the ordinary sound of a system is exactly the person to hear a wrong note in it. The contingency of the catch is real, and much of this book rests on it, but it was the contingency of the right anomaly reaching the right person, not of a delay so loud that anyone at all would have heard it.

And even then, by his own account, it nearly did not happen. Asked afterward how he had caught it, Freund gave a six-word answer: "Really required a lot of coincidences" (2024d, post 112180406142695845). Laid out, the coincidences pile up unnervingly, and no single one of them was the catch. It took the whole stack. He happened, that week, to be doing performance work at all. In another release cycle he would have been reviewing other people's code and tidying details rather than profiling, and, as he put it, he "might not have looked

at the time" (2024c, 01:17:04).

He happened to be running a version of `xz`, a compression package whose `liblzma` library could be called by other programs, old enough to still exhibit the symptom. Had the maintainer of his Debian system already moved to the newer release, `5.6.1`, which quieted the noisy `valgrind` complaint, he was, he says, "not sure" he would have figured it out (2024c, 01:17:25). He happened, too, to have already brushed against the same strangeness once before. Some weeks earlier, automated testing in PostgreSQL had begun throwing a `valgrind` error, and Freund, busy with other things, had done the reasonable thing: suppressed the warning, assumed a coming update would clear it, and moved on. When the strangeness surfaced a second time, on the benchmark machine, it landed on a mind that had already grazed it. He has said he is not sure he would have caught it on the second pass "if it hadn't in the back of my mind already been . . . primed for there's something really weird here" (2024). And he happened to be testing on a server processor with the chip's turbo boost, the feature that briefly raises its clock speed under load, switched off so his measurements would stay steady. On that slower, more even core the extra work was, in his words, "really exaggerated," drawn out into a window long enough to notice; on his faster laptop the same burst would have been briefer and easier to overlook (2024c, 01:18:17).

The strangest coincidence of all does not belong to Freund. It belongs to whoever built the thing he was beginning to circle. The operation was visible because it had been over-built. Had its concealed code done its job more cheaply, the extra effort would have cost not half a second but a few hundred microseconds, which is to say millionths of a second, far too little to surface as anything at all; with "less obfuscation," Freund has said, he "probably would not have noticed that anything was wrong" (2024c, 30:48). The thing was caught, in part, because whoever made it had hidden it too expensively. That irony runs under the entire story, and the craft of what they built is a subject this book comes back to.

None of which makes the catch a matter of luck and nothing else. The ac-

counts written in the first days got the balance right. One early explainer credited "sheer luck and Freund's careful eye," the two halves held in a single phrase (Goodin 2024). The researcher who narrated the discovery called it "a mix of luck, coincidence, and expertise," and said that without Freund's investigation the thing might never have been found at all (Roccia 2024, 39:43). Another security researcher, writing that same week, observed that Freund "had a unique testing environment and a set of coincidental setup issues which allowed him to discover the issue," and doubted that anyone else was running quite the same combination (Beaumont 2024). The environment was part of it: a quiet, low-noise benchmarking machine on which a half-second anomaly had nowhere to hide. On an ordinary desktop, Freund has noted, the ambient churn of something as unremarkable as an "idle Chrome" browser could have swallowed the signal whole (2024a). The honest account is neither luck nor brilliance but the two together: a faint signal, and a person equipped, by temperament and by setup, to register it.

There is an older version of this kind of discovery, worth naming because the shape keeps recurring. In 1989 the astronomer Clifford Stoll opened his account of chasing an intruder through the precursors of today's internet not with a break-in but with a bookkeeping error: a 75-cent shortfall in a university computer's accounts, a discrepancy small enough that nearly anyone would have written it off, which turned out to be the visible edge of a foreign espionage operation (1989). Stoll's own explanation of why he bothered doubles as a description of Freund's instinct. A large error is obvious and easy to find, he wrote, but "errors in the pennies column arise from deeply buried problems" (1989). He nearly let it go all the same. "Other things seemed more pressing, so I dropped the problem," he wrote of one of several moments when the investigation almost ended before it began (1989). What a story like that turns on is not the size of the anomaly but the choice, made or unmade on an ordinary afternoon, to treat a trivial one as worth pursuing. Freund's half-second is a discrepancy of that same family, and so is the territory it opens onto.

Set that chain against what was at stake and it turns vertiginous. No institu-

tion caught this. No automated scanner raised an alarm, no audit surfaced it, no monitoring system fired. What prevented a working backdoor already moving through the channels that feed widely used Linux systems from remaining hidden was a run of unrelated accidents that converged, in one week, on a single engineer's test machine. Nothing in the system had arranged a second chance. It is tempting to call that luck and leave it there. The more exact word is contingency: not that the catch was random, but that it was conditional, hung on particulars that could as easily have fallen the other way. The catch was contingent, not systemic, and that is the uncomfortable fact the rest of this book turns on.

What he had found, though it would be days before anyone could say so with any confidence, was not a bug. It was a backdoor: a deliberately built, deliberately concealed way into a system, placed there by someone who wanted quiet access to it. It had been hidden in backdoored releases of `xz`, the compression software package whose `liblzma` library sits quietly on a great many Linux systems, the systems that carry much of the traffic of the internet. Tracing it would consume the next several days and draw in distribution maintainers, security responders, and a scattered, self-assembling crowd of reverse engineers. Naming whoever had spent two years putting it there would prove, even now, impossible. All of that was still ahead. For the moment, the whole chain of consequence rested on a smaller and stranger fact: the security of a great deal of the world's digital infrastructure had just come to hang, with no one having arranged things that way, on whether one engineer at home judged a half-second worth chasing. He judged that it was.

Chapter 2

Blood in the Water

Freund had decided the half-second was worth chasing. That decision is the cheapest part of the story and the part on which everything else depends. He had a login path doing work nothing had asked of it, a profiler pointing at a place that made no sense, and a benchmark he had actually sat down to run. He could have filed the anomaly, suppressed the warning, and gone back to measuring database performance. Instead he stayed with it, and the staying took days.

It did not feel like a breakthrough while it was happening. It felt like one strange reading leading to another. “Initially, I didn’t think it was, like, a backdoor or anything,” Freund said afterward. “It was like a weird symptom. . . . but the more time I spent the weirder it got” (2024). That sentence is the chapter in miniature. There was no moment of revelation and no single clue that arrived wearing the answer. There was a symptom that refused to resolve and an engineer who refused to stop pulling on it.

The work fit itself into an ordinary week, which is part of why the legend leaves it out. The first day he gave over to the anomaly almost entirely: he had, as it happened, a day with no meetings on the calendar, and he spent most of it inside the problem. The next day was not his to spend. “I remember sitting in a bunch of meetings,” he said, “and, like, not really being able to concentrate

because it feels like . . . I should, I need to continue looking into this" (2024c, 28:41). The person who would catch one of the most patient supply-chain attacks ever found in open-source software was a Microsoft engineer paid to work in open source, and the chase still had the texture of ordinary work: a home machine, a calendar, and meetings through which the anomaly kept pulling at his attention.

He has also been candid that the closer the investigation came to its end, the less precisely he can reconstruct it. The strangeness built slowly at first, "gradually increasing," and then, once he understood what he was looking at, the experience compressed into something he can no longer time. "After that, once I figured out that it was nefarious," he said, "such a whirlwind, I kinda don't have the timing . . . down anymore" (2024c, 25:59). The chronology that follows is his, assembled in retrospect, and he is the first to say its final hours blur. What can be fixed is the shape: a couple of hours to be sure something was genuinely wrong, then, by his own estimate, a few days from that certainty to the warning he would send the world (2024c, 27:36).

The method underneath those days is worth slowing down on, because it is the opposite of the thriller version of discovery. What Freund did, over and over, was profile, record, and refuse the easy explanation. Profiling means running a program under an instrument that samples it thousands of times a second and reports where its time is actually going, rather than where a programmer assumes it goes. A slow login, by itself, only says that something took too long. A profile begins to say where the time went.

"To analyze I primarily used `perf record -e intel_pt//ub`," he wrote, naming `perf`, the standard Linux profiler, driven by Intel Processor Trace (`intel_pt`), a feature built into the processor that can record the path that execution takes through a program (Freund 2024b). The purpose of the setup was "to observe where execution diverges between the backdoor being active and not," and where the profile localized that divergence he switched to `gdb`, a debugger that can freeze a running program in place, "setting breakpoints before the divergence" and stepping through what came next (Freund

2024b). The pairing matters. `perf` could tell him where ordinary expectations and actual execution began to part company. `gdb` let him stop at the edge of that parting and walk forward instruction by instruction.

That description, written after the discovery, makes the comparison sound cleaner than it could have felt in the middle of the hunt. A profiler is most useful when the investigator can produce two runs that differ in one important way: the strange behavior appears in one and disappears in the other. Without that contrast, a profile can become a map of ordinary machine noise. Freund's task was therefore not only to inspect the slow path, but to make the slow path reliably come and go. Every condition he isolated (the way `sshd` was launched, the build flags in use, the version on the machine) mattered because it sharpened the comparison.

A tool like that does not answer the question for you. It gives you coordinates. The investigator still has to decide whether a hot path is normal, whether a missing name is just a build artifact, whether one run differs from another because the machine was noisy or because the software actually changed. None of this is exotic to a systems engineer, and all of it is specialized, which is exactly the point the previous chapter insisted on: Freund was not a random user tripping over a random delay. Catching the thing required someone who already lived inside these tools.

In a compressed version of the sequence, written while the disclosure was still new, Freund described the work as a chain rather than a leap. He had quieted the machine for micro-benchmarking, saw `sshd` processes using surprising CPU even though the attempted logins were failing immediately, profiled `sshd`, found time in `liblzma` that `perf` could not attach to a symbol, and remembered an odd `valgrind` complaint from PostgreSQL testing weeks earlier (Freund 2024d, post 112180406142695845). `valgrind` is a diagnostic tool that watches a running program's memory behavior and complains when the program does something that looks invalid. That is the investigation's real grammar: not "I saw the answer," but "I saw a symptom, and it connected to another symptom I had almost dismissed."

The rhythm of the work was repetition, and most of it was negative. A profile would point somewhere; Freund would form the most boring hypothesis that fit, a cache that had not warmed, a scheduling quirk, an artifact of measurement, and then set about killing it. When it died he formed the next one. This is the same instrument-tuning discipline that had made him notice the anomaly to begin with, now turned from a benchmark onto a target: run the operation, watch where the time actually goes, compare it against what the code claims to be doing, and follow the gap. Most of what an investigation like this produces is dead ends, and the dead ends are not failures. They are the mechanism. The space of innocent explanations is emptied one boring possibility at a time, until the only one left standing is the one nobody wants to be true. What separates the person who finds a thing like this from the thousands who run the same software untroubled is rarely a flash of brilliance. It is the willingness to spend several days subtracting comfortable answers.

The cast of mind is older than the tools, and it has a literature. Decades before any of this, the astronomer Clifford Stoll chased a different intruder through the precursors of the internet using nothing more exotic than the same disposition, and he set it down in terms that fit Freund precisely. “Well, I’d invested a couple hours in trying to understand a trivial problem,” Stoll wrote of the moment his own hunt nearly ended. “I got stubborn: dammit, I’d stay there till midnight, if I had to” (1989). Stubbornness, not genius, is the through-line. The work itself Stoll described as a kind of astronomy, in which “we passively observed a phenomenon, and from a few clues tried to explain the event and find the location of the source” (1989). That is the posture Freund was in: a handful of sparse, intermittent readings on one quiet machine, and the slow business of reasoning backward from them to a cause he could not yet see.

The modern equivalent of Stoll’s logbook was mundane. Freund had, he noted with some relief, “a shell history of what I had looked at,” a literal record of the commands he had run, which let him retrace his own path through the investigation without trusting it to memory (2024c, 27:17). Stoll had kept a logbook for the same reason, and offered the astronomer’s rule of thumb behind

it: “if you don’t write it down, it didn’t happen” (1989). Freund, without setting out to, had written it down.

The first solid clue was the one that had drawn his eye to begin with: `sshd`, the program that answers remote logins, was spending processor time it had no business spending. The attempted logins he was watching were not even succeeding. They were failing quickly, as wrong-user attempts should, and still the process was burning time. The previous chapter is named for that cost, seen from one direction as the extra half-second a login took. The new work was localizing it. The profiler did not merely report that `sshd` was busy; it reported where inside the work the cost was pooling.

Beneath the login machinery, the time was collecting under a routine: in Freund’s own description of the profile, the “high cpu in `sshd`” sat “below `_get_cpuid()`” (2024d, post 112187000944648334). The name is worth holding onto. It returns much later in this book, when the question turns from how the backdoor was caught to how it was built. A security researcher who walked an audience through the discovery months afterward paused on exactly this detail and told them to “keep that information in mind, because it will be interesting and important for the next part” (Roccia 2024, 5:35). For now it was simply a coordinate: the expensive thing was happening here, in a place a login had no reason to go.

The symptom was not Freund’s alone, which matters for a chapter built so heavily on one person’s account. Lasse Collin, who had maintained `xz` for years, would later write, in the commit that tore the backdoor out, that on machines exposed to the ordinary background noise of the internet the cost was visible in aggregate: “the backdoor noticeably increased CPU load, resulting in degraded user experience and thus overwhelmingly negative user feedback” (Collin 2024a). The two observations came from opposite ends of the same fact. Freund saw a single login crawl on an idle benchmarking machine; the maintainer was describing server-wide load on busy public hosts being rattled all day by automated bots. One cause, two manifestations, and the symptom stands corroborated from a channel entirely separate from the discoverer’s own

retrospective telling.

The second clue was stranger than the first, and harder to explain away. The processor time was not only in the wrong place; it was in a place that, by the rules of the software, should not have existed. When Freund followed the cost down into `liblzma`, the compression library that `sshd` had no obvious reason to be running at all, the profiler reported instruction after instruction executing in code it could not name. There was, in his flat description, "no symbol" (2024a).

In compiled open-source software, the human-readable names of functions, called symbols, are how a profiler can tell you what a program is doing; install the debugging information and the names appear. A missing symbol is not automatically sinister. Programs can be stripped of names, generated code can be hard to label, and optimized binaries can frustrate the clean map a reader imagines. The clue was the combination: unexplained time, inside a compression library, during a login, in code the normal naming machinery could not identify. Code with no useful symbol was code running where the machinery of transparency had gone dark at exactly the point Freund needed light.

Freund could see what the cost was even before he could see why. "If the above decides to continue, the code appears to be parsing the symbol tables in memory," he wrote. "This is the quite slow step that made me look into the issue" (Freund 2024b). The hedge is his own: *appears to*, the language of contemporaneous observation rather than finished analysis, and he kept that discipline throughout. The expensive thing, the half-second the book is named for, was a program laboriously reading through internal name-tables as it started up, before it had done any of the work a login actually requires. A compression library, at the moment a login began, was rummaging through the symbol tables of the program that had loaded it. What a library was doing there, Freund could not yet say.

The third clue was the one that nearly sent him the wrong way, and it is the clue the chapter's caution rests on. The anomaly was intermittent. It would not reproduce on command, and a symptom that comes and goes is an inves-

tigator's worst material and an attacker's best protection. For a stretch, in his own retrospective phrase, "it didn't reproduce" at all (2024a). The breakthrough, when it came, was a condition rather than a cause. The strangeness appeared only when `sshd` was started a particular way.

"It got a lot easier after I figured out that I could reproduce this without a running `sshd`," Freund recounted. "At some point, I forked my investigation to see why I couldn't reproduce a problem . . . in running `sshd` from a console. Then everything was normal. Only when I started it . . . from `systemd` . . . it was also clearly suspicious" (2024c, 25:12). Here a second piece of plumbing enters the story: `systemd`, the program that starts and supervises most of the services on a modern Linux system. A server does not normally start its login daemon because a person types the command into a terminal and watches it run. It starts it through a service manager, under a defined environment, with the dependencies and startup path a distribution has arranged for real machines.

That qualification matters because the clue was narrower than "started by `systemd` means malicious." Services can behave differently under a supervisor for perfectly ordinary reasons. The suspicious part was that the expensive, unnamed work inside `liblzma` appeared on the real service-launch path and disappeared on the simplified path Freund used for comparison. Launched by hand from a terminal, `sshd` behaved. Launched by `systemd`, the way it actually runs on many servers, it did not. For an investigator, that was valuable twice over: it made the phenomenon reproducible, and it said the behavior depended on context a compression library had no obvious reason to care about.

The other noise pointed the same way. The injected code, Freund found, "caused `valgrind` errors and crashes in some configurations," the backdoor's own assumptions colliding with an environment it had not anticipated and throwing off diagnostic complaints that a quieter system would never have produced (Freund 2024b). The complaint mattered because it was not a signature. No tool recognized the thing as a backdoor. The tool simply objected to memory behavior that looked wrong. The meaning came from the human being who remembered the complaint, connected it to the high CPU use, and kept testing

after the first clean run told him nothing was there.

One of the small accidents inside that complaint was technical enough to be easy to lose and important enough not to flatten. Freund later wrote that, as far as he could tell, `valgrind` would not have complained without `-fno-omit-frame-pointer`, a compiler setting that preserves a more regular record of function calls on the stack. The hidden code around `_get_cpuid()` expected that stack frame to look a certain way. With Freund's build, it did not. "Without having seen the odd complaints in `valgrind`," he wrote, "I don't think I would have looked deeply enough when seeing the high cpu in `sshd` below `_get_cpuid()`" (2024d, post 112187000944648334). The point, for now, is not the full stack-layout mechanics. It is that the attacker's assumptions met an investigator's particular debugging environment and made noise. The backdoor did not announce itself. It misbehaved.

This is the seam where the investigation's discipline shows, and it is worth marking what Freund did not yet claim. Stoll, the older detective, had a rule for exactly this passage of a hunt: "Record your observations. Apply physical principles. Speculate, but only trust proven conclusions" (1989). Freund speculated, as anyone would, and trusted little of it until it held. He had a slow step with no symbol, a cost that depended on `systemd`, a scatter of `valgrind` complaints, and a growing certainty that none of it was innocent. What he did not have yet was the shape of the thing. The full anatomy of the payload, what it intercepted and what access it granted, was not visible from where he stood, and he did not pretend otherwise. That anatomy, and the question of what the thing would have done at scale had no one caught it, are taken up in later chapters; from where Freund sat, both were still ahead of him. In the moment he had a set of symptoms and a hardening suspicion, and the suspicion still had to be turned into a proof.

Proof, in this setting, meant something colder than suspicion. A slow login could be argued with. A profile could be disputed as a local build problem, a strange machine, a measurement artifact, a distribution quirk. Even a crash under a diagnostic tool could remain a bug until someone tied it to a mecha-

nism. To warn the open-source security world that a trusted upstream project had been backdoored, Freund needed evidence another maintainer or responder could inspect without living inside his week of frustration. He needed the trail to leave the realm of "this machine is behaving strangely" and enter the realm of files, bytes, and reproducible differences.

The proof began with a wrong turn, which the honest version of the story keeps. For a while the evidence pointed at the wrong culprit. Part of what sent him there was the intermittency itself: when a build of the upstream source did not always reproduce the behavior, the natural suspicion was that the problem lived not in the project's own code but in something Freund's particular Debian system did to it on the way to becoming an installed program. "At first I thought this was a compromise of debian's package," he wrote, "but it turns out to be upstream" (Freund 2024b).

The distinction is the entire hinge of the catch, and it needs unpacking for anyone who has never had reason to think about how software arrives on a computer. Debian is not the author of `xz`. A distribution such as Debian takes upstream software, applies its own packaging rules and build settings, and turns the upstream release into installable packages for its users. If the problem had lived only in Debian's packaging, the failure would have been grave but bounded to that downstream path. If it lived upstream, in the official material published by the project itself, every distribution that trusted and built from that material had to be treated as a possible next link.

When an open-source project publishes a new version, it does not simply hand users the live, public history of its source code, the working text that developers read and edit. That history lives in a `git` repository, a versioned record of every change, hosted on GitHub for anyone to inspect. What the project often ships as its release is a tarball: a single packaged archive, a snapshot bundled for download and then built into the finished program by a distribution, a package builder, or a user. The two are supposed to correspond. A person who audits the public `git` source is supposed to be looking at the same code the tarball will build. Freund's investigation turned on the discovery that, for `xz`, they did

not correspond at all.

The malicious step lived in the tarball and nowhere else. "One portion of the backdoor is *solely in the distributed tarballs*," he found (Freund 2024b). He traced it to a single line in a build file named `build-to-host.m4`, one of the small scripts that prepare a program to be compiled. "That line is *not* in the upstream source of `build-to-host`, nor is `build-to-host` used by `xz` in `git`," he wrote. "However, it is present in the tarballs released upstream, except for the 'source code' links, which I think github generates directly from the repository contents" (Freund 2024b).

Read slowly, that one sentence is the whole concealment in a single breath: the trigger was absent from the public source, absent from the path the project actually built from in the `git` repository, present in the tarballs the world downloaded, and absent again from the archives GitHub generates automatically from the repository. This was not a crude mismatch anyone would necessarily see by opening the front page of the project. It was a mismatch between artifacts that most users reasonably treat as equivalent: the visible source history and the official release archive. The operator had not needed to hide the whole attack from public view. The operator needed only to split it across places that almost no one checked against each other.

The choice of a build file was part of what made the split work. Build systems are the backstage of software: the scripts and macros that turn source text into the executable files a machine can run. They are essential, but they are not where most readers of source code linger, and in many projects they are partly generated, partly inherited, and partly treated as plumbing. A small extra line in a file like `build-to-host.m4` did not look like a new feature, a new algorithm, or an obvious change to the compression library's public behavior. It looked like release machinery. That was why the tarball mattered. The malicious code did not need to live in the place where everyone expected to review the program. It could live in the machinery that prepared the program to exist.

The implication takes a moment to absorb, and it is the reason the discovery reached past one machine. The promise of open-source software is that anyone

can read it: the source is in the open, and enough eyes on it will catch what is wrong. The attack had been built to defeat exactly that promise without seeming to. The trigger was missing from the repository those eyes would read. The payload files were there, but disguised as test material and inert without the tarball-only step that activated them. The poison sat across the boundary between source and package, on the reasonable assumption that the two matched. Freund had not found a bug in `xz`. He had found that the relationship between what `xz` showed the world and what it shipped to the world had been quietly severed, and that the severing was the attack. And that is where the real attack surface of open-source infrastructure turns out to lie: not only in code that too few people read, but in trust that too few people can check.

He could watch it happen as the program built. "The build system would end up building the same file multiple times and replacing it with different content," Freund said, and when "there is like suddenly an object file that was like 50 KB or something larger than before, then that was clear that there was something bad going on" (2024). A build is allowed to be complicated. It may generate files, compile the same source in different modes, produce intermediate objects, or carry around test data that looks strange to a reader who does not maintain the project. The clue was not strangeness in the abstract. It was the same file being rebuilt and replaced with different contents, and an object file abruptly tens of kilobytes heavier than it had any reason to be.

At that point the investigation changed character. Before, Freund had been trying to account for symptoms: CPU time, missing symbols, launch conditions, `valgrind` complaints. Now he had a build process producing different bytes. The rebuild that sometimes produced different output was, in his words, the point where there was no longer a "legitimate reason" left to reach for (2024a). The discipline the previous chapter described, suspecting the dull cause first and chasing it until it either accounts for the anomaly or runs out, had finally driven every dull cause out of the room.

The payload itself, it turned out, had been hiding in plain sight the whole time. The bulk of the malicious code, Freund found, was already sitting in the

public repository, disguised as test data: "in an obfuscated form in `tests/files/bad-3-corrupt_lzma2.xz`" and `tests/files/good-large_compressed.lzma`, files committed upstream like any other (Freund 2024b). The disguise was plausible because a compression library's test suite is expected to contain malformed and unusual samples. A file with a name like `bad-3-corrupt_lzma2.xz` announces itself as junk on purpose; its job, in an innocent test suite, would be to prove that the decompressor fails safely on bad input. That is what made it useful cover.

The trigger that armed those files was the line that lived only in the tarball. Keep the trigger out, as it had been kept out of the `git` repository, and the test files were inert. Include the trigger in the release archive, and the build could extract and assemble what the repository alone did not appear to contain. This division of the attack, the payload committed in the open and the trigger slipped into the package, is the operator's craft, and the craft is the subject of a later chapter. Here it is enough that Freund had found the seam between the two and understood what it meant.

That finding did not require him to finish reverse-engineering the payload. It required him to show that the official release artifact had been made to do something the public source history did not explain. In a normal bug report, the difference between symptom and cause can remain partly open. In a backdoor disclosure, the difference is existential: if the official tarball differs maliciously from the `git` source, the trust decision has already changed, even before every instruction in the payload has been named. Freund had crossed that threshold.

What he refused to claim is as telling as what he found. In the same message that laid out the clues, he was scrupulous about the limits of his own account. "I am *not* a security researcher, nor a reverse engineer," he wrote. "There's lots of stuff I have not analyzed and most of what I observed is purely from observation rather than exhaustively analyzing the backdoor code" (Freund 2024b). By his own framing this was a report of symptoms and one decisive structural finding, not a complete dissection. The full account of how the payload worked, who could use it and to do what, would be assembled over the following days and

weeks by people whose actual profession was reverse engineering. The same man who had begun by doubting it was a backdoor at all now had to be the one to write the sentence that said it was. The restraint is not false modesty; it is the same disposition that produced the catch. A refusal to claim more than the evidence proved is precisely what let him recognize, in the end, that the evidence proved something extraordinary.

By the time he was ready to say it, the saying was plain. On March 29, 2024, Freund sent a message to oss-security, the public mailing list where the open-source world reports its security problems, and the message opened without preamble: "The upstream `xz` repository and the `xz` tarballs have been backdoored" (Freund 2024b). He had not gone straight to the public list. In the timeline Russ Cox later assembled from the record, Freund first raised the alarm privately on March 28, the day before the public post, to the handful of people who would need to begin moving at once (Cox 2024). The private warning bought a few hours; the public one started everyone else's clock. What is striking about the disclosure sentence is how little it strains. It reaches for no drama at all. An engineer who had spent days proving that trusted code could no longer be trusted announced it in eleven words and a period. The drama was in the fact, not in the phrasing.

The discovery has since been reconstructed by others, which is its own small check on the shape of the story. A 2025 MSR paper analyzing the operation describes the same opening Freund lived: an investigator who, "aided by a stroke of luck," began "with his observation of anomalous behavior: unusually high CPU usage by `sshd` processes during SSH logins," and followed it to the attack (Przymus and Durieux 2025, 93). The paper is secondary to Freund's own accounts for the discovery sequence, but its outline agrees with the first-person record on the shape, the luck included. The only thing it places differently is the date. The paper dates the discovery to March 28, the day Freund first raised it privately; the book dates the disclosure to March 29, the day the warning went public and the rest of the world's clocks began to run. The two are not in conflict. They are the two halves of the same hinge.

When Freund pressed send on the oss-security post, the investigation stopped being his. The half-second that had been a private irritation on a home benchmark was now a public fact, and the question it opened, what exactly had almost happened and how close it had come, belonged from that moment to everyone with a Linux system to worry about, which was very nearly everyone. Answering it would take the world the next forty-eight hours even to begin.

Chapter 3

Forty-Eight Hours

The message went out on Friday, March 29, 2024, under a subject line that named the catastrophe and softened nothing: “backdoor in upstream xz/liblzma leading to ssh server compromise.” It went to oss-security, the public mailing list where the open-source world airs its security problems (Freund 2024b). A researcher who later reconstructed the affair for a conference audience fixed the moment to the minute: the story “began on Friday, [March] 29, at exactly 8:51,” with Andres Freund’s message arriving under that subject line (Roccia 2024, 2:38). The archived post bears him out: its header is stamped 8:51 in the morning, Pacific time (Freund 2024b). What is not in dispute is the effect. The previous chapter ended with Freund pressing send. This one begins the instant the message arrived everywhere at once.

Freund’s email did more than raise an alarm; it documented, in a few clipped sentences, how he had already tried to route one. “Given the apparent upstream involvement I have not reported an upstream bug,” he wrote. “As I initially thought it was a debian specific issue, I sent a more preliminary report to security@…ian.org. Subsequently I reported the issue to distros@. CISA was notified by a distribution” (Freund 2024b). Three things are folded into that paragraph, and each would shape the hours that followed. The first is restraint: “apparent upstream involvement” is as far as Freund would go toward saying

that the project's own maintainer was implicated, and he went no further toward naming anyone, a discipline the rest of this book inherits. The second is the path the warning took: first to Debian's security team, then to the closed `distros@` list where the major distributions coordinate on embargoed flaws, then onward to the United States government. The third is buried in the passive voice of the last clause. CISA was notified by a distribution. The Cybersecurity and Infrastructure Security Agency (CISA), the civilian arm of American cyber defense, did not detect one of the most sophisticated supply-chain attacks yet found in open-source software. It was told, by a volunteer-run Linux distribution, after the fact.

The orderly look of that paragraph is a thing of hindsight. The decision underneath it had felt nothing like a procedure. "It felt very clear that it needed to go out very broad," Freund said afterward of the hours before he posted. "Mechanics of that, I was, like, wildly unsure about. . . . who do you inform?" (2024c, 57:31). The what was obvious and the how was not. There was no playbook for an engineer who had just established that a piece of the world's baseline software had been sabotaged by one of its own maintainers, and no switchboard to call. He improvised the routing, the routing worked, and the fact that it had to be improvised is part of what the next two days revealed.

Because the catch did not close the emergency. It opened it. Freund's half-second of suspicion had, by the time he posted, hardened into proof that the official releases of `xz` carried a backdoor. What no one yet knew on Friday morning was how far those releases had traveled, what exactly the hidden code did, or whether the people now scrambling to contain it were themselves already compromised. The disclosure converted a private investigation into a public race, and the race is the subject of this chapter.

To see why the response took the shape it did, it helps to know the shape it was supposed to take. Open-source culture treats openness as close to sacred. "In free software, doing things openly and transparently is normally almost a religious credo," the developer and writer Karl Fogel observes in his standard handbook on running such projects, before noting the single place the credo

yields. ". . . Security bugs are different" (Fogel 2020, 113). The exception is well worn: when a serious vulnerability surfaces, the usual practice is a quiet, time-limited embargo, a small circle of trusted developers and distributors who build and stage a fix in private and then disclose to the public all at once, so that defenders and attackers learn of it in the same instant. Freund's case fit none of that. There was no good-faith upstream to coordinate with, because the upstream was the problem. The maintainer's account was the adversary's account. And the backdoor was not a latent flaw waiting to be exploited; it was a live weapon already shipping in releases.

So the embargo behind the Friday post was unusually short and unusually strange, and most of it has been told only once, by one of its participants. By Red Hat's account, the coordination began on Wednesday, two days before the public disclosure. "On Wednesday, March 27, Andres contacted the Debian security team via their contact email," the company wrote in a later retrospective, "and let them know about the oddities he found in a SSH slowdown when using a new XZ library that was shipped by Debian" (Freire 2024).[1] Debian was the entry point because Debian was where Freund had first seen the symptom, and from there the alarm climbed into the cross-vendor machinery.

At Red Hat the tip did not stay with one team. "InfoSec involved Red Hat's Product Security (ProdSec) department, the ones in charge of securing the software that we provide to our customers and community," the account continues, and what assembled was "a global multidisciplinary team containing many of our finest developers, PenTesters, offensive analysts, managers, directors and program managers" who "started the then confidential efforts to properly establish, understand and assess the findings from Andres" (Freire 2024). The self-congratulation is Red Hat's own; what matters for the record is the speed and the shape. Within a day of an emailed tip, a major vendor had stood up a confidential incident team and split it along the two questions that would or-

[1] Red Hat dates Freund's first contact with the Debian security team to Wednesday, March 27 (Freire 2024). The widely cited public timelines place it a day later, on March 28 (Boehs 2024; Cox 2024). The discrepancy is a single day and is left unresolved here; the public disclosure date, Friday, March 29, is not in dispute.

ganize the entire response: had the malicious code reached anything already shipped, and what, precisely, did the payload do.

The closed mailing lists could only reach so far. To pull in the vendors who were not on them, Red Hat reached past the volunteer community to the federal government. "[D]ue to the limited reach of this community, we engaged the US Cybersecurity and Infrastructure Security Agency (CISA) to start a collaborative Vulnerability Information and Coordination Environment (VINCE) ticket," the company wrote (Freire 2024). The VINCE ticket became the hub the volunteer distribution lists could not be: a place where operating-system vendors, hardware companies, and national response teams could see the same facts at once, with Freund himself kept in the loop as the investigation moved. By Red Hat's count the ticket drew in a remarkable crowd. Collaboration moved "around the clock through 48 participating entities: BSDs, proprietary software vendors, national Security Response teams and the heavyweights of the hardware and software landscape," all "working together to advance and level the field on the findings . . . all done at speed to make the issue public as soon as possible" (Freire 2024).

The decision those participants reached was to abandon the long embargo and go public almost at once. The logic was the inverse of the usual case. An ordinary embargo buys time to build a fix before attackers learn of the hole; here the fix already existed, because the remedy was simply to stop using the two poisoned versions and fall back to an older one, and every additional day of silence was a day the backdoor kept shipping to anyone who updated. So the circle chose a moment and disclosed in concert. "On Friday, all the ducks (and there were a lot of them) were lined up," Red Hat wrote, and "the participants in the VINCE ticket, including the finder, agreed to bring this to the public as soon as possible. As soon as Andres' post went live in OSS-Security, CISA alerted the general public and Red Hat published a post as well" (Freire 2024). The simultaneity that looked, from outside, like the spontaneous reflex of a startled ecosystem had in fact been arranged. March 29 was Good Friday; the timing ruined a fair number of holidays, and the people in the ticket chose to ruin them

anyway.

Once the post was public, the distributions moved in something close to unison. Within hours the open-source news site LWN, reporting the disclosure, appended a single-line update that caught the breadth of it: “there are advisories out now from Arch, Debian, Red Hat, and openSUSE” (Corbet 2024). Four major distributions, four advisories, in the span of an afternoon. What that terse sentence compresses is worth slowing down on, because the advisories were not a formality. Each was a distribution telling its own users, in its own voice, to act now.

Debian’s was the spare, official kind. Its security advisory named the discoverer and located the damage precisely: “Andres Freund discovered that the upstream source tarballs for `xz-utils`, the XZ-format compression utilities, are compromised and inject malicious code, at build time, into the resulting `liblzma5` library” (Bonaccorso 2024). The hyphenated `xz-utils` is Debian’s name for the source package, and `liblzma5` the library it builds; the sentence is the tarball-versus-source split, seen from the receiving end, the same split the previous chapter watched Freund uncover. Then the directive, plain enough to function as an alarm: “Users running Debian testing and unstable are urged to update the `xz-utils` packages” (Bonaccorso 2024).

Red Hat’s was the loud kind. Fedora, the community distribution Red Hat sponsors, had shipped the poisoned versions to users on its fastest-moving branches, and the advisory did not stand on ceremony. “Yesterday, Red Hat Information Risk and Security and Red Hat Product Security learned that the latest versions of the ‘`xz`’ tools and libraries contain malicious code,” it began, naming the two compromised releases, `5.6.0` and `5.6.1`, and the freshly assigned identifier `CVE-2024-3094` (Red Hat 2024). Then it raised its voice. “PLEASE IMMEDIATELY STOP USAGE OF ANY FEDORA RAWHIDE INSTANCES for work or personal activity,” it shouted, in capitals, telling users that the development branch would be rolled back to a safe `xz` and could be trusted again only afterward (Red Hat 2024). A vendor bellowing at its own users in capital letters, inside a day, is the response arc in a single line.

openSUSE's was the ingenious kind. Its maintainers had rolled the package back on March 28 and shipped a clean snapshot built from a known-good backup, and the version string they gave the reverted package was a small piece of defensive craft: `5.6.1.revertto5.4`, a number engineered to sort as newer than the malicious `5.6.1` while actually carrying the older, safe `5.4` code, so that the ordinary update machinery would pull users back to safety rather than strand them on the backdoor (Meissner 2024). The same advisory carried a quieter, more honest admission. "As of March 29," it noted, "reverse engineering of the backdoor is still ongoing" (Meissner 2024). The distributions were pulling the package and ordering reinstalls before anyone fully understood what the package did.

Gentoo's response had begun, quietly, before the emergency was even public. Its terse first notice told users only what they needed: "A serious bug is being investigated. Please downgrade ASAP until we have a fix" (James 2024a, comment 2). But the commit that blocked the package was time-stamped the evening of Thursday, March 28, the night before Freund's post, and the developer Sam James noted that the block had gone up "since yesterday, after Debian, suse, fedora, et. al all did the same" (James 2024a, comment 2). The coordinated pull, in other words, was already running under embargo before the public ever heard of it. The discipline of that response shows in how Gentoo rated the danger. A security developer recorded that the team privately judged the "likelyhood of an impact on Gentoo" to be "small," then rated the bug at maximum severity anyway, reasoning that "with the backdoor not fully analyzed yet" they had to "opt for a A0 severity, assuming a (currently unclear) worst case scenario where the backdoor could still be included and may act with root privileges" (James 2024a, comment 9). A0 was the top of Gentoo's scale, the rating held for the gravest blockers. The responders defaulted to the worst case precisely because they could not yet rule it out, which is what a disciplined immune response looks like and also what an absence of margin looks like.

The consumer-facing distributions spoke in the second person. Kali Linux, the security-testing distribution, told its users plainly that if they had updated

"on or after March 26th, but before March 29th, it is crucial to apply the latest updates today" (Kali Linux 2024). The word was today. Even distributions whose stable users were never at risk reacted with a caution that rippled outward for days. Canonical, the company behind Ubuntu, found no exposure in the version of `xz` headed for its next long-term release, but it had pulled and rebuilt its packages out of caution, and the disruption pushed back a milestone: "the Beta release for Ubuntu 24.04 LTS (Noble Numbat) has been pushed to April 11, 2024 (previously April 4, 2024)" (Zemczak 2024). That slip came on April 3, days into the emergency rather than in its first hours, and it is a useful measure of how far the uncertainty propagated. A well-funded corporate distribution whose stable channel was never in danger still lost a week of its flagship release schedule to a backdoor in a compression library.

While the distributions reverted, the platform at the center of the story did something blunter. The public `xz` repository was hosted on GitHub, and on the night after the disclosure GitHub took the whole thing down. A participant in the Gentoo bug thread reported the moment to the minute: "Github has disabled the XZ repository entirely as of 1:29 AM UTC" (James 2024a, comment 17). The authoritative repository for the code the world was now urgently trying to inspect had vanished mid-investigation.

The takedown was sensible as triage and clumsy as justice, because GitHub did not only suspend the account behind the backdoor. It suspended the project's longtime maintainer too. The co-maintainer who had introduced the backdoor operated under the name "Jia Tan," a persona this book takes up later and whose human author has never been identified; the other account belonged to Lasse Collin, who had built and maintained `xz` for the better part of two decades and had been manipulated into accepting "Jia Tan's" help. GitHub suspended them both. In Collin's own later accounting, "GitHub accounts of both me (Larhzu) and Jia Tan were suspended. Mine was reinstated on 2024-04-02" (Collin, n.d.). For four days the maintainer of the compromised project was locked out of it while the rest of the world picked through the wreckage.

Inside the distributions the first hours were less a fix than a triage operation still defining its own scope. In Debian's bug thread a developer answered the obvious, anxious question, what should we do, with a candor that captures the moment: "Yes, a multi-team task force is working on it and will inform users once it is known how to proceed, inclusing how much to throw away and rebuild" (Hess 2024, msg #97). How much to throw away and rebuild was not rhetorical. Part of what made the scramble disorienting was that the responders could not immediately be sure they were clean themselves. As one Debian contributor pointed out, the people who build and upload the distribution's packages tend to run exactly the fast-moving branches the backdoor had reached: "many people (and I'd guess that includes DDs/DMs) run their workstations/laptops with testing/unstable," he wrote, using the project's shorthand for Debian Developers and Maintainers, so it was "not enough to know that Debian stable is likely not affected" (Hess 2024, msg #82). The channels that carried the backdoor and the channels that carried the response were, to an uncomfortable degree, the same channels.

As the distributions pulled the package, a second response was already underway, faster and more diffuse: the worldwide, self-organizing effort to take the thing apart. It started within hours. Filippo Valsorda, a well-known cryptography engineer, posted his first reaction less than three hours after Freund's email, and it reads like a man watching a building collapse and cataloging the architecture on the way down. "Woah. Backdoor in `liblzma` targeting `ssh` servers," he wrote. ". . . It has everything: malicious upstream, masterful obfuscation, detection due to performance degradation, inclusion in OpenSSH via distro patches for `systemd` support… Now I'm curious what it does in `RSA_public_decrypt`" (Valsorda 2024, post 3kotxq46zj223). Two clauses in that breathless inventory reach backward and forward through this book. "Detection due to performance degradation" is Freund's half-second seen from the outside, the entire catch compressed into four words. And "inclusion in OpenSSH via distro patches for `systemd` support" is the supply-chain path a later chapter has to explain, the unlikely chain by which a compression library ended

up inside the program that guards remote logins. The last line, the curiosity about `RSA_public_decrypt`, points straight at what the backdoor was for, a question the next chapter takes up.

What followed was collaboration in real time, conducted in public and governed by the community's own etiquette. "I'm watching some folks reverse engineer the `xz` backdoor, sharing some *preliminary* analysis with permission," Valsorda posted, the asterisks his own and the "with permission" a small courtesy in the middle of an emergency (Valsorda 2024, post 3kowjkx2njy2b). This is the distributed, public system that had not made the catch, now doing the next job: not one institution dissecting the payload behind closed doors, but a scattered crowd of specialists narrating the analysis to one another as it happened, each deciding what to share and when.

The work produced practical checks almost as fast as findings. Freund's own disclosure email, written before any of the reverse engineering had finished, already carried a practical attachment: "Vegard Nossum wrote a script to detect if it's likely that the `ssh` binary on a system is vulnerable, attached here. Thanks!" (Freund 2024b). The point is the clock: the public triage began before the payload was fully understood. openSUSE had said as much on the day (Meissner 2024), and the tooling and the triage advanced alongside the analysis rather than after it.

As the analysis spread, the incident was also being entered into the formal machinery of vulnerability tracking. It received an identifier, `CVE-2024-3094`, the standard catalog number by which the world's security systems would refer to it ever after, assigned by Red Hat in its role as a numbering authority. The catalog entry carries a credits field, and the field is worth reading for what it says about everything that came before. The record of a patient, multi-year operation names exactly one human being, and it is the engineer who tripped over it by accident: "Red Hat would like to thank Andres Freund for reporting this issue" (CVE Program 2024). The attribution of the operator stays open, as it does throughout this book. The attribution of the discovery is fixed, in the permanent record, to a man who had been running a database benchmark.

The federal government's own statement arrived the same day, in the spare register of an agency that had been told rather than tipped off. "CISA and the open source community are responding to reports of malicious code being embedded in XZ Utils versions `5.6.0` and `5.6.1`," the alert read, before glossing, for a general audience, what the obscure package even was: "XZ Utils is data compression software and may be present in Linux distributions" (Cybersecurity and Infrastructure Security Agency 2024a). The phrasing is quietly telling. CISA placed itself beside the open-source community, "CISA and the open source community are responding," not above it, which is an accurate description of where the government stood in this story. Its instruction was a single sentence asking for three things: "CISA recommends developers and users to downgrade XZ Utils to an uncompromised version—such as XZ Utils `5.4.6` Stable—hunt for any malicious activity and report any positive findings to CISA" (Cybersecurity and Infrastructure Security Agency 2024a). Downgrade, hunt, report. A private discovery had become, within a day, a government-issued call to act.

That CISA had a role to play at all rested on groundwork laid before the incident. Six months earlier, in September 2023, the agency had published a roadmap for open-source security that staked out exactly this kind of coordinating function: "CISA will continue to coordinate vulnerability disclosure and response for OSS vulnerabilities by leveraging relationships with the OSS community," it had written, anticipating even the pattern the XZ Utils response would follow, processes to "specifically look for upstream issues in open source packages that critical infrastructure organizations depend on and quickly notify affected users of the identified vulnerabilities" (Cybersecurity and Infrastructure Security Agency 2023, 8). The roadmap described coordination after a vulnerability surfaces, not detection of a slow trust compromise before release, which is the limit worth keeping in view. But it is why a federal alert about a compression library was part of the machinery at all.

The first operational question that machinery faced was brutally concrete: who was running the poisoned versions, and who could prove they were not.

That is the question a software bill of materials, an itemized inventory of the components inside a piece of software, is meant to answer. As a federal report on the format put it, some uses "require complete or mostly complete graphs, such as the ability to 'prove the negative' that a given component is *not* on an organization's network" (National Telecommunications and Information Administration 2021, 12). An inventory like that does nothing to catch a hidden backdoor in advance; what it speeds, once the alarm is public, is exactly the frantic accounting the response now demanded.

Step back from the forty-eight hours and the speed of all this cuts two ways. The response was fast, broad, and competent, and its very competence throws into relief how little of it had anything to do with the catch. Every distribution that reverted, every researcher who dissected the payload, every practical check that followed, all of it was triggered by the disclosure. None of it found the backdoor. That was the burden of a separate observation, made that week by the security researcher Kevin Beaumont, and it is the uncomfortable core of the chapter. "Nobody else had raised concerns," Beaumont wrote, "and I don't believe any existing security tooling or processes would have caught this (I realise there will be a torrent of vendors claiming they detect this… but they will detect this now that somebody told them)" (Beaumont 2024). The jab at after-the-fact detection marketing is earned. No public alarm and no automated control had surfaced the thing before Freund; the entire apparatus that moved so impressively on Friday had been inert on Thursday.

What that apparatus did accomplish, once it moved, was real, and Beaumont named it precisely: "Because Andres privately researched the issue and got the Linux distributions to take it seriously, he averted this reaching any kind of wide (or even small) deployment in the real world" (Beaumont 2024). The backdoor was evicted before it reached the stable releases that most of the world actually runs. Red Hat, looking back, allowed itself the understatement that the episode "had the potential to really hurt the open source community" (Freire 2024). The potential was the point, and how narrowly it was contained is the subject of the chapter that follows.

It is tempting to read the clean disclosure as evidence that the system simply works well, and a comparison cuts against that comfort. A decade earlier, in 2014, a flaw called Heartbleed had exposed a different weakness in the same broad ecosystem, and its disclosure had gone badly. A peer-reviewed study of that episode found that "patching was delayed because the Heartbleed disclosure process unfolded in a hasty and poorly coordinated fashion," with several major vendors "not notified in advance of public disclosure" (Durumeric et al. 2014, 486). Against that, Freund's single deliberate email, handing the distribution security teams the initiative before any public scramble, looks like a model of coordination. But Heartbleed also supplied a measurement that should sober anyone inclined to relax. Even when disclosure went well enough that "the most prominent websites patched within 24 hours," the same study "observed vulnerability scans from potential attackers within 22 hours" (Durumeric et al. 2014, 486). The window between a flaw going public and attackers moving on it is counted in hours, not weeks. Heartbleed was an accidental bug, found in good faith, not a deliberate backdoor, and its numbers are its own; but it measures the clock the XZ Utils responders were racing, and explains why a circle of forty-eight entities chose to ruin their holidays rather than wait.

So the forty-eight hours resolve into a single, double-edged fact, and the book's argument lives in the hinge between its halves. The cure was systemic. The catch was not. Once the alarm was public, the open, distributed mechanism functioned about as well as anyone could ask: an outsider had found the thing, a coordinated disclosure had been arranged across nearly fifty organizations, every major distribution had reverted within hours, a scattered crowd of researchers had reverse-engineered the payload while practical detection began, and the backdoor had been contained before it reached a single stable release. That is not the picture of a fragile system failing. It is the picture of an immune response working. What the same forty-eight hours also show is that none of that capacity was ever pointed at finding the backdoor in the first place. The response was a cure the system could mount only because an accident had bought it the time to do so: a chain of coincidences, set out in the previous chapters,

that surfaced the anomaly, met by an engineer with the rare instincts to chase it down. The system had no built-in margin to catch what Freund caught. It had a formidable capacity to react once he had.

The clearest verdict on how the disclosure was handled came, in the end, from the person it had hurt most. When Collin returned to his project and tore the backdoor out, he used the commit message to thank the man who had exposed the betrayal of his life's work: "Thanks to Andres Freund for finding and reporting it and making it public quickly so others could act without a delay" (Collin 2024a). The maintainer whose account had been suspended, whose project had been hijacked, and whose trust had been the attack's actual target endorsed the very choice that broke the security world's usual quiet, going public at once, because the only thing that made the response possible was that nobody waited.

That endorsement should not be mistaken for a clean triumph, and the responders themselves were the first to say so. Three weeks after the disclosure, when the acute emergency had passed, the same Debian developer who had worried about compromised maintainers noted what the fast, public response had quietly left undone. "It feels as if there are still many discussions about how to prevent such things in the future," he wrote, "but less so about the concrete fallout of the particular backdoor, where it seems most people were lead to conclude from media reports, that an attack was only possible if `sshd` was actually running an reachable" (Hess 2024, msg #152). The immune response had been public and quick, and it had also turned to the comfortable question, how do we stop the next one, while the unglamorous one a worried user actually had, am I safe, went underanswered. That is the honest shape of it: neither the triumph Red Hat described nor the doom the headlines reached for, but a real capacity that worked because, this once, it was given the time. The harder question is the one the responders were racing, mostly without stopping to articulate it. What, exactly, had almost happened? What would the backdoor have done if Freund had moved a few days slower, or never run the benchmark at all? That is the counterfactual the next chapter takes up.

Chapter 4

What We Almost Lost

The responders never quite stopped to ask the question aloud, because they were too busy answering it: what, exactly, had almost happened? Strip away the panic and it has a clean shape. Suppose the half-second had gone unremarked. Suppose Andres Freund had been traveling that week, or had written the latency off as a bad benchmark, or had simply never run the test, and the two poisoned releases of `xz`, versions `5.6.0` and `5.6.1`, had ridden the ordinary update machinery out of the fast-moving development channels where they had surfaced and into the stable distributions that most of the world actually runs. What would the world have woken up to?

It helps to begin with what the compromised software is, because the answer is the first measure of the stakes. `xz` is a compression utility, one of the small, unglamorous programs that shrink files for storage and transport and expand them again on the other end; `liblzma` is the library inside it that does the actual work, a piece of code that other programs call without ever thinking about it. It is precisely the kind of component that is everywhere and noticed nowhere. Nadia Eghbal, in the most careful account of the labor beneath digital infrastructure, drew the distinction that makes the danger legible: “we directly rely on open source code to keep our phones, laptops, cars, banks, and hospitals running smoothly,” she observed, and “if an open source project goes down,

it can literally break the internet" (Eghbal 2020, 14). The phrase is bounded, not breathless. But it names the right order of magnitude. The thing with a backdoor in it was not an app. It was part of the floor.

The contemporaneous record reached, almost at once, for the language of the near miss. Akamai, summarizing the incident days later, allowed itself one controlled flourish: "This backdoor almost became one of the most significant intrusion enablers ever — one that would've dwarfed the SolarWinds backdoor. The attackers were almost able to gain immediate access to any Linux machine running an infected distro, which includes Fedora, Ubuntu, and Debian. Almost" (Akamai Security Intelligence Group 2024). The triple "almost" is doing real work. The word can be read two ways. It can mean the catastrophe did not happen, so the system held. Or it can mean the catastrophe was averted, not absent: that it was fully built, fully capable, and stopped by something outside the design rather than by the design itself. The second reading is the one the record supports. Freund himself supplied the first caution, in the disclosure email: "Luckily `xz 5.6.0` and `5.6.1` have not yet widely been integrated by linux distributions, and where they have, mostly in pre-release versions" (Freund 2024b). Close, then, but not yet everywhere. The task is to say precisely what the thing could have done, and to whom, without letting "what could have happened" inflate into a claim the evidence does not carry.

Start with what the backdoor did, because the first public account of it was wrong in an instructive way. The earliest widely read disclosure-day report, in the open-source news outlet LWN, framed the threat as a login problem: "It appears that the malicious code may be aimed at allowing SSH authentication to be bypassed" (Corbet 2024). Authentication is the step where a server checks that a visitor is who they claim to be, and an authentication bypass means slipping past that check, logging in without the right credentials. That was bad, but it was not the whole of it, and within days the reverse engineers had corrected the frame. As Akamai put it, "It was originally reported as an SSH authentication bypass backdoor, but further analysis indicates that the backdoor actually enables remote code execution (RCE)" (Akamai Security Intelligence Group

2024). Remote code execution is the more serious capability, and the one worth defining carefully: it means the ability to make a machine run commands of the attacker's choosing, issued from somewhere else on the network, with the privileges of the vulnerable process. Here, as the demonstration below shows, that meant `root`. Not "log in as someone." Run anything.

The cryptographer Filippo Valsorda, narrating the live teardown as it unfolded, drew the line in a single terse sentence: "It's RCE, not auth bypass, and gated/unreplayable" (Valsorda 2024, post 3kowjkx2njy2b). Russ Cox, whose timeline became one of the standard references, stated the same capability in a sentence a general reader can hold: the backdoor "watches for the attacker sending hidden commands at the start of an SSH session, giving the attacker the ability to run an arbitrary command on the target system without logging in: unauthenticated, targeted remote code execution" (Cox 2024). Three properties are folded into that sentence, and each matters. The commands arrive at the start of a session, before any login: pre-authentication, in the jargon, meaning the server runs the attacker's instructions before it has decided whether the visitor is allowed in at all. They are targeted, aimed by the operator rather than sprayed. And they execute arbitrary code, anything the operator chooses to send.

In a later reconstruction for a DEF CON audience, the Microsoft security researcher Thomas Roccia inventoried the capability as three commands the hidden code would honor. The first was, in his words, "an SSH authentication bypass using root login," the power to log in as `root`, the all-powerful administrative account, with no valid password; the second "a remote code execution," the power to run an arbitrary command; the third simply closed "the pre-authentication [connection] that [was] previously open[ed]," tidying up after itself (Roccia 2024, 29:22). The machinery behind that second command was a single, ordinary function in the standard C library, `system()`, which takes a line of text and hands it to the shell, the part of the system that runs commands. The route was `system()`, the library call, a detail worth fixing because it is often confused with `systemd`, the unrelated service-management program

through which the backdoor reached the login server in the first place. The trigger had its own elaborate machinery: cryptography, test files, and concealment. For the counterfactual, what matters is the plain fact of the capability. A reachable, backdoored server could be made to run whatever its attacker wanted.

That this was a finished weapon and not a latent flaw is not a matter of inference. A security researcher, Anthony Weems, built and published a working trigger, described in its own documentation as a "cli to trigger the RCE assuming knowledge of the ED448 private key" (Weems 2024). On a vulnerable server he set a watchpoint, connected, and had the backdoor run the harmless command `id`, which reports the identity of the account under which a command is running. The answer came back "uid=0(root) gid=0(root) groups=0(root)" (Weems 2024). Those eleven characters, `uid=0(root)`, are the stakes in miniature: an arbitrary command, sent across the network to a server that had authenticated no one, executing with the highest privilege the machine has. *Ars Technica*'s Dan Goodin gave the general-reader version, with the right note of honesty about its limits: "Anyone in possession of a predetermined encryption key could stash any code of their choice in an SSH login certificate, upload it, and execute it on the backdoored device. No one has actually seen code uploaded, so it's not known what code the attacker planned to run. In theory, the code could allow for just about anything, including stealing encryption keys or installing malware" (Goodin 2024). What we almost lost was not a specific act of sabotage. It was a general-purpose key to the back of the house, and what the holder of the private key would have done with it remains, deliberately and permanently, unknown.

The phrase "predetermined encryption key" points to the design property that makes the counterfactual genuinely frightening and at the same time keeps it honest. The door opened only for whoever held the matching private key. Valsorda spelled this out when asked what he meant by calling the backdoor "gated": "it takes the attacker's private key to use the backdoor (it's NOBUS)," he wrote, and by "unreplayable" he meant that "even if we observe an attack against one host, we can't reuse it against another host" (Valsorda 2024, post

3koyhqjgcke2k). NOBUS is a term of art from the intelligence world, short for "nobody but us": a backdoor engineered so that only its intended operator, holding a secret key no one else possesses, can pass through, while everyone else faces what looks like an ordinary, sound lock. The XZ Utils backdoor was NOBUS by construction. It was not a hole anyone could walk through once they knew about it; it was a private entrance keyed to a single party. Roccia made the same point from the other side: "There is a hardcoded certificate used, so that means only the attacker itself can log in to the compromised server, because he already [has] the private key" (Roccia 2024, 29:51).

That property cuts two ways. It bounds the danger: a deployed backdoor would not have been a free-for-all, and a captured attack could not be replayed against the next machine, because the attacker's signature was bound to one host. But it also explains why ordinary network detection would have struggled once it shipped. The backdoor was built to be invisible to everyone but the key-holder. On any input it did not recognize, it behaved like an unmodified server. "Apparently the backdoor reverts back to regular operation if the payload is malformed or the signature from the attacker's key doesn't verify," Valsorda noted, with the consequence for defenders that "unless a bug is found, we can't write a reliable/reusable over-the-network scanner" (Valsorda 2024, post 3kowkezwz6g2q). It did not announce itself on the network: "No evidence of it calling home so far," he reported in the first days (Valsorda 2024, post 3koyopvgoga2n). It left nothing in the records an administrator would consult; in Sam James's flat summary, "Successful exploitation does not generate any log entries" (James 2024b). It did not even slow a victim's machine in any way an operator would notice. Kevin Beaumont, testing a deliberately infected box, found nothing to see: "I had to double check it was actually vulnerable as I wouldn't even see a speed issue. For me, it was a completely transparent backdoor — where `sshd` was running from disk as usual, with the usual file hash and no extra network activity" (Beaumont 2024).

The open-source response, once the alarm was public, worked: distributions reverted, researchers dissected, the thing was contained. But that immune

system never engages with a backdoor that, in operation, is indistinguishable from healthy tissue. There were no exploitation logs to raise an alarm, no network chatter to flag, no performance hit to notice on a production server, and, absent a bug, no reliable over-the-network behavioral signature for a scanner to match. The one cost the backdoor could not fully suppress was a faint delay in the milliseconds before login, on a machine watched closely enough, by someone curious enough, to ask why. That, and not any detection system, is what surfaced it. The capability was designed to defeat exactly the apparatus that later performed so well. It was caught through the one channel the operation had not quite closed.

If the capability was catastrophic, the reach was specific, and the credibility depends on the difference. The backdoor did not threaten "every Linux machine," and saying so is not a hedge; it is the truth, and the more carefully it is told the heavier the real danger lands. Begin with the question a general reader will rightly ask: how could a flaw in a compression library possibly reach the program that guards remote logins? The two have nothing to do with each other. The answer is a quirk of how the major distributions assemble their software. As Przymus and Durieux put it in their 2025 MSR paper, "OpenSSH itself does not depend on `liblzma`, but this dependency is indirectly enforced by Linux distributions that patch OpenSSH to support `systemd`. . . . OpenSSH is patched and linked against `libsystemd` to support `sd_notify`" (Przymus and Durieux 2025, 93). The chain is worth slowing down on. OpenSSH is the standard suite that provides `sshd`, the server that handles remote logins. Several large distributions modify their copy of `sshd` so it can report its status to `systemd`, the program that manages services on most modern Linux systems, and that modification links `sshd` to `libsystemd`, which in turn pulls in `liblzma`. A compression library ended up loaded inside the login server, through a side door no upstream developer had built and few had reason to notice.

Cox drew the boundary that follows from this, and it is the single most disciplining sentence in the record: `liblzma` was "a dependency of OpenSSH `sshd` on Debian, Ubuntu, and Fedora, and other `systemd`-based Linux systems that

patched `sshd` to link `libsystemd`. (Note that this does not include systems like Arch Linux, Gentoo, and NixOS, which do not patch `sshd`.)" (Cox 2024). The danger lived in a particular assembly choice, not in `xz` itself, and distributions that had not made that choice were never exposed through this path at all. Red Hat's own investigation drew the perimeter tighter still. "Over the course of the investigation, we found that this is a kind of dormant malware targeting a very specific set of systems," the company reported: "Specifically the `x86_64` architecture, requiring both `systemd` and `sshd`. It was not targeting BSDs, Android cell phones, wi-fi routers, IoT devices, Raspberry Pis, or anything else. Essentially, it had to be a standard Linux server" (Freire 2024).

The remaining conditions narrow it further, and they come from the discovery side. Freund's email enumerated the filters the malicious code checked before it would arm itself: it was "targeting only `x86-64` linux," it ran only "as part of a debian or RPM package build," the specific packaging machinery of the two big distribution families, and, "due to the working of the injected code," it was "likely" able to "only work on `glibc` based systems," `glibc` being the foundational C library those distributions are built on (Freund 2024b). James's community FAQ turned the filters into a profile of who was actually at risk: a system had to run a distribution using `glibc`, and had to "have versions `5.6.0` or `5.6.1` of `xz` or `liblzma` installed," which was "likely only true if running a rolling-release distro and updating religiously" (James 2024b). That last clause is the crux of the bound. The poisoned versions were so new that, in the window before the catch, they had reached mostly the users who live on the bleeding edge, not the stable servers that make up the bulk of the installed base. And none of it, James was careful to say, was a defect in the components the backdoor strung together: "This is not a fault of `sshd`, `systemd`, or `glibc`, that is just how it was made exploitable" (James 2024b). The attack composed legitimate machinery; it did not break any of it.

So how large is "global," told honestly? The exposed population, had the versions reached stable, would have been the internet-facing Linux servers running the vulnerable configuration, which is a very large number without being

"everything." SSH is the standard way administrators reach servers remotely, and many of the machines that answer it are the machines that keep web infrastructure running. Beaumont offered the one concrete exposure figure in the early record, and it should be read as what it was, a first-week estimate from internet-wide scanning rather than a settled census: "OpenSSH runs on almost 20 million IPs as of today," he wrote, "and is almost 10 times more prevalent than RDP." His own counterfactual was characteristically understated: "Had somebody successfully introduced a widely deployed backdoor, it would have been bad later" (Beaumont 2024).[1]

The danger scaled with something the 2024 internet has and the early internet did not: homogeneity. Clifford Stoll, chasing a different intruder through a more various computing world in the late 1980s, noticed the protection that variety afforded. "If everyone used the same version of the same operating system, a single security hole would let hackers into all the computers," he wrote; the networks he knew ran a dozen incompatible systems, so "no single attack could succeed against all systems," and "diversity in software is a good thing" (Stoll 1989). The world of 2024 is closer to the one Stoll warned about. A handful of `systemd`-based distributions, built on `glibc`, running OpenSSH, sit beneath a large population of internet-facing servers, and a single dependency reached into that population. The reach of the counterfactual is a direct function of that homogeneity: the less varied the substrate, the more one poisoned library is worth.

For a sense of magnitude that is measured rather than asserted, the closest precedent is Heartbleed, the 2014 flaw in another piece of ubiquitous open-source plumbing, the OpenSSL cryptographic library. A peer-reviewed study put numbers on how far a single vulnerability in shared infrastructure can reach: "we can reasonably bound the proportion of vulnerable Alexa Top 1 Million

[1]Beaumont's figure is a same-week count from internet-wide scanning, reported on March 31, 2024; it is carried here as his contemporaneous estimate of how widely OpenSSH is exposed, not as an independent census, and not as a count of vulnerable hosts. The population actually at risk was a subset of internet-facing servers, those running the specific affected configuration described above, not all of the roughly 20 million addresses answering SSH.

HTTPS-enabled websites as lying between 24–55% at the time of the Heartbleed disclosure" (Durumeric et al. 2014, 478). Between a quarter and a half of the most popular secure websites, vulnerable at once. That is the Heartbleed measurement, not an XZ Utils figure, and the distinction is load-bearing: it is offered as the sourced analogue for how large a single shared dependency can loom, not as an estimate of the backdoor's reach, which stays bounded by version, distribution, build path, and release channel. The comparison the responders themselves reached for was bigger still. Goodin reported that Roccia's widely shared infographic visualized "the sprawling extent of the nearly successful endeavor to spread a backdoor with a reach that would have dwarfed the SolarWinds event from 2020" (Goodin 2024), the realized supply-chain attack in which, as the security firm FireEye disclosed at the time, intruders had trojanized "SolarWinds Orion business software updates in order to distribute malware" (FireEye 2020). SolarWinds is what a supply-chain compromise looks like once it lands; the comparison is the responders' attributed assessment of what the XZ Utils backdoor might have exceeded, not the narrator's measurement.

All of which returns to "almost," and to why the word is the wrong frame. Run the counterfactual one more time and notice what is and is not in it. The mechanism was finished: a working, weaponized remote-code-execution capability, demonstrated on a test bench. The targeting was deliberate and in place: the right architecture, the right build path, the right login server. The reach was real and large: a ubiquitous dependency moving toward the stable channels of the world's dominant server platform. Nothing in the design of the open-source supply chain stopped any of it. The system did not detect the backdoor, did not reject it, did not slow its progress toward release. Every layer meant to provide assurance, the code review, the trust placed in a maintainer, the release process, passed the backdoor through, because the backdoor came in through those layers rather than around them.

What stopped it was external to all of that, and contingent. The responders, in the moment, reached for the plainest possible word. James, writing for the

community in real time, set it in bold: "While not scaremongering, it is important to be clear that at this stage, we got lucky, and there may well be other effects of the infected `liblzma`" (James 2024b). Luca Boccassi, a `systemd` and Debian developer, put it with a forward-looking edge on disclosure day: "We were pretty much on the brink of disaster, and got saved because someone's login got slowed down enough that they went 'mmh hang on a sec'. It seems to me we just got very, very lucky here. Will we be so lucky the next time this happens too?" (Corbet 2024, bluca comment, 2024-03-29). Those are their words: the contemporaneous reading of people who had just watched a catastrophe pass close enough to feel.

But "luck" alone is not quite right. What averted the disaster was not a coin landing the right way. It was contingency meeting competence: a chain of coincidences that surfaced a faint performance anomaly, and an engineer with the rare combination of instinct, expertise, and stubbornness to chase half a second of unexplained latency to its source rather than dismiss it. Remove either half and the backdoor keeps moving toward stable deployment. That is why "almost" misleads. It suggests a system with a margin, something that nearly engaged and did. There was no margin. There was an accident, and the right person standing where the accident surfaced. Valsorda, watching the teardown, named both things at once: "This might be the best executed supply chain attack we've seen described in the open, and it's a nightmare scenario: malicious, competent, authorized upstream in a widely used library," and then, "Looks like this got caught by chance. Wonder how long it would have taken otherwise" (Valsorda 2024, post 3kouaom62oi2b). Best executed, and caught by chance. "Authorized upstream" is the whole of it in two words: the attack ran through legitimate trust, not a flaw in the code.

Set beside the other supply-chain disasters of the era, the XZ Utils incident rhymes with each and matches none, and the mismatches are instructive. The closest comparison for ambition is Stuxnet, the sabotage of Iran's nuclear centrifuges that Kim Zetter reconstructed: its authors spent extravagantly on secret flaws, "four zero-day exploits in a single attack," where, by the analysts' account

she relays, "one zero day was bad enough" and a single such bug could fetch "$50,000 or more" on the right market (Zetter 2014). The XZ Utils operation spent almost nothing on bugs. It needed no zero-days at all, because it acquired something more valuable than a stockpile of flaws: the legitimate authority to ship code inside a trusted project. The decisive investment was social, not technical.

For what a ubiquitous open-source vulnerability does when it actually lands rather than being caught, the reference is Log4j, the December 2021 flaw the U.S. Cyber Safety Review Board called, in its register and not the narrator's, the kind of vulnerability that "can create a once-in-a-generation security event" (Cyber Safety Review Board 2022, iv). Within days, the Board recorded, Cloudflare alone "observed 400 exploitation attempts per second" (Cyber Safety Review Board 2022, 4). That is the realized version of the catastrophe the XZ Utils backdoor only approached, and the contrast disciplines the counterfactual in both directions: Log4j was trivially exploitable by anyone who learned of it, while the backdoor was the inverse, gated to a single key-holder and invisible on the wire. "Almost broke the internet" does not mean the catastrophe would have looked like Log4j's stampede. It would have been quiet, exclusive, and operator-only, which is worse for the question of who was behind it, not better.

One more disanalogy has to be stated plainly, because the most alarming comparisons mislead on exactly this point. The fastest-spreading incidents of the era, WannaCry and NotPetya, were self-propagating worms; WannaCry, Nicole Perlroth records, hit "200,000 organizations in 150 countries" within twenty-four hours (Perlroth 2021). The XZ Utils backdoor was not a worm. It did not spread on its own. It sat and waited for its operator to connect, which makes its likely reach smaller than a worm's and its stealth far greater.

That is the shape of what was almost lost, told as honestly as the record allows: a finished, general-purpose remote-code-execution capability, invisible in operation; wired into the login servers of the dominant server platform on Earth through a dependency almost no one knew was there, and moving toward release paths that could have carried it deeper into the installed base; and locked

to the holder of the private key. Not every machine. Not a worm. Not a certainty. But fully built, and stopped by an accident rather than by anything in the system meant to stop it.

Everything else follows from those three facts. The capability had to be built and hidden inside ordinary test files. The reach depended on the path from an upstream release to a running server. And the lock on the door points to the hardest problem of all. A backdoor that only the holder of the private key can open, leaving no logs and no trace, is not merely a technical achievement; it is a design for never being identified. Someone held the private key that fit that door. Who that was, the world still does not know, and the reasons it does not know are themselves part of the story.

Part II

The Long Setup

The cuckoo lays her eggs in other birds' nests. She is a nesting parasite: some other bird will raise her young cuckoos. The survival of cuckoo chicks depends on the ignorance of other species.

— Clifford Stoll, *The Cuckoo's Egg*

Chapter 5

One Person in Finland

Lasse Collin is a software developer in Finland, and for more than a decade he was the person behind `xz`, the compression software whose `liblzma` library is folded quietly into a great deal of the Linux world (Tukaani Project, n.d.-d). Compression is plumbing. It shrinks data so that files move and store more cheaply, and `liblzma` is the piece other programs reach for to do it without thinking about how. Collin did the work that keeps such plumbing sound, the unglamorous upkeep that has no release-day audience: fixing reported bugs, answering mail, reviewing the occasional contribution, cutting a new version when one was due. He did it without pay, in his own time, and for most of those years without much company. The code had a longer history than its current name. The archived LZMA Utils project page described an older package of gzip-like command-line tools, helper scripts, and a small decompression library, and told its users to move to XZ Utils, which supported the legacy `.lzma` format and could emulate the old tools (Tukaani Project, n.d.-a). The legacy branch's own files listed Collin among the authors and directed project contact to his Tukaani address, and by XZ Utils `5.0.0`, released in 2010, the AUTHORS file stated that XZ Utils was developed and maintained by Lasse Collin (Tukaani Project, n.d.-b, n.d.-c, 2010).

He is also, by his own choice, largely absent from the story told about him.

After the backdoor was found, reporters came looking, and Collin mostly did not answer. "I won't reply for now," he wrote on the incident page he keeps at tukaani.org. "I might reconsider after my planned article is out" (Collin, n.d.). The article was to be his own account of what had happened. "I'm writing an article how the backdoor got into the releases and what can be learned from this," he wrote there. "This is taking much longer than I had thought. I've made progress but I don't want to estimate when it's ready" (Collin, n.d.). As of the page's last update, in January 2025, it was still unwritten.

That silence sets the terms for any portrait of him, including this one. What can fairly be said about Collin is what he has said in public and what the documented record shows, and nothing of an inferred inner life beyond it. He did not ask to become the human face of an infrastructure failure, and the urge to read a private state into his reticence is precisely the urge the record does not license. The portrait is spare because the man kept it so, and the spareness is part of the truth.

Set the person aside for a moment and look at the arrangement, because the arrangement is the point. A compression library relied on by most of the machines that carry the internet was maintained, in effect, by one volunteer. There was no team. There was no employer assigning the work, no second maintainer to share the load or to take over when the first stepped back, no security department of the kind that comes standard with commercial software. The whole project rested on a single pair of hands, and on the continued willingness of those hands to keep at it.

Software engineers have a grim shorthand for this exposure. They call it the bus factor: the number of people who would have to be hit by a bus before a project lost the knowledge it needs to survive. The academic study of the attack by Piotr Przymus and Thomas Durieux observes that for projects like xz the figure "becomes alarmingly low" (Przymus and Durieux 2025, 91). For xz, it was one. The same paper draws the contrast that the rest of this chapter circles: "Unlike proprietary software with dedicated security teams and resources, these essential OSS projects often rely on unpaid maintainers who

balance project responsibilities with full-time jobs and personal commitments" (Przymus and Durieux 2025, 91). Karl Fogel, in his long-running handbook on running open-source projects, gives the missing quality a name. He calls it survivability, "the project's ability to continue independently of any individual participant or sponsor . . . the likelihood that the project would continue even if all of its founding members were to move on to other things" (Fogel 2020, 59). A critical library carried by one unpaid volunteer with no backup scores near zero on that measure.

A single maintainer, though, is not only a single point of failure for the work. He is a single point of trust. In a project with no second maintainer there is no one to review the maintainer's own changes, no one positioned to vet a newcomer before relying on him, no separation between the hand that writes a change and the hand that approves it. The usual name for that control is separation of duties: the person who makes a change should not be the only person empowered to approve it. Commercial software treats that separation as elementary: a change is reviewed by someone other than its author, a release is signed by more than one party, and the departure of any single engineer is meant to be survivable by design. A volunteer project of one has none of those checks, not because anyone judged them unnecessary but because there was never a second person to perform them. The same solitude that leaves the maintainer alone with the work also quietly removes every safeguard a second set of eyes would have supplied.

It is tempting to answer that open source long ago solved this, and in places it has. The Linux kernel is maintained by a deep hierarchy of largely paid developers and a web of mutual review built up over more than thirty years: subsystem maintainers who answer to other maintainers, changes that pass through several reviewers before they land, companies and a foundation that fund the work as a matter of corporate interest. But the kernel is the well-resourced exception, not a pattern that reaches down to a project like `xz`, and its defenses, examined later in this book, stop well short of the small, critical dependencies stacked beneath it. `liblzma` had none of them. It had Collin.

The imbalance is the point. On one side, a piece of software woven into the machines the modern economy runs on, the kind of dependency whose quiet failure would ripple through banks and airlines and hospital systems before anyone could name the cause. On the other, one person, unpaid, working in the time he could spare, free to slow down or stop whenever the work stopped being worth it to him. The digital economy presents itself as industrial and automated, a thing of data centers and redundancy and uptime guarantees. A surprising amount of it rests on arrangements exactly this thin, and `xz` was neither the thinnest of them nor the rarest.

There is a reason the solitude had deepened over the years rather than easing. Open source had quietly changed shape. Its early romance was collaborative, many hands on a shared project, contributors who were also users mending what they themselves ran. What replaced it, as open-source code became the infrastructure everyone else builds on, was lonelier. Nadia Eghbal's central observation is that a small number of maintainers, very often just one, now carry libraries used by millions who never contribute anything back, and that the gap between the people who consume the code and the people who tend it has only widened (Eghbal 2020). The bottleneck is no longer whether code gets written; it is whether anyone has the time and the attention to keep it alive. `xz` was a pure specimen of the pattern: an uncountable base of dependents, and one person at the source.

None of this was hidden, and none of it was treated as a scandal. It was simply how open source worked, and to a large extent still does. Eghbal, whose *Working in Public* is the closest thing the field has to an economics of maintenance, locates the difficulty in motivation. "Creation is an intrinsic motivator," she writes; "maintenance usually requires extrinsic motivation" (Eghbal 2020, 89). The energy that builds a useful thing is not the energy that sustains a decade of its upkeep, and upkeep is most of the lifespan. Software, she notes, "once written, is never really finished . . . in order to continue running, software almost always requires some sort of ongoing maintenance" (Eghbal 2020, 117). We are used to thinking of publication as the end of responsibility, "as

when a writer publishes a book, or a pianist finishes a performance," but an open-source maintainer "is expected to maintain the code they published for as long as people use it . . . in some cases, this could be literally decades" (Eghbal 2020, 96). For xz, that was not much of an exaggeration. The developer Jacob Thornton, who cocreated the Bootstrap web framework, has a phrase for the bargain: open-sourcing a project, he says, is "free as in puppy," free to take home and then yours to feed for the rest of its life. "Puppies grow and get old," he told one audience, and "pretty soon . . . you're like, 'Oh my god, so much time is required for me to take care of this thing!' " (Eghbal 2020, 122–23). The joke lands because the cost is real and arrives late, long after the applause for building the thing has died down.

Two features of the work make it especially easy to leave unfunded. The first is that maintenance is noticed only when it stops. A library that compresses data correctly draws no attention whatever; it becomes everyone's problem in the same instant it fails, and never a moment before. The second is the language used to describe it. Eghbal's point is that the very framing of maintenance as a labor of love is what makes the money impossible to discuss: the vocabulary of passion and gift turns a question of resourcing into a question of character, so that asking to be paid can feel like betraying the thing one loves. She calls it the elephant in the room, the sustainability problem everyone can see and no one is placed to raise (Eghbal 2016, 57). The value, meanwhile, runs one way. Companies build on the work, ship products that assume it will keep working, and return, on the whole, neither money nor hands.

The reward structure runs the other way, and the maintainers Eghbal interviewed knew it. She quotes Eric Holscher, of the Read the Docs project, on why he keeps at unpaid work: "It's a labor of love. I could close this project tomorrow . . . but I've been doing it for 5 years and I don't want to see that happen" (Eghbal 2016, 57). The phrase is warm, and it is also a bind, because love is the one thing that cannot be invoiced. The demands, meanwhile, arrive regardless. Another developer, Daniel Roy Greenfeld, told her he gets "regular demands for unpaid work . . . by healthy high profit companies large and small," and that if

he does not respond quickly enough, or declines a poor contribution, "I/we get labeled a jerk" (Eghbal 2016, 82). And the exit, when it comes, is rarely clean. Fogel describes the slide from the inside: a swamped maintainer "usually doesn't notice it right away. It happens by slow degrees," the project simply does not "hear much from him for a while," then there is a guilty burst of catching up, then quiet again, but "there's rarely an unsolicited formal resignation" (Fogel 2020, 149). Fogel's account assumes a community standing ready to notice the gap and recruit a replacement. The hazard, for a project of one, is what happens when the replacement arrives unbidden, from outside.

Against that arrangement, Collin's own words read less like confession than like description. In June 2022, in a message to the project's development mailing list, he wrote: "I haven't lost interest but my ability to care has been fairly limited mostly due to longterm mental health issues but also due to some other things. Recently I've worked off-list a bit with Jia Tan on XZ Utils and perhaps he will have a bigger role in the future, we'll see. It's also good to keep in mind that this is an unpaid hobby project" (Collin 2022).

It is the last sentences that the rest of this book turns on, and they deserve to be read for what they say rather than for what hindsight wants them to say. Collin named a condition, not a plan. He was a single unpaid volunteer who stated that his ability to care had been limited, and who mentioned, almost in passing, that a helpful newcomer might come to do more. He could not have known what that aside would come to mean. To read it as the instant burnout opened the door is to draw a causal line the record does not support and the man himself never drew. The honest reading is narrower and more unsettling: the conditions of unsupported solo maintenance were on open display, in the maintainer's own words, on a public list, nearly two years before anyone had reason to look. The "Jia Tan" named in the message is the subject of the next chapter; what matters here is the condition the message documents, not the figure it names.

What keeps the condition from looking like Collin's personal failing is that it is not Collin's alone. The engineer who caught the backdoor, Andres Freund,

is himself a long-tenured open-source maintainer, and asked afterward about the people on the other side of these projects, he reached for his own experience without prompting. "I know the feeling," he said, of being "overwhelmed, you can't keep up, and people are . . . chiding you for . . . not doing enough in their eyes, not jumping through all the hoops that they want you to," and then, of the whole picture, "very familiar" (Freund 2024c, 01:02:01). Even ordinary friction, nothing remotely like an organized campaign, "still felt like failure," he added (Freund 2024c, 01:02:48).

The pattern surfaces wherever maintainers are asked. A 2024 study of open-source maintenance commissioned by the Sovereign Tech Fund, Germany's public financier of critical software, drew on dozens of interviews; that a national government had set up a fund to pay for open-source upkeep at all was itself a sign the problem had outgrown the volunteers. One contributor, a research scientist the report calls Johann, described how a job and a family crowd out unpaid work: "very often this hobby, open-source projects, are the first thing to [go]" (Ellis and Bollampalli 2024, 20). Another, a longtime contributor to the Rust language the report calls Alex, described the slower erosion and the users who supply it: "the constant weight of that can become very burdensome, especially as it streams in over the span of a decade… Some folks are super rude. They're like, 'why have you not done this? This is ridiculous. How can anyone not do this?' So seeing that over time can be very crushing…" (Ellis and Bollampalli 2024, 23). The thread running through these accounts is not drama but accumulation: a weight that streams in over a decade and is never quite set down. And none of it is anyone's job. The work is done after the paid work is finished, in the hours that would otherwise be rest, which is why a busy stretch at the office or a hard season at home registers at once as a project falling behind.

The surveys put numbers under the anecdotes. In Tidelift's 2023 survey of open-source maintainers, the most common answer to how a project is staffed, by more than two to one, was "I am a solo maintainer" (Tidelift 2023, 27): 44% had no co-maintainer at all. A majority, 58%, had either quit a project or come to

the edge of quitting one, with burnout among the most common reasons they named (Tidelift 2023, 29), and the share reporting that maintenance added to their personal stress had climbed to 54%, with loneliness up to 42% (Tidelift 2023, 26). The figures come from a funding vendor's self-selected survey and are best read as directional, but the direction is consistent, and an earlier Linux Foundation and Harvard survey of contributors found the same departures for the same reasons: people left when contributing had become "unattractive," when "their efforts were met with unreasonable demands from end users, attacks from others, as well as 'negative personal and professional outcomes' " (Nagle, Wheeler, et al. 2020, 52). That survey was not idle curiosity. It had grown out of an effort, after an earlier scare over unmaintained infrastructure, to take a census of the projects the world most depended on and least supported (Nagle, Wheeler, et al. 2020). The numbers have voices behind them. "I feel like not having enough time to implement all the ideas I have is leading me to getting burned out," one maintainer wrote in the same Tidelift survey (Tidelift 2023, 38). Collin, on this evidence, was not an aberration. His situation sat close to a pattern the data repeatedly showed: one person doing ordinary work under ordinary conditions, in a system that had quietly made those conditions the norm.

Two kinds of pressure are worth holding apart here. Everything these maintainers describe is ordinary pressure: diffuse, unorganized, the friction of too many users and too little time, the occasional rude stranger. It is the weather of unpaid maintenance, and most maintainers weather it. The pressure that would later close around Collin was not ordinary, and it did not arrive by accident. That part of the story belongs to a later chapter; what matters here is that it worked at all, and it worked because the ordinary kind had already worn the ground down.

The culture had begun, dimly, to notice. In November 2023, four months before the disclosure, the Kernel Summit, an annual gathering of Linux's core developers held within the Linux Plumbers Conference, set aside a session on maintainer stress and burnout and brought in an outside psychologist, Gloria Chance, to lead it. Putting "psychological safety" in front of kernel developers

was itself a small landmark: the community was beginning to name the suffering rather than treat it as the weather. The session reached for plain words for what it meant. It defined being overwhelmed as the point at which "intensity of your feelings outmatches your ability to manage them," and cataloged how that "looks like," from "anxiety, stress, impatience" to "fatigue" and feeling "helpless, or hopeless" (Chance 2023): the lay vocabulary the kernel world was now using, in front of its own developers, for a state it had long filed under the cost of doing the work. The remedy on offer, though, kept pointing back at the person. "Resilience is a muscle," one slide ran, quoting the musician Pink: "Flex it enough and it will take less effort to get over the emotional punches each time" (Chance 2023). Another held that resilience lets us "not only adapt ourselves to stress and disappointments" but "grow the insight to avoid actions that might lead us to face such situations" (Chance 2023), as though the maintainer's task were to steer clear of the circumstances that overwhelmed him. Naming the strain was overdue, and the people in that room meant well. But a muscle is something the individual trains in himself, and the thing actually overloaded was not the individual. It was the arrangement: one unpaid person under a load that several paid ones would have struggled to carry. The reflex, confronted with structural strain, was to improve the resilience of the person rather than the resourcing of the work, and the book takes that mismatch up again in its later chapters.

When the backdoor became public on March 29, 2024, the strain stopped being a conference topic and became, briefly, the story. In the community FAQ that the Gentoo developer Sam James assembled in the first hours, readers were told that "Lasse regularly has internet breaks and was on one of these as this all kicked off," and were asked to "be patient with him as he gets up to speed and takes time to analyse the situation carefully" (James 2024b). The maintainer of the compromised project was away from the project when the alarm sounded, then returned to an emergency already underway. The reckoning that followed was real but unfocused. Evan Boehs, whose timeline became the most-read public reconstruction of the attack, wrote on the day of disclosure that "in the fallout, there is much to learn about mental health in open source" (Boehs 2024),

and set beside it a post that had gone up that day from a developer who writes as Glyph: "I really hope that this causes an industry-wide reckoning with the common practice of letting your entire goddamn product rest on the shoulders of one overworked person having a slow mental health crisis without financially or operationally supporting them whatsoever" (Glyph 2024). Glyph's phrasing is one observer's reading, offered in heat on the day the news broke, and the book does not adopt its diagnosis of any person. But the structural half of the sentence, a product resting on one unsupported person, was exactly right, and it was the half the industry, for a few weeks, seemed prepared to hear.

Not everyone framed it as a welfare problem. The security researcher Michał Zalewski, writing the day after disclosure, described the same long arc in cooler terms: "After a while, the maintainer just isn't all that into it anymore; they are eager to pass the baton to anyone with a pulse and some modicum of skill" (Zalewski 2024). As a claim about any one person it would be unfair, and Zalewski does not make it one. As a description of a pattern, the slow disengagement of the long-tenured solo maintainer, it names with some precision the condition that makes a handover to a helpful stranger feel not like a risk but like a relief.

For all the energy, the reckoning mostly produced talk. Attention spiked in the weeks after the disclosure and then receded, the way attention does. The one response that would have changed the arrangement, paying the people who hold the infrastructure up, was the hardest to sustain and is the subject of a later chapter. What the episode clarified, at least, was the structure. A critical piece of the world's software had been resting on one unsupported person, in plain sight, for years, and the millions who depended on it had not been looking at the person at all.

Put the pieces together and the shape of the vulnerability is plain, and it is not, in the first instance, a technical one. The people sorting through the wreckage saw it at once. On the second day of the disclosure, a Debian contributor named Pierre Ynard, reading the bug thread, set it down in a sentence: "it is said that the whole situation started with the lack of resources from the upstream

maintainer to maintain the project . . . which again, we can suspect at this point, was exploited with the same modus operandi to get a compromise vector coopted in, in the form of a new maintainer" (Hess 2024, msg #40). The hedges are honest. "It is said," "we can suspect": a practitioner reasoning in real time, on the strength of two days. The reading was early, hedged, and close to the shape the record now supports. The opening was the want of resources; the vector was the relief of help. The upstream maintainer he means is the person who runs the original project, the source from which the Linux distributions take their copies, which is to say Collin.

There is a sharper way to put what was exposed. In a project held up by one person, that person's judgment is the security. Users trusted `xz` because they trusted Collin: his care, his caution, his sense of what did and did not belong in the code. There was little else in the way of mechanism. No committee reviewed his decisions, no second signature countersigned a release, no process existed apart from him. That is an efficient way to run a project on goodwill, and it leaves the project exposed, because winning the confidence of one unsupported person is far easier than defeating a system of checks that was never built. The attack surface, in the end, was not the code. It was the trust placed in one man, and the conditions that had left him alone to hold it.

This is the argument the chapter has been assembling, and it has to be stated with care, because the careless version is both seductive and wrong. The careless version is that a burned-out maintainer handed his project to an attacker. What the record supports is quieter and structural. A critical library was maintained by one unpaid volunteer with no backup, inside a culture that treated unsupported solo maintenance as normal; the work had no natural end and the support had never arrived; and into that arrangement, help was always going to look welcome. In such a project, accepting help could look less like risk than relief. None of that made the compromise inevitable, and none of it is a verdict on Lasse Collin, who did, for years and without pay, work that much of the digital world depended on and almost none of it funded. What the conditions did was make the compromise possible. The vulnerability was structural before it

was technical, and it was in place long before a line of the backdoor was written. What it needed next was someone willing to be the help. He had already begun to appear.

Chapter 6

Jia Tan Appears

The first trace is so ordinary that it takes an effort of will to find it sinister. On October 29, 2021, a small contribution arrived on xz-devel, the development mailing list where the few people who followed `xz` discussed its upkeep. It added an `.editorconfig` file, a short text file that tells code editors how to handle whitespace so that contributors do not trip over one another's tabs and spaces (Cox 2024; Kaspersky GReAT 2024a). It was housekeeping, the kind of tidy, unremarkable patch a conscientious newcomer sends to a project he means to stay with. The sender signed it "Jia Tan." Two and a half years later, after the backdoor was found, that patch would be read as the opening move of a long and patient supply-chain operation. On the day it arrived, it was a config file. No one had any reason to look twice, and no one did.

The difficulty is to hold two facts together. The operation was malicious, and much of the work that made it possible looked genuinely useful at the time. Almost everything "Jia Tan" did across the first phase of the operation was indistinguishable from the work of an ordinary volunteer, and it was indistinguishable not because the disguise was elaborate but because, for most of that time, there was nothing to disguise. The work was real. The bug fixes fixed bugs, the documentation documented, the patches improved the software. To reconstruct the cultivation honestly is to resist reading the finished operation

backward into its first, blameless-looking steps, and to ask instead what a reasonable maintainer, or a reasonable observer, could actually have seen at the time. The answer, again and again, is: a helpful contributor. Who was in fact behind the name remains unknown, and this book does not pretend otherwise; the attribution is taken up, and left open, in a later chapter. For now the name is only a name, kept in the quotation marks that mark it as the alias of a party still unidentified.

The footprints in this opening stretch were not all in `xz`. Within days of the mailing-list patch, in early November 2021, the same contributor, working on GitHub under the account `JiaT75`, opened a change to `libarchive`, a separate and widely used library for reading and writing archive files. Attached to an innocuous improvement of an error message, the change quietly replaced a call to `safe_fprintf`, a function that prints text while neutralizing dangerous control characters, with the plain `fprintf`, which does not. It was merged with no substantive discussion on November 15, 2021, and reverted on March 29, 2024, after the XZ Utils disclosure (Boehs 2024; emaste 2024; Goodin 2024; JiaT75 2021). Read in 2024, with the rest of the operation visible, the swap looks like reconnaissance: a small, deniable weakening of a safety boundary, dropped into a busy project to test the water. Read in 2021 it looks like a minor patch to an error message. Dan Goodin, recounting it later for *Ars Technica*, put the whole problem in a few words: "No one noticed at the time" (Goodin 2024). The security foundations that afterward dissected the campaign would describe the move as a probe "to see who would notice" (Bender Ginn and Arasaratnam 2024), but that reading was available only on the far side of the disclosure, and the change was not even in `xz`. At the time it was one more small contribution from one more helpful stranger.

The first change to `xz` itself followed a few months later. In late January 2022 "Jia Tan" submitted a small hardening fix to the library's compression code, the routine kind that adds a missing safety check, and Lasse Collin merged it into the repository on February 6, 2022, recording the work under the contributor's name (Cox 2024). It was, again, useful and unremarkable. A small fix

that guards existing code against bad input is exactly the sort of contribution a maintainer is glad to receive and has little reason to scrutinize. From there "Jia Tan" settled into the rhythm of a regular contributor.

What followed was not a sprint toward a payload but something closer to its opposite: a year and a half of steady, real work. Russ Cox, who later reconstructed the operation's social-engineering campaign commit by commit, compressed the whole of it into a sentence. "Over a period of over two years," he wrote, "an attacker using the name 'Jia Tan' worked as a diligent, effective contributor to the xz compression library, eventually being granted commit access and maintainership" (Cox 2024). The point is not simply that Cox calls the contributor an attacker. It is how he describes the work: diligent, effective. The operation's first phase was not a deception laid over the contribution; the contribution was the deception. The academic reconstruction of the attack, by Piotr Przymus and Thomas Durieux, found something sharper still when it sorted the contributions by type, because the bulk of the work was not even code. "[T]he main contribution of [the Attacker] during those years," they write, "was not in the source code but instead in the documentation and translation" (Przymus and Durieux 2025, 96). Those are the lowest-scrutiny contributions a project receives. A fixed typo or a translated message string is welcome, easy to accept, and almost never read with suspicion, yet each one is another commit under the contributor's name, another increment of standing. The credibility accrued without ever inviting the kind of review that code attracts.

The privileges followed the credibility. Some form of write access to the repository, the permission to change the code directly rather than submit changes for someone else's approval, arrived in 2022, before the first visible exercise of merge authority the following January. Przymus and Durieux note that this access was never abused in the obvious way: the operator did not push malicious code, and did not rewrite the project's history to cover the operation's tracks (Przymus and Durieux 2025, 94). The `git` repository, the canonical, public, version-by-version record of the software's development, stayed clean throughout. Whatever the payload would turn out to be, it would

not travel through the one place everyone could watch. That distinction matters a great deal later; here it is simply one more reason the cultivation drew no fire. The identity had reach as well as depth. Kaspersky's analysts, picking through the account afterward, counted more than five hundred contributions under `JiaT75` across several unrelated open-source projects going back to early 2022, so that the `xz` work sat inside a wider pattern of ordinary participation, a plausible open-source résumé built in public; by the same account the account itself had been registered in January 2021 and left dormant for months before the first patch (Kaspersky GReAT 2024a). These are a vendor's retrospective findings, not contemporaneous alarms, and that is exactly the point. Assembled after the fact, they trace a deliberate, patient construction of legitimacy. Encountered one patch at a time, they were just a productive newcomer.

A security company captured the ordinariness better than any sympathizer could. "Almost two years ago," Akamai's researchers wrote in the days after the disclosure, "a developer under the name of Jia Tan joined the project and started opening pull requests for various bug fixes or improvements. So far, nothing is out of the ordinary; this is how things work in the open-source world" (Akamai Security Intelligence Group 2024). The sentence is an admission against interest, a security vendor conceding that the attack's opening was indistinguishable from the normal functioning of the thing it attacked. There was no anomaly to detect, because the behavior was the norm.

That norm deserves to be stated plainly, because the operation exploited it precisely, and it is less a flaw than a founding commitment. Open-source software is built by people who mostly do not know one another and mostly never will, and its culture made a virtue of exactly that. Karl Fogel, whose handbook on running free-software projects is close to a standard text, states the welcoming principle as an ideal to aim for: "The goal is to make every user realize that there is no innate difference between himself and the people who work on the project — that it's a question of how much time and effort one puts in, not a question of who one is" (Fogel 2020, 143). Who one is should

not matter; what one contributes should. It is a generous idea, and it is also, read from the far side of XZ Utils, the precise gap the operator stepped through. Fogel makes the same point from the other direction with a story he plainly means as encouragement. A project once had a contributor whose work was good enough that no one thought to wonder about him, until it emerged that he was thirteen years old. "Because that kid didn't write like a thirteen-year-old," Fogel writes, "no one knew that's what he was" (Fogel 2020, 95). Online, a contributor is only the quality and consistency of his work; everything else (age, location, employer, intention) is invisible unless he chooses to reveal it. Fogel offers this as liberation, and for a gifted teenager it is. Run against a patient operator it is the whole problem in a line: a competent, consistent contributor is indistinguishable from a competent, consistent attacker, because competence and consistency are nearly all the medium transmits.

None of this made pseudonymity itself a warning sign, because pseudonymity was unremarkable. A 2020 survey of free- and open-source contributors found that most, around 90%, worked under their real name or a stable handle linked to it, but that a small and persistent minority, roughly one in sixteen, contributed under a screen name connected to no real identity anywhere, and the culture accommodated them without friction (Nagle, Wheeler, et al. 2020, 75). The same survey found that for nearly two-thirds of their contributions, respondents had known no one on the project before they joined (Nagle, Wheeler, et al. 2020, 53). Trust, in other words, was routinely extended to strangers as a condition of the work getting done at all. A persistent, useful, pseudonymous contributor was not an anomaly to be explained. It was a recognized and accommodated part of the culture that kept open source alive. The habit was old, and within the field entirely respectable. Cox, writing in 2019 about the hazards of depending on other people's code, well before anyone had reason to connect the thought to `xz`, described why programmers accept the arrangement so readily: "Because it's easy, it seems to work, everyone else is doing it, and, most importantly, it seems like a natural continuation of age-old established practice. But there

are important differences that are being ignored" (Cox 2019, 37). Accepting a helpful stranger's code was not a lapse peculiar to Collin or to xz. It was the default behavior of the entire ecosystem, the thing that let a volunteer keep a library alive without auditing the life story of everyone who sent a fix.

Trust, once extended, compounds in a particular way: a contributor earns it not in a leap but by accretion, one small correct change after another, until a standing has been built that no one quite remembers granting. Linus Torvalds, recalling for the writer Glyn Moody how the early Linux kernel came to rely on its contributors, described the mechanism without a trace of alarm: "They started out so small, that I never got the feeling that, hey, how dare they impose on my system. Instead, I just said, OK, that's right, obviously correct, and so in it went. . . . And again they grew gradually, and so at no point I felt, hey, I'm losing control" (Moody 2001). That is how legitimate maintainership has always been earned. It is also the path the operator walked: each change obviously correct, the trust gradual, the loss of control invisible because at no single moment was any control visibly lost.

All of which is why the most important corrective comes from the one person with the least reason to be generous about it. After the backdoor was exposed, Collin went back through the commits the internet had begun to flag as suspicious and annotated them in public, one by one. What he produced is not a confession but a debunking, and it is the discipline the reconstruction has to borrow. Many of the changes that looked sinister in hindsight were, Collin pointed out, simply good. Of one substantial reworking of the library's decoder he wrote: "These are good and were created at my request. They are big but it's hard to split them into smaller pieces. The original versions of these are from 2023-04-24. I made small edits but it was agreed that I would commit these in his name" (Collin 2024b). Reviewing with full knowledge of the betrayal, the maintainer still rated the work as good, and as his own idea. The labor was real. That is not a mitigating detail; it is the mechanism. The helpfulness was not cover for the operation. For the better part of two years it was the operation.

Collin's review punctures the hindsight reading at the level of detail, too.

Among the commits the public treated as tells was one whose recorded timezone did not match Collin's own, a discrepancy that looked, after the fact, like evidence of an impostor working from another part of the world. Every `git` commit is stamped with a timezone, and once the operation was known, those stamps were combed for patterns; that analysis belongs to a later chapter and stays genuinely unresolved there. But this particular clue dissolved on contact with the person involved. "This is one of the supposedly suspicous commits due to the timezone," Collin wrote, preserving his own typo. "In reality Jia put it in my name by common agreement because I had done a significant portion of it" (Collin 2024b). The commit was Collin's own work, attributed to him by an informal arrangement in which the two of them moved changes between their names by handshake. Read forward, it is a clue. Read accurately, it is two collaborators in a tiny project not standing on ceremony. The lesson is not that the suspicious signals were all innocent, but that in real time they were unreadable, and that a great many of them, examined closely, are innocent still. The academic study reaches the same caution by a colder route. The operation, Przymus and Durieux conclude, "shows how routine project involvement can be methodically exploited, underscoring the difficulty of distinguishing between well-intentioned and malicious contributors" (Przymus and Durieux 2025, 96). The difficulty is not incidental to the finding. It is the finding.

What the patience bought, slowly, was authority. The currency of an open-source project is not money but standing, and standing converts, by degrees, into power over the code. The conversion is visible in the record. On January 7, 2023, "Jia Tan" visibly merged a change for the first time, the act of accepting a contribution into the official code rather than merely offering one, which is the practical mark of having been trusted with the keys (Boehs 2024). By the middle of that year the trust ran in the other direction as well: a performance optimization that Collin himself had written was committed to the repository under "Jia Tan's" hand on June 27, 2023. The maintainer was now routing his own work through the newcomer. Groundwork for the same optimization had been laid a few days earlier by yet another contributor, one calling himself "Hans Jansen,"

a name that would recur, and that the book returns to when it takes up the question of who was really at the keyboard (Boehs 2024).

None of this required force, and that is the unsettling part. It required persistence. Collin's review, again, shows the texture of it. Of one set of additions to the library's programming interface he wrote, "I found and still find these useless additions to the API. He kept insisting on them so eventually I gave in. There's nothing technically wrong, I just think they don't improve readability" (Collin 2024b). He kept insisting, so eventually I gave in. It is worth sitting with how ordinary that is: a maintainer with limited backup, faced with a contributor who is friendly, productive, and simply will not let a point drop, eventually accepts a harmless change he still does not think improves the software. The same pattern recurs in the record more than once. There is no exploit in it, no breach, nothing a security tool could flag. There is only sustained pressure, applied through ordinary collaboration, and a project structured so that pressure on one person could be enough. And that pressure did not come only from inside the collaboration. While "Jia Tan" pressed Collin patiently across the code review, a separate chorus of insistent strangers had begun pressing him from outside, on the same mailing list, toward the same end. That campaign is the subject of the next chapter; what matters here is that the two halves were complementary, the helpful insider and the demanding outsiders converging on one maintainer with limited backup.

Step back from the detail and the strategic shape is plain. Eighteen months of genuinely useful work can be a more effective way into a critical project than a technical exploit, because the project's defenses are built to catch bad code, while the work that earned trust was mostly useful, ordinary work. The defenses assume that bad faith arrives looking like bad faith. This arrived looking like help. Set against the ordinary run of attacks on open source, the patience is almost eccentric. The largest catalog of these incidents assembled before XZ Utils, a 2020 review by Marc Ohm and colleagues of 174 malicious packages, found that most of them, 61%, worked by typosquatting: registering a package under a name a keystroke away from a popular one, then waiting for a developer

to fumble the spelling and install the imposter by mistake (Ohm et al. 2020, 12). Typosquatting is a numbers game, cheap and crude and instantly disposable. The XZ Utils operation was its opposite in every dimension. It did not impersonate a trusted project; it became one, from the inside, over years, by doing the real work. On the distribution of supply-chain attacks it sits at the far, rare, expensive end, the patient exception in a field dominated by smash-and-grab. The patterns that would later be offered as warning signs, the gradually escalating involvement, the slow accumulation of privilege, were patterns only in retrospect. Drawn forward into a checklist after the disclosure, they describe the operation exactly. Applied beforehand, they describe almost every dedicated volunteer open source has ever had.

What the patience also bought was cover for the person underneath it, and that cover has held. The care taken over the long cultivation was itself a kind of tradecraft, the discipline to spend the better part of two years being useful for the sake of a payoff that might never come. Molly, a systems administrator at the Electronic Frontier Foundation (EFF) who goes by a single name, named both halves of that discipline in one breath, speaking to *The Intercept* after the disclosure: "The care taken to hide the exploits in binary test files as well as the sheer time taken to gain a reputation in the open-source project to later exploit it are abnormally sophisticated" (Mazurov 2024). The reputation-building and the concealment were the same project, pursued with the same patience. The concealment is a later chapter's subject. The reputation-building is this one's, and at the time of writing the figure who did it remains a name in quotation marks and nothing else. Who "Jia Tan" was, whether one person or several, working for whom, is taken up where the evidence for it lives, in the chapter on the hunt, and is not resolved there, because it has not been resolved anywhere.

The cultivation worked because it asked the system to do the one thing it was built to do, accept good work from a stranger, and gave it good work to accept. By the start of 2023 the stranger held the keys. There had been no break-in. A key had been handed over, one small, reasonable concession at a time, inside a project where help was scarce and the pressure kept increasing. But

the handing over was not yet finished, and the patience inside the collaboration had a louder counterpart outside it. Before the keys changed hands for good, a small crowd of strangers would spend eight weeks telling Collin, in escalating tones, that he could not be trusted to keep them.

Chapter 7

The Sock Puppets

The turn arrived as a reply to a patch. In April 2022 a contribution from "Jia Tan" sat on xz-devel, waiting on the attention of the one person who could merge it. On April 28 an account no one on the list had seen before posted its first message into that thread.[1] The sender signed himself "Jigar Kumar," and he dispensed with introductions. "Patches spend years on this mailing list," he wrote. "5.2.0 release was 7 years ago. There is no reason to think anything is coming soon" (xz-devel mailing list 2022, msg00557).

The complaint, as a complaint, had a defect that took two years to surface. The slowness was real; the release the message mocked was genuinely seven years old, and a frustrated stranger is part of the weather of any open-source mailing list. But the specific neglect on display did not exist. By the time "Jigar Kumar" appeared, Lasse Collin had already merged four of "Jia Tan's" patches, each recorded in the project's history with a "Thanks to Jia Tan" in the commit message (Cox 2024). The maintainer being accused of ignoring the newcomer's work was, by the visible record, accepting it steadily. Evan Boehs, whose timeline became the reconstruction much of the later reporting leaned on, marks this as the moment the operation acquired a second voice: a patch arrives from

[1] The archived message's own date header reads April 28, 2022, the form cited here; Russ Cox's timeline of the operation dates the same message April 22 (Cox 2024). The book cites the primary archive's dates for the campaign's correspondence throughout.

"Jia Tan," and a new persona enters to press for it (Boehs 2024). Dan Goodin's account in *Ars Technica*, written for readers a long way from any mailing list, compresses the sequence to its shape: a patch is submitted, and "almost immediately" a "never-before-seen participant" joins the list to argue that the longtime maintainer is not keeping up (Goodin 2024).

Up to this point the operation's story has been a story about code. From here it is a story about people. Accounts of the kind that began to gather on xz-devel in the spring of 2022 have an old name online: sock puppets, identities invented and worked by a hidden hand, which speak, like the toy, only because there is an arm inside. The chapter takes its title from the term, so the ground rule comes first. What the record documents is behavior: what the accounts wrote, when they wrote it, to whom, and what became of them afterward. That the accounts were coordinated, that they were created for the purpose, that they belonged to the operation at all, is inference, drawn after the fact by named observers whose judgments arrive, as will be seen, with their own hedges attached. The reconstruction that follows states the inference where the evidence invites it and keeps the line between the two in view throughout.

The second voice arrived three weeks later, on a front of its own. Collin also maintained XZ for Java, a separate implementation of the same compression format written in the Java programming language, and on May 19, 2022, an account calling itself "Dennis Ens" opened a fresh thread about it: "Is XZ for Java still maintained? I asked a question here a week ago and have not heard back" (xz-devel mailing list 2022, msg00562). On its face it was the most ordinary message on the list, a user asking after a quiet project, listing the features he was waiting on, wondering whether to carry his own changes locally in the meantime. That ordinariness is worth pausing over, because nothing about the second front resembled the first. One voice was abrasive, the other patient and reasonable; one pressed on the C library, the other on the Java one; and no visible thread connected them except the thing they would, over the following month, converge on.

The convergence came on June 7, when "Jigar Kumar" stepped into "Den-

nis Ens's" Java thread with the campaign's goal stated aloud. "Progress will not happen until there is new maintainer," he wrote. "XZ for C has sparse commit log too. Dennis you are better off waiting until new maintainer happens or fork yourself. Submitting patches here has no purpose these days. The current maintainer lost interest or doesn't care to maintain anymore. It is sad to see for a repo like this" (xz-devel mailing list 2022, msg00566). To fork a project is to copy its code and carry on development independently, the open-source world's standing threat of last resort. The entire campaign is in that message in miniature. The demand is named: a new maintainer. The target is the maintainer's character rather than the code: he "lost interest or doesn't care." And one stranger now addresses another by first name, as though a community of the dissatisfied had formed, a chorus performing consensus in the open for an audience of one.

Collin answered the next day. His reply is the message an earlier chapter read closely, the one in which a maintainer under pressure disclosed that his ability to care had been limited by "longterm mental health issues," reminded the list that the work was an unpaid hobby project, and mentioned that he had been working off-list with "Jia Tan," who might come to have a bigger role (Collin 2022). What belongs to this chapter is the reply's closing thought: "Perhaps the moment after the `5.4.0` release would be a convenient moment to make changes in the list of project maintainer(s)" (Collin 2022). Six weeks of manufactured impatience had produced something no technical exploit could have: the maintainer himself, on the public record, floating a handover and naming the moment for it. Nothing had been taken. Something had been offered.

The chorus answered the concession by demanding it faster. "With your current rate, I very doubt to see `5.4.0` release this year," "Jigar Kumar" wrote on June 14. "The only progress since april has been small changes to test code. You ignore the many patches bit rotting away on this mailing list. Right now you choke your repo. Why wait until `5.4.0` to change maintainer? Why delay what your repo needs?" (xz-devel mailing list 2022, msg00568). As a user's complaint the message is strange. It names no defect, advocates no patch, asks

nothing of the software at all. Its one subject is the calendar of the maintainer change, and its one effect, on a maintainer who had conceded the principle six days earlier, is to make the concession feel overdue. The spelling here and throughout is the messages' own, preserved verbatim; the errors will matter later.

A week later the polite voice returned, and the campaign showed what it was. On June 21, "Dennis Ens" replied to Collin's disclosure with a message framed as concern: Collin needed to recognize his limits, the community wanted more than the two projects were getting, and one of them should be handed off (xz-devel mailing list 2022, msg00569). It was the same demand dressed as solicitude. Thirteen days earlier, on this same list, under this same pressure, Collin had disclosed a private difficulty in one sentence. Now it was being returned to him as an argument for the handover. What that message did to the man who received it is not in the record, and this book does not guess. What the message did on the record is plain, and it is the strongest evidence in the correspondence that the pressure was engineered rather than ordinary. Genuine impatience is careless and diffuse; it wants the software fixed. This wanted what the other voice wanted, reached for the most personal material available on the list, and pressed on it in service of that single end. The cruelty, read soberly, is not color. It is the tell.

The following day the campaign made its direction unmistakable. "Is there any progress on this?" "Jigar Kumar" asked back in the patch thread where he had first appeared. "Jia I see you have recent commits. Why can't you commit this yourself?" (xz-devel mailing list 2022, msg00570). It is the least dramatic message in the sequence and the most clarifying. The grievance was never speed. By late June the chorus was no longer asking Collin to work faster; it was asking, in the open, why "Jia Tan" did not yet hold the authority himself. The helpful insider of the previous chapter and the impatient outsiders of this one had converged on a single point, and the point was the keys.

"It worked." The verdict is Russ Cox's, in the commit-by-commit reconstruction this book has leaned on, and he delivers it flat: "It seems likely that

they were fakes created to push Lasse to give Jia more control. It worked. Over the next few months, Jia started replying to threads on xz-devel authoritatively about the upcoming 5 . 4 . 0 release" (Cox 2024). By autumn the newcomer was speaking for the project's next release; the first visible exercise of merge authority, on January 7, 2023, the previous chapter has already shown. Outside analysts read the campaign the same way. Thomas Roccia, a security researcher at Microsoft who assembled one of the early public reconstructions, described the extra personas as existing to press the maintainer "to change the maintenance itself to Jia" (Freund and Roccia 2024). Akamai's researchers called the maneuver "an interesting form of social engineering": fake accounts sending "myriad feature requests and complaints about bugs to pressure the original maintainer" (Akamai Security Intelligence Group 2024). Their sentence runs a half step past the record in one respect, crediting the campaign with "causing the need to add another maintainer"; causation inside one person's decisions is exactly what a documentary record cannot show, and what the sourced sequence supports is narrower: pressure, then a concession, then an authority that grew. The distinction is small and load-bearing, and it opens the question the rest of the chapter has to answer: what, precisely, marks these accounts as fake, and what merely reads that way in hindsight?

Start with what is documented. The accounts left no discoverable public lives. "Jigar Kumar is never seen again" after the pressure ends, Boehs records, and "Dennis Ens," writing from "a similar name+number formatted email," is likewise "never seen outside of xz discussion," with no associated accounts ever discovered for either (Boehs 2024). Both addresses follow one template, a name and a number at a free webmail service. The reporters at *The Intercept* who went looking for the people behind the names produced the cleanest portrait of what they found, which was nothing: "The users involved in the complaints seemed to materialize from nowhere — posting their messages from what appear to be recently created Proton Mail accounts, then disappearing. Their entire online presence is related to these brief interactions on the mailing list dedicated to XZ; their only recorded interest is in quickly ushering along updates to the

software" (Mazurov 2024). The hedges in that portrait, "seemed to," "appear to be," are the reporters' own, and they are the right ones. The absence of a footprint is documented; what the absence means is not.

The same reporting contributes the record's one genuinely forensic datum. Where the chorus wrote from Proton Mail, an encrypted email service that generates a cryptographic key when an address is set up, the keys associated with the accounts "were created on the same day, or mere days before, the users' first posts to the email group" (Mazurov 2024). Identities minted on the eve of their first complaint are what a purpose-built chorus would look like. The reporting attaches the limit in the same parenthesis, and the book keeps it attached: users can regenerate keys, so the addresses may be older than the keys they now carry. Suggestive, not dispositive. The template itself, though, proved legible the moment anyone had reason to look. Two years on, when accounts of the same shape turned up in Debian's bug tracker hurrying the backdoored release toward the distribution, a push a later chapter reconstructs, the veteran Debian developer Thorsten Glaser named the pattern in a sentence: "all three of the involved ones were 'string + number @ freemailer' `#JiaT75` sockpuppets" (Hess 2024, msg #87). His judgment came after the backdoor was public, which is the point. The heuristic is simple, and no one had reason to run it in time.

Everything about coordination and purpose past this point is analysis, and the honest analysts say so. Molly, the EFF systems administrator the previous chapter quoted on the operation's patience, told *The Intercept* that new, history-less accounts appearing to coordinate on specific goals at key moments "fits the pattern of using networks of sock accounts for social engineering that we've seen all over social media," and then supplied the sentence this chapter is obliged to keep stapled to that assessment: "Of course, it's also possible these are just coincidences" (Mazurov 2024). The academic reconstruction by Piotr Przymus and Thomas Durieux reads the accounts as "likely created to add credibility and pressure," part of "a pattern that retrospectively appears suspicious" (Przymus and Durieux 2025, 92); likely and retrospectively are doing honest work in that sentence. Kaspersky's analysts, who compiled the fullest inven-

tory of the three identities involved, the `JiaT75` GitHub account, the lookalike free-mail addresses, an Internet Relay Chat handle, the mailing-list messages, and the code itself, noticed two things at the layer where personas are built: the identities' implied geographies were conspicuously scattered, "perhaps to dispel hints of coordination," and "[m]isspellings and grammar mistakes are similar across the three identities' communications" (Kaspersky GReAT 2024a). That observation is why the quotations in this chapter preserve every error: in the analysts' reading, the shared mistakes are the nearest thing the record holds to a fingerprint. None of this is proof, and none of it is offered as proof. It is a convergence of independent readings on one inference: that the chorus had a single author, or a single employer.

If the inference is right, the method has a name and a manual. The pattern, *The Intercept* observed, fits "what's known in intelligence parlance as 'persona management,' the practice of creating and subsequently maintaining multiple fictitious identities"; a leaked document from the defense contractor HBGary Federal sets out the meticulousness the craft can involve, down to the construction of an elaborate online life for each invented person (Mazurov 2024). The campaign's personas conspicuously lacked one; the elaborate footprint the tradecraft prescribes was, in the same reporting's words, "decidedly missing." The detail cuts two ways, and the book declines to pick. Thinness could be read as amateurism, or as economy: these identities needed to convince no investigator, only a tired maintainer on a quiet list, and they were built to that specification and no further. What the thinness says about who built them belongs to the chapter on the hunt.

Either way, the move itself is old. Thomas Rid, the scholar of disinformation whose history *Active Measures* follows a century of manufactured politics, describes the standing repertoire in terms that need no adaptation here: operations have always employed "influence agents and cutouts," a cutout being the disposable intermediary who stands between an operation and its target, who "may pretend to be something they are not," and "online accounts involved in the surfacing or amplification of an operation may be inauthentic" (Rid 2020).

Rid also states, better than any source in the XZ Utils record, why none of this was visible from inside the mailing list: "It is very hard to distinguish . . . between a cunning influence agent on the one hand, and a genuine activist on the other." In practice, he adds, "one individual can be both genuine and an exploited asset, a witting and unwitting collaborator at the same time" (Rid 2020). That edge matters for this story, because the grievance the chorus voiced was not invented. The seven-year release gap was real. The overextended solo maintainer was real. The complaint was, in the abstract, one that genuine users hold and sometimes state just as rudely. A demand does not have to be false to be exploited, and a campaign built from true materials is the kind hardest to see while it is happening and hardest to prove afterward.

Open source's own handbook had described both the play and the costume, years earlier and with no enemy in mind. Karl Fogel's guide to running free-software projects, which earlier chapters drew on for its portraits of maintainership, advises allied contributors, under the heading "Appear as Many, Not as One," that "having several people chime in with agreement early on can help it along, by giving the impression of a growing consensus. Others will feel that the proposal has momentum, and that if they were to object, they'd be stopping that momentum" (Fogel 2020, 72). Fogel is describing legitimate coordination among colleagues who actually exist, and he flags the hazard even so. The sock puppets ran the malicious build of the same mechanism: a simulated consensus, several voices lending one proposal momentum, pointed at a single reader. The same handbook catalogs the figure the personas dressed as: the genuinely difficult community member, "not overtly rude" but a manipulator of process, who looks "for wedgepoints in the project's procedures, to give themselves more influence than they might otherwise have," and whose harm spreads quietly because "neither the behavior nor the damage it causes is apparent to casual observers" (Fogel 2020, 104). Every project of any age has met that person, and projects had learned to absorb him as a cost of openness. That is what made the costume effective. An insistent stranger with strong opinions about the repository is the most ordinary nuisance in open source. From inside the mailing

list there is no way to tell the genuine article, who in Fogel's account acts from temperament rather than calculation, from the same figure being operated as cover.

Even the choice of target was traditional. Maintainers have been pressured out of their posts by impatient users since before most of the modern internet existed. In the early 1990s the first two leads of Linux's networking code, Ross Biro and then Fred van Kempen, each gave up the role under sustained community impatience; Glyn Moody's history of the free-software movement quotes the community's own Networking-HOWTO recording that users wanted something reliable and that pressure on van Kempen rose as it had on Biro (Moody 2001). That pressure was genuine, the ordinary friction of a young community wanting working software. The rhyme is the point: the force that wears down maintainers had existed for three decades. The campaign against Collin invented nothing. It synthesized a known, ambient force at higher concentration and aimed it with intent, which is also why it read as weather to everyone watching.

If the reading of the chorus as technique still carries a residue of hindsight, the strongest corrective arrived two weeks after the disclosure, from institutions reporting an attempt they had caught. In mid-April 2024 the OpenJS Foundation, the nonprofit home of JavaScript projects used by billions of websites, and the Open Source Security Foundation (OpenSSF) published a joint alert built on a firsthand account: the OpenJS council had "received a suspicious series of emails with similar messages, bearing different names and overlapping GitHub-associated emails," imploring the foundation to update one of its popular projects to "address any critical vulnerabilities" while citing no specifics, and the author or authors "wanted OpenJS to designate them as a new maintainer of the project despite having little prior involvement" (Bender Ginn and Arasaratnam 2024). The foundations drew the comparison themselves, in print: the approach "bears strong resemblance to the manner in which 'Jia Tan' positioned themselves in the XZ/`liblzma` backdoor" (Bender Ginn and Arasaratnam 2024). The XZ Utils attack, the alert warned, "may not be an isolated in-

cident." For this chapter the OpenJS attempt is the contrast case the XZ Utils record lacks. Same shape: multiple names, overlapping addresses, manufactured urgency, a demand for maintainership without a history of contribution. Different outcome: the council recognized it and stopped it. A move that can be recognized and stopped is a technique, not a coincidence of rude strangers, and an institution catching it in the act is as close as the public record comes to showing the play from the defender's side.

The alert then did something institutions rarely do in the middle of an incident: it published the pattern. Its list of suspicious patterns in social-engineering takeovers opens with three items that read as a postmortem of the xz-devel correspondence, though they were written as guidance for everyone: "[f]riendly yet aggressive and persistent pursuit of maintainer" by "relatively unknown members of the community"; a "[r]equest to be elevated to maintainer status by new or unknown persons"; and "[e]ndorsement coming from other unknown members of the community who may also be using false identities, also known as 'sock puppets' " (Bender Ginn and Arasaratnam 2024). This is where the chapter's title term stops being internet slang and becomes institutional vocabulary, defined in print by bodies responsible for the ecosystem's security. Further down the list sits the sharpest entry: "[a] false sense of urgency, especially if the implied urgency forces a maintainer to reduce the thoroughness of a review or bypass a control." Urgency, in that sentence, is not a mood but a weapon aimed at the one safeguard a tired volunteer still operates, the review itself, and the observation will be worth remembering when the backdoored release reaches the distributions and the hurrying begins again downstream. And the alert recorded scale: the OpenJS team had "recognized a similar suspicious pattern in two other popular JavaScript projects" and flagged its concerns to CISA, the American government's civilian cyber-defense agency (Bender Ginn and Arasaratnam 2024). Whether any of those attempts shared an author with the XZ Utils campaign is precisely what the documented record cannot say. What it can say is that by spring 2024 the technique was visible across the ecosystem, which is

a finding in its own right, and one the chapter on the hunt returns to: a method this reproducible implies an actor, or actors, capable of running it more than once.

Why does the technique work? The foundations' alert answers as bluntly as any source in the record, in guidance addressed to maintainers at large: "These social engineering attacks are exploiting the sense of duty that maintainers have with their project and community in order to manipulate them," and the interactions to watch for are the ones that "create self-doubt, feelings of inadequacy, of not doing enough for the project" (Bender Ginn and Arasaratnam 2024). That is the institutions describing a method, and the book reads it here as nothing else; what the campaign actually produced in Collin is his and not the record's. But set the method beside the documented messages and the fit is exact. Every note the chorus struck was aimed at duty rather than competence: patches "bit rotting away," a choked repository, a community that wanted more. No message claimed Collin's code was bad. Every message implied his stewardship was insufficient, and insufficiency is the one charge that a culture that frames maintenance as a duty owed to users leaves a volunteer no answer to except more of himself. The kernel community's burnout session, convened the following year and described earlier in this book, would catalog the lay registers of that vulnerability, the maintainer who is "ignored," the maintainer who is "disrespected" (Chance 2023); mapping those registers onto this campaign is the book's analytic gloss, not the session's, but it is hard to reread the two voices, one supplying the disrespect and the other the disappointed patience, without noticing how cleanly they divided the work.

The deepest answer, though, came from the engineer who would eventually catch the backdoor. Andres Freund, a longtime open-source maintainer himself, read the pressure campaign after the fact, knowing exactly what it had been, and what struck him was not its artfulness but its familiarity. "I've seen people behave worse," he said of the messages, worse from real people, writing under stable identities they had used for years (Freund 2024c, 01:04:16). The observation deserves its full weight, because it explains the camouflage better than

any analysis of tradecraft. A manufactured chorus of hostile strangers passed as genuine because genuine hostility toward maintainers is common enough, and routinely worse, that nothing in the campaign's register stood out against the background. The operation did not have to imitate something rare. It imitated the ordinary abuse of the people who hold open source up, the standing weather an earlier chapter documented, and the imitation passed because the original is everywhere.

Strip the analysis away and keep only what is documented, and the residue is still striking in its shape: two accounts with no past and no future, arriving within weeks of each other; a grievance manufactured in the thread of the patch it served; a personal disclosure converted into leverage within thirteen days; a demand stated, repeated, and met. Add the analysis, marked as analysis, and the shape acquires a name: a persona-management campaign, thin in construction and precise in aim, run against a maintainer whose circumstances an earlier chapter laid out in full. What this phase attacked was not code. It attacked the norms that make open source work at all, the patience with strangers, the presumption of good faith, the duty felt toward users, and it worked because those norms were load-bearing and unguarded. The campaign had run for eight weeks, and it had its yield. The maintainer had floated his own succession in public, the successor was on hand and already trusted, and the strangers who had demanded it had begun to go quiet. What remained was the transfer itself: the keys, the privileges, and the peculiar, unaudited intimacy of an open-source handover. That is the next chapter.

Chapter 8

The Handover

The campaign had its concession, and from here the story can sound, in the retelling, like a theft. It was not one. Nothing in what follows was broken into, nothing was stolen, and no flaw in any line of code was exploited. Between the summer of 2022 and the beginning of 2024, the most consequential rights an open-source project has to give passed from Lasse Collin to "Jia Tan" through a sequence of ordinary, documented, welcome acts: the right to accept changes into the official repository, the right to speak for the project in public, and finally the right to create and sign the software the world would download. Every step had a precedent. Every step had a justification. Most of them are still visible in the public record. The operation needed no vulnerability in the code, because the vulnerability was the transfer itself, a transaction in trust that no one anywhere was charged with auditing.

The analysts who reconstructed the operation afterward converged on that reading from independent directions. Piotr Przymus and Thomas Durieux, in the academic study an earlier chapter drew on, classified what they had found as "a new breed of supply chain attack that manipulates software engineering practices themselves . . . to establish legitimacy and maintain long-term control" (Przymus and Durieux 2025, 91). The practices, not the software. Kaspersky's analysts reached for the comparison the book has already met: what had distin-

guished the SolarWinds compromise from earlier supply-chain attacks was the intruders' prolonged, covert access to the place where the software was made, and those intruders had to break in to get it. "In this XZ Utils incident," they wrote, "this prolonged access was obtained via social engineering and extended with fictitious human identity interactions in plain sight" (Kaspersky GReAT 2024a). The security researcher Kevin Beaumont, writing while the response was still underway, compressed the same finding into a clause: "somebody appears to have hijacked the open source XZ project by social engineering the volunteer developer into handing over maintainer access" (Beaumont 2024). The hedge is his, and it is the right one. But the verb wants correcting. A hijacking implies force. What the record shows is closer to an application: one made over eighteen months, to a system whose rules for granting trust were public, reasonable, and met.

Those rules deserve to be read closely, because they are good rules and the operation obeyed all of them. Karl Fogel's handbook, the one earlier chapters consulted on the welcoming ideal and the difficult community member, devotes a section to choosing committers, the people trusted to write to the official repository. "A good basis for choosing committers is the Hippocratic Principle: first, do no harm," Fogel writes. "The most important criterion is not technical skill or even deep familiarity with the code, but simply that a person show good judgement," and if a candidate's "patches apply cleanly, do not contain bugs, and are mostly in accord with the project's log message and coding conventions, and there are enough patches to show a clear pattern, then an existing committer should propose him for commit access" (Fogel 2020, 151). Set that standard against the record of the previous two chapters: clean, conventional patches, no bugs, a clear pattern, sustained for a year and a half. The criterion is demonstrated good judgment, and good judgment was demonstrated. Nadia Eghbal, surveying the ecosystem at large, observed that maintainers "often emerge *de facto*, based on who authored the project or put in significant time or effort" (Eghbal 2016, 61): not appointed, accreted. And at the top of the ecosystem, the model's most successful practitioner described the same mechanism

as a virtue. "First somebody volunteers to maintain it," Linus Torvalds wrote of the kernel's subsystems in *Just for Fun*, the memoir he published in 2001. "Then the process for maintaining all the subsystems becomes organic. People know who has been active and who they can trust, and it just happens. No voting. No orders. No recounts" (Torvalds and Diamond 2001, 121).

"No voting. No orders. No recounts" is the absence of an auditor stated as a strength, by the person at the center of the model where it has worked best. In the same memoir Torvalds described what accepting good work does to the person who accepts it: "it was a way of getting people to trust me. And the trust compounds. When people trust you, they take your advice" (Torvalds and Diamond 2001, 120). Two decades later, speaking after the XZ Utils disclosure, his premise had not changed: "Open source in many ways relies in a certain amount of trust, where you trust the developers, you trust your co-maintainer, you trust the people around you to do the right thing" (Mastery Learning 2024, 0:40). Eric Raymond, the movement's most confident early theorist, had argued in *The Cathedral and the Bazaar*, the essay that became something like open source's founding self-description, that no audit was needed because reputation did the work: "The open-source community's internal market in reputation exerts subtle pressure on people not to launch development efforts they're not competent to follow through on. So far this seems to have worked pretty well" (Raymond 2001, 48). The market worked exactly as described. It screened for competence, and the operator was competent. What it does not screen for, because no reputation market can, is intent. The sociologist Susan Leigh Star, writing about infrastructure in general and with no adversary anywhere in mind, described in 1999 how newcomers join one: "Strangers and outsiders encounter infrastructure as a target object to be learned about. New participants acquire a naturalized familiarity with its objects, as they become members" (Star 1999, 381). That is apprenticeship. It is also, step for step, the operation's first phase. The filters were not evaded. They were satisfied, by an adversary who had evidently studied what they filter for, and the handover that followed was not a failure of the community's rules but their correct application.

In the public record the transfer begins on May 19, 2022, the day the previous chapter marked: the day "Dennis Ens" first asked whether XZ for Java was still maintained. Collin's reply carried the first public signal that the helpful insider might become something more: "Jia Tan has helped me off-list with XZ Utils and he might have a bigger role in the future at least with XZ Utils. It's clear that my resources are too limited (thus the many emails waiting for replies) so something has to change in the long term" (Cox 2024, 2022-05-19). Three weeks later came the June 8 reply two earlier chapters have read, with its float of a maintainership change after the `5.4.0` release (Collin 2022). And on June 29, Collin made the standing of the arrangement explicit: "As I have hinted in earlier emails, Jia Tan may have a bigger role in the project in the future. He has been helping a lot off-list and is practically a co-maintainer already. :-) I know that not much has happened in the git repository yet but things happen in small steps. In any case some change in maintainership is already in progress at least for XZ Utils" (Cox 2024, 2022-06-29). The message rewards slow reading. Practically a co-maintainer already: the trust is acknowledged as a standing fact before any repository permission reflects it. Not much has happened in the git repository yet: the visible, auditable record lags the real transfer, by the maintainer's own description. The handover happened in the social layer first, where no audit reaches, and the technical record spent the next year catching up. One disclosure-day reader of the correspondence noticed a further implication and offered it as an inference, which is what it is: the off-list channel "makes the activities of 'Jia Tan' harder to audit and could even have been why they diverted discussion off-list," though "I'm sure it seemed innocuous to the maintainer" and only "thanks to hindsight" does it read as "a significant social element of this attack" (Corbet 2024, helsleym comment, 2024-03-30). Hindsight is doing real work in that sentence, as the commenter concedes. What is documented is simpler: the collaboration that became a co-maintainership ran, by Collin's own account, through a channel no one else could see.

The formal steps followed across the next year, each one small, each one ordinary. In October 2022, by Thomas Roccia's reconstruction, the `JiaT75`

account was added to the Tukaani organization on GitHub, the organizational home of the project's repository there (Roccia 2024, 9:00); membership in an organization is a standing rather than a specific power; the specific rights came one at a time. By the turn of 2023, direct merge authority was visible in the record, as an earlier chapter showed, and by the middle of that year Collin was routing his own work through the newcomer's hands. Akamai's researchers summarize the escalation's shape: "Eventually, after building trust and credibility, Jia Tan began to receive permissions for the repository — first, commit permissions and, eventually, release manager rights" (Akamai Security Intelligence Group 2024). The summary is secondary, and the public record does not date the final step; what it supports is the direction, from the right to change the code toward the right to package and publish it. The endpoint is fixed by the release artifacts themselves.

Around the dated permissions ran a quieter set of transfers that no one dated at the time, because no one was watching them at all. When the project's development moved onto GitHub's infrastructure, Przymus and Durieux found, it was the operator who "created the organization and the repository and therefore gained ownership of them," and who used the occasion "to create an issue template where his email was used instead of the email of the main contributor" (Przymus and Durieux 2025, 97); by the same study's account he had become the project's de facto community manager, handling announcements and releases, more publicly active than Collin himself (Przymus and Durieux 2025, 97). In March 2023, by Evan Boehs's reconstruction, the primary contact address for `xz` in `oss-fuzz`, a Google service that continuously tests open-source software for crashes and exploitable bugs, was updated from Collin's to "Jia Tan's" (Boehs 2024). Dan Goodin's account adds the sequel that matters to the next chapter: the new contact "requested that oss-fuzz disable the ifunc function during testing," a change Goodin reports kept the service from detecting the malicious changes the operator would soon make to XZ Utils (Goodin 2024). The project's xz.tukaani.org subdomain, for its part, came to point at pages under GitHub's control, extending the operator's reach over the project's

public face (Boehs 2024). Each act is administrative and individually plausible, the kind of housekeeping a project's most active member does. Together they are authority moving through small edits that no process flags. And here the chapter must be as precise as the primary record allows, because the takeover was bounded, and the boundary matters. Collin fixed it himself on the incident page he keeps at tukaani.org: "Only I have had access to the main tukaani.org website, git.tukaani.org repositories, and related files. Jia Tan only had access to things hosted on GitHub, including xz.tukaani.org subdomain (and only that subdomain)" (Collin, n.d.). The operator never held the whole project. He held the GitHub side of it, which turned out to be the side the world downloaded from, and the edge of that reach will matter again when the book takes up the hunt.

Of all the rights that moved, one mattered above the rest, and it needs a moment of plain English. An official release of `xz` is not the git repository; it is a tarball, the bundled file earlier chapters have described, created by a maintainer and posted for the world to download. To vouch for it, the maintainer signs it: using a private cryptographic key that only he holds, he generates a signature that anyone can check against his published key. A valid signature proves two things, that the file was produced by the holder of the key and that not a byte of it has changed since. Many distributions check those signatures before accepting a new version. The authority to create and sign releases is therefore the single most consequential right in the chain that runs from a volunteer's laptop to the world's servers, and it is the right that transferred. Collin's incident page states the division that resulted with the precision of a land registry: "Tarballs created by Jia Tan were signed by him. Any tarballs signed by me were created by me" (Collin, n.d.). His commit-by-commit review names the operator's output: the release tarballs of `5.2.12` and `5.4.3` through `5.4.6` "were created and signed by Jia Tan. These have been checked and they don't contain malicious content" (Collin 2024b). That sentence bounds the damage, and the bounding is part of the story. The operator created and signed five clean releases before the two poisoned ones, which is to say that the signing authority

was held, exercised, and trusted across five releases before it was abused.

What a signature cannot prove is good faith. The mathematics vouches for the keyholder's identity and the file's integrity; it is silent on the keyholder's intentions, and once the trusted signer is the adversary, the strongest verification instrument the ecosystem has becomes worthless precisely where it is most needed.[1] The people cleaning up understood this immediately. In Gentoo's bug thread, in the first days of the response, one participant put the corollary as a question: "If you don't trust them you can't trust what they've signed, can you?" (James 2024a, comment 11), and Sam James, the Gentoo developer the book met on disclosure day, confirmed it: "But yes, if we can't trust them, we can't trust them.." (James 2024a, comment 13). James rolled the distribution back to `xz-utils-5.4.2` with a rationale that is candid about its own logic: "This is the last release signed by Lasse Collin, the previous signer of `xz-utils` releases. Downgrade to this out of an abundance of caution. We are not aware of any issues that *specifically* require this" (James 2024a, comment 16).[2] No flaw had been found in the versions being abandoned, and the commit message says so. The fix was not winding back code. It was winding back the handover, to the last artifact vouched for by the man whose trust had never been transferred.

Why did none of this trip an alarm? The honest answer is that there is no alarm to trip, by design, and for reasons that are not foolish. Fogel's handbook describes how maintainership is actually granted in a healthy project: "the usual way is that an existing maintainer posts to a private mailing list consisting only of the other maintainers, proposing that the candidate be invited to join," and the privacy is deliberate, because candid discussion of a person requires it: "For this process to be open and frank, the mere fact that the discussion

[1]The gap between what a signature certifies and what it cannot returns in a later chapter as a general principle: a signature can establish who produced a release and that it arrived unaltered, never whether the signer acted in good faith.

[2]"The last release signed by Lasse Collin" is precise about the release *tarball*, which Gentoo's packaging verifies against Collin's key; the `v5.4.2` tag in the git repository was made by "Jia Tan," while the `5.4.2` tarball itself was created and signed by Collin (Collin 2024b). The split between what the repository shows and what the shipped artifact carries returns at the center of the next chapter, as the operation's decisive concealment choice.

is taking place at all should be secret" (Fogel 2020, 63–64). The grant of trust is private, unaudited, and secret by design, in the service of kindness and candor. But the model assumes the one thing xz did not have: other maintainers. Collin was alone, so the private deliberation had a membership of one, a man whose circumstances three chapters have laid out. And once granted, the trust does not expire. Fogel again, arguing against revoking the access of committers who have drifted away: "You trusted her judgement before, so why not trust it always? If high school diplomas do not expire, then commit access certainly shouldn't" (Fogel 2020, 152). He offers it as humane policy, and it is. It is also the guarantee the operation needed on the far side of the handover: trust, once extended, is never re-examined.

The 2020 contributor survey earlier chapters have drawn on, a Linux Foundation and Harvard study of the people who keep open source running, recorded the same model from the inside, in the voices of the people operating it. Asked why their projects did not require digital signatures on commits, one respondent described where security actually rests: "Trust is placed in the subsystem maintainers who review, sign-off, and forward changes, and in the public review system, rather than trusting individual contributors" (Nagle, Wheeler, et al. 2020, 65). Read against this story, the sentence is exact. The model's entire security budget is spent on trusting the maintainer, and it has no provision for the case in which the maintainer is the problem; once the operator became one, the model had nothing left to check. Another respondent stated the economics: "No projects require it, because the friction that [it] would cause for contributions is never worth the security benefits of having it" (Nagle, Wheeler, et al. 2020, 65). That is not negligence. It is a rational allocation by projects starved for contributors: verification adds friction, welcome removes it, and a project that optimizes for contribution, because contribution is the thing it cannot live without, is structurally optimized against verification.

The survey's sharpest finding, though, is about how these arrangements come to exist, which is that they do not come to exist; they persist. Asked about two-factor authentication, the second check (a code from a phone, a hardware

key) that keeps a stolen password from being enough, one respondent explained its absence in words the report adopted as the general pattern: "It wasn't a decision, it was the default" (Nagle, Wheeler, et al. 2020, 66). Nobody had decided to run the project unguarded, because nobody had decided anything. The handover had no auditor for the same reason: the audit was not refused, it was never instituted. And one survey question reads, in hindsight, as if it had been drafted with the `xz` tarballs in view: it asked whether projects sign their released versions "so that recipients can verify who released it even if the distributing repo might be subverted" (Nagle, Wheeler, et al. 2020, 64). Of the 720 contributors who answered, 35.97% said none of their projects signed releases, 41.53% said some did, and 22.5% said all did (Nagle, Wheeler, et al. 2020, 64). The numbers describe an ecosystem that cannot count on the defense. The XZ Utils case is worse than the numbers, because `xz` had it. The releases were signed, immaculately. The signatures were the operator's.

Even the law had words for the gap before the handover began. In May 2021 the executive order on cybersecurity that the United States issued in the wake of SolarWinds listed, among the baseline practices of secure software development, "auditing trust relationships" (Executive Office of the President 2021, sec. 4(e)(i)(B)). Read quickly, that is the missing control, named a year before Collin floated the handover. Read closely, it is aimed at a different species: the term the order defines is the "auditing trust relationship," "an agreed-upon relationship between two or more system elements" (Executive Office of the President 2021, sec. 10(b)), machines agreeing with machines. The framework that grew from the order, the National Institute of Standards and Technology's catalog of secure-development practices, recommends commit signing and code-owner review (National Institute of Standards and Technology 2022, 9, PS.1.1, Examples 3 and 4). Both practices are sound, and both presuppose the one thing the operation falsified: that the code owner is the party to be trusted. The campaign's entire object was to become the code owner, after which the operator's changes were the authorized, reviewed, signable ones. That is the book's reading of a gap, not a charge the framework makes against itself; the controls were

not bypassed, they were captured. The same year's federal specification for the software bill of materials, an ingredients list for software, encourages signing so that a user can confirm "whether the signature is legitimate" (National Telecommunications and Information Administration 2021, 16). The signatures on the backdoored releases were legitimate. Whether the legitimate signer was acting in good faith is a question none of the instruments knew how to ask, because the apparatus models software as artifacts moving between systems, and the XZ Utils handover was a transaction between two human beings on an exhausted mailing list.

When the institutions responded to the XZ Utils backdoor itself, they finally named the right layer, and the joint alert from the OpenJS Foundation and OpenSSF that the previous chapter quoted is worth returning to for what it prescribes. Open-source projects welcome contributions from anyone, the foundations wrote, "yet granting someone administrative access to the source code as a maintainer requires a higher level of earned trust, and it is not given away as a 'quick fix' to any problem" (Bender Ginn and Arasaratnam 2024). That sentence carries the whole distinction this chapter turns on: the welcome is for contributions; authority is a different grant, and surrendering it under pressure is the move to refuse. The alert's recommended safeguards read as the photo-negative of the handover. "Know your committers and maintainers, and do a periodic review. Have you seen them in your working group meetings or met them at events, for example?" (Bender Ginn and Arasaratnam 2024). No one had met "Jia Tan." No one had seen him. Everything known about him was a username, a commit history, and a free-mail address. The list's sharpest item concedes the operation's whole premise: "If possible, have a second developer conduct code reviews before merging, even when the PR comes from a maintainer" (Bender Ginn and Arasaratnam 2024), a PR being the pull-request form in which a proposed change arrives on GitHub. Even when it comes from a maintainer: the guidance asks projects to distrust the very role the trust model exists to trust, because that role is where the leverage lives. And the hedge at the front, if possible, is not careless drafting. It is the labor problem restated:

the control assumes a second developer, and the book has spent three chapters inside a project that did not have one.

It would be easy, at this point, to conclude that the welcoming culture was the flaw, and the conclusion would be wrong in a way the record itself rebuts. Eghbal preserves the sentence with which one well-known developer publicly handed a project over: "I don't have time to maintain this project anymore, so I gave you commit access to make any changes you'd like" (Eghbal 2020, 96).[3] It is the benign twin of "practically a co-maintainer already," written in good faith, and that handoff did no harm, as most such handoffs do not. What varies across the ecosystem is how much vetting attaches to the gift. Debian, Eghbal notes, requires an extensive onboarding in which a new developer reads a manual, finds a mentor, and meets an existing maintainer in person who can vouch for his identity, while "it's common among JavaScript developers to give away commit access more freely. The idea is to distribute the burden of maintenance by making it easy for others to contribute, and it's assumed that strangers are trustworthy until proven otherwise" (Eghbal 2020, 46). The assumption is not naïveté; it is the engine. The disposition the operation converted into co-maintainership is the same one that makes open source produce at all, and the community states it plainly, as a value worth teaching: the kernel's burnout session, the one an earlier chapter described, lists "[b]uilding trust and integrity through positive acts at the right time in the right way" among the marks of a healthy project (Chance 2023). The gap was never the norm itself. The gap was the absence of anything around it: the transfer of authority, as distinct from the welcome of contribution, had no procedure, no second opinion, and no review, and that absence, like the rest, was nobody's decision.

The concentration of authority the handover exploited is, in the same breath, a real architectural strength, and the people who built the model understood both halves. John Ousterhout, the creator of the Tcl programming language,

[3]The sentence is the developer Felix Geisendörfer's, quoted by Eghbal (Eghbal 2020, 96); it comes from the blog post that Dominic Tarr would later cite to explain the `event-stream` handover, the precedent a later paragraph of this chapter reaches.

described the structure to Glyn Moody as "sort of the opposite of design by committee. . . . There's typically one person who is the god or the tsar who has final authority over everything that goes into the package" (Moody 2001), and credited it with a coherence that committees never achieve. Moody himself, writing in 2001, named the matching fragility: "without the right kind of leadership and a process that allows power to pass smoothly to successors, energy will dissipate and free-software projects worldwide will dwindle into irrelevant programming pastimes" (Moody 2001). Succession, the moment final authority changes hands, was identified as the model's point of failure two decades before "Jia Tan," and the well-resourced projects engineered for it. The kernel writes its authority down: the first `MAINTAINERS` file, committed in January 1996, ran 107 lines and named three people; by version `5.8` of the kernel in 2020 it ran 19,033 lines and named 1,501 maintainers (Linux Foundation 2020, 8), a public, versioned registry of who is trusted with what. Debian turned trust itself into procedure: the anthropologist Gabriella Coleman, in her ethnography of the project, describes how requiring each new developer to obtain the signature of an existing one links their keys until "nearly all maintainers are connected" in what its developers call a cryptographic "web of trust" (Coleman 2013, 143). None of this is a prescription for a compression library with one volunteer; Debian's apparatus is the work of a large institution with decades of accumulated procedure. The point of the contrast is narrower. The problem was understood, and the solutions were built, wherever there were people and money to build them. They were never resourced for the long tail, where `xz` lived, and where the authority over a library inside everything was one exhausted volunteer's to give away in a mailing-list reply.

Nor does it take an adversary to make an unmanaged transfer go wrong, which is the last reason to keep the norm and the gap distinct. Eghbal records the case of Express, a widely used web framework whose stewardship sat with a company called StrongLoop while the actual work fell to one unaffiliated maintainer, Doug Wilson; when the arrangement finally broke, Wilson renounced it in public: "No matter what happens, I will not ever commit again to any

repository under StrongLoop's name" (Eghbal 2016, 104). The keys and the burden had come apart, with no malice anywhere in the story. And the thinness ran downstream of xz as well as upstream. In the Debian bug thread after the disclosure, Pierre Ynard, whose verdict on the upstream conditions an earlier chapter quoted, pointed at the distribution's own arrangement: "this package has been NMU'd in Debian for more than 10 years now - thanks to Sebastian for his work - and that might have been a point of weakness which was exploited" (Hess 2024, msg #40). An NMU, a non-maintainer upload, is Debian's mechanism for keeping a package alive when its official maintainer has gone quiet: a defensible, generous norm, and a decade of it meant that the package's path into Debian ran on the same thinly stretched goodwill as its path out of Collin's hands. The same shape appears at both layers, and at both layers the norm was carrying load that nothing else had been funded to carry.

None of this was unforeseeable, and the evidence that it was foreseen is specific. The 2020 catalog of malicious-package attacks an earlier chapter cited, by Marc Ohm and colleagues, did more than count typosquats: it drew the attack tree, the diagram of every known route into the software supply chain, and one labeled branch reads, in the paper's own words, that "attackers may become maintainer themselves through social engineering" (Ohm et al. 2020, 7). The path the operation walked was a diagrammed node in a peer-reviewed venue before "Jia Tan" held co-maintainership. The same page names the adjacent vector, the takeover of a project whose maintainer has stepped away, with a term the authors borrow from memory safety, *use after free* (Ohm et al. 2020, 7): the maintainer withdraws, and the vacant trust slot is occupied by someone else. That is the structural shape of an exhausted solo project, generalized into a recognized pattern, with no reference to xz and four years before anyone needed one. The diligence that would have applied had been prescribed too, by Russ Cox, in the 2019 essay on dependencies the book has quoted before: "You would not hire a software developer you have never heard of and know nothing about. You would learn more about the person first: check references, conduct a job interview, run background checks, and so on" (Cox 2019, 38). Once the

program ships, he observed, it "literally depends on code downloaded from this stranger on the Internet" (Cox 2019, 37). In XZ Utils the stranger was not merely depended on. He was made the maintainer, with no reference checked, because there was no one whose job it was to check and no process anywhere that required it.

The exposure had even been measured. Census II, the companion study to the contributor survey, set out in 2020 to identify the most depended-upon open-source packages and then looked at where they lived: seven of the ten most-used packages in its analysis were hosted under individual developer accounts, the personal account of a single person rather than an organization, where, as the report put it, a change is "significantly easier to make, and to make without detection" (Nagle, Wilkerson, et al. 2020, 28). Easier to make without detection, written in 2020 from 2018 data, as a description of the ecosystem's normal state. The report's own marquee example was the closest precedent XZ Utils has. In 2018 a popular JavaScript library called `event-stream` was compromised when, in the report's words, "a malicious actor gained legitimate publishing rights to the event-stream package, and then wrote a backdoor into the package itself" (Nagle, Wilkerson, et al. 2020, 28). Legitimate publishing rights: the authority was transferred, not breached, and then abused. Dominic Tarr, the maintainer who handed it over, explained himself afterward; in Eghbal's words, "far from recklessness or a mistake, handing off maintenance to strangers was considered a best practice among many JavaScript developers" (Eghbal 2020, 96). The XZ Utils pattern in miniature, six years early, publicly dissected, and absorbed by the ecosystem as a story about one library rather than about the transaction. The vector was cataloged, the diligence was prescribed, the precedent was famous. The defenses were not missing because no one had thought of them. They were declined, rationally, one undecided default at a time.

The oldest voice among the book's sources had named the move as well, thirty-five years earlier. Clifford Stoll, describing a Trojan horse in 1989, reached past the technology to the technique: "Deliver a gift that looks attrac-

tive, yet steals the very key to your security. Sharpened over the millennia, this technique still works against everyone except the truly paranoid" (Stoll 1989). The level is different, and the difference measures what changed. Stoll's gift was a fake program, attractive on the surface and hollow inside. The XZ Utils operation's gift was years of real work, attractive because it was genuinely good, and what it took was not a password file but the position of the very person anyone would have asked to check. The technique is as old as Stoll says. The refinement was to make the gift indistinguishable from the thing the recipient most needed, which, against an unsupported solo maintainer, meant one thing: help.

The last word on the transfer belongs to the man who eventually exposed it, because he is a working maintainer himself and he read the record the way a maintainer reads it. Asked what the episode should teach, Andres Freund pointed at the precise joint this chapter has anatomized: the manufactured pressure of the previous chapter existed, in his words, "to pressure the maintainer to relinquish control. So I think, if you get pressured to relinquish control, that's a pretty big warning flag" (Freund and Roccia 2024). And he declined the comfort of treating the exposure as a small-project problem, observing that the surface is not confined to the long tail: "It's not that hard to . . . become a committer in some big projects either" (Freund 2024c, 01:06:43). The warning generalizes because the transaction generalizes. Wherever software is maintained by people, authority must sometimes change hands, and almost nowhere is the change anyone else's business. In XZ Utils it had now changed hands completely. By early 2024 the operator held everything the cultivation had been for: the standing to write to the official repository, the administrative reach over the project's public face, and the keys that signed what the world would download. The next chapter is about what he shipped with them.

Chapter 9

The Payload

What the operator shipped, once the analysts finally took it apart, provoked a reaction that complicates everything else about it: the people best equipped to despise the thing admired it. That response has a respectable pedigree. Donald Knuth, the computer scientist whose multivolume *The Art of Computer Programming* gave the field its working sense of its own craft, used his 1974 Turing Award lecture to insist that the word "art" in that title was exact and not decorative: "When I speak about computer programming as an art," he said, "I am thinking primarily of it as an art form, in an aesthetic sense" (Knuth 1974, 670). The claim he built on it was about reading, not writing. "When we read other people's programs," Knuth went on, "we can recognize some of them as genuine works of art" (Knuth 1974, 670). The recognition happens in the act of dissection, which is exactly where it happened here: not in the operator's composition of the backdoor but in the analysts' taking it apart, the same way Lasse Collin read his own poisoned repository afterward and marked, commit by commit, what had been done to it.

The vocabulary the reverse engineers reached for was not loose. Thomas Roccia, a security researcher at Microsoft, said it plainly on a podcast in the first days: "This is super sophisticated," and unlike anything he had seen before (Freund and Roccia 2024). Kaspersky's analysts, closing a teardown dense

enough to satisfy any specialist, wrote that "several highlights make this threat unique" (Leite 2024). Naming that artistry is part of taking the threat seriously; flinching from it would understate what the world nearly absorbed. But the admiration has to be held at arm's length, and the limit set in the same breath as the praise: technical admiration must not become glamour. The craft mattered only because it exploited real blind spots, the places where humans and their tooling do not look. Knuth's lecture supplies its own escalating palette for skilled work, "elegant," "exquisite," "sparkling," and one more besides, "noble" (Knuth 1974, 670). That last word is the one this program cannot earn.

The first move was the quietest. Collin, reviewing the wreckage line by line after the disclosure, put his finger on the exact commit, in January 2024, where more than two years of genuinely useful contribution turned into staging for an attack: "This is the first commit preparing for the backdoor," he wrote against the addition of a pair of innocuous-looking test files (Collin 2024b). Test files are an unglamorous category. A compression library has to prove it can compress and decompress real data, including malformed or "corrupt" data it should reject cleanly, so a directory of binary sample inputs is ordinary and expected. The payload rode inside that ordinariness. As Collin described it once he had cut it out, "the executable payloads were embedded as binary blobs in the test files" (Collin 2024a), opaque chunks of data that no reviewer reads because there is nothing in them a human can read.

The directory was effectively unreviewable, and not by accident. The project's own README had explained, long before "Jia Tan" arrived, that many of its test files were built by hand and could not be regenerated from anything more legible: "Many of the files have been created by hand with a hex editor," it noted, so "there is no better 'source code' than the files themselves" (Cox 2024, 2024-02-23). Russ Cox, whose timeline became one of the standard references, flagged that the note predated the operator: the concealment exploited a legitimate, standing practice rather than a flaw introduced for the purpose. The test harness made the trick cheaper still. As Collin observed, the test runner selected its inputs by wildcard, so dropping a file into the directory

was enough to wire it into the build's own checks: "Backdoor files. . . . Note that `tests/test_files.sh` uses globs to pick the files. So just adding files means that a decompression test will be done with them" (Collin 2024b). The camouflage was layered even inside the fixtures. The malicious files carried real test content, genuine coverage for the library's RISC-V filter, but padded far past what that content needed; the surplus was the payload. "The RISC-V test files also have real content that tests the filter," Collin wrote, "but the real content would fit into much smaller files" (Collin 2024a). And the fixtures were not even doing the job their category implied. Andres Freund, in the disclosure email, noted that in the first poisoned release "the files were not even used for any 'tests' in `5.6.0`" (Freund 2024b). Calling them test fixtures was itself the disguise.

The hiding place was chosen for what it is. Test infrastructure is the least-read, most-trusted code in a project. Nadia Eghbal, surveying maintenance at large, cataloged the genre's neglect from the inside: there are "tests that are no longer useful, tests that are slow, tests that are flaky, tests that don't do what the developer thought it would, and tests that are orphaned by their authors" (Eghbal 2020, 133–34). When the institutions later wrote up the attack for other maintainers, they named the same blind spot precisely. The joint OpenJS and OpenSSF alert listed, among the warning signs of a social-engineering takeover, "PRs containing blobs as artifacts," and gave the case at hand as its example: "the XZ backdoor was a cleverly crafted file as part of the test suite that wasn't human readable, as opposed to source code" (Bender Ginn and Arasaratnam 2024). Not human readable, as opposed to source code: the payload lived in the one corner of an open project where openness stops meaning anything, because there is nothing legible there to be open about.

How a blob in a test file becomes a running hook inside a login server is a staged chain, and the staging is where the craft is most visible. The trigger that started it has already appeared in this book: the single malicious line in `build-to-host.m4`, the build-system macro that an earlier chapter watched Freund trace, present in the released tarball and absent from the source repos-

itory. What that line set in motion is the rest of the chain. At the moment a packager prepared the software, the macro decoded the first test fixture, `bad-3-corrupt_lzma2.xz`, into a shell script; that script performed a more elaborate decode on the second fixture, `good-large_compressed.lzma`, producing a further script; and that script extracted a compiled object file, `liblzma_la-crc64-fast.o`, and slipped it into the compilation of `liblzma`, the compression library at the center of the whole story (Akamai Security Intelligence Group 2024). The formal record compresses the same sequence into a sentence: "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" (CVE Program 2024). There is a small, dark elegance in the means. To unpack each stage, the chain used `xz` itself, the very tool it was poisoning, as the instrument of its own corruption (Kaspersky GReAT 2024b).

Linking a hostile object file into a library is only useful if something calls into it, and the mechanism that arranged that is the most quietly clever part of the build. The extraction script made a one-character edit to a line of the library's source, changing a call to a function named `__get_cpuid` into a call to `_get_cpuid`, deleting a single underscore (Kaspersky GReAT 2024b). That edit pointed the call at the attacker's object file instead of the system's. The redirection then exploited a legitimate optimization feature called an indirect function, or `ifunc`: a mechanism that lets a program choose, at the moment it loads, the fastest version of a routine for the particular processor it finds itself running on. The groundwork for that feature had been laid in the repository months earlier, in mid-2023, and Cox marked the interpretive limit honestly when he reconstructed it: the change "could be an innocent performance optimization by itself" (Cox 2024, 2023-06-22), and standing alone it was indistinguishable from one. Turned to the operator's purpose, the `ifunc` machinery became the doorway through which the smuggled code seized the program's flow of execution at startup.

From that foothold the payload reached forward into the login server it had no ordinary business touching. At startup it installed what is called an audit

hook into the dynamic linker, the part of the operating system that resolves a program's references to library functions into actual addresses in memory as the program loads (Freund 2024b). The hook let the backdoor watch that resolution happen and wait for one function in particular. "It appears to wait for '`RSA_public_decrypt@....plt`' to be resolved," Freund wrote, preserving his own uncertainty in the verb. "When called for that symbol, the backdoor changes the value of `RSA_public_decrypt@....plt` to point to its own code" (Freund 2024b). In plain terms, the program keeps a table of pointers to the library functions it calls, and the backdoor rewrote one entry in that table so that a call meant for the system's code arrived instead at the attacker's. The rewrite was possible only inside a narrow window. The table is made read-only once loading is finished, a routine hardening step, and the backdoor struck before it closed: "It is possible to change the `got.plt` contents at this stage because it has not (and can't yet) been remapped to be read-only" (Freund 2024b). The whole construction is a sequence of legitimate mechanisms, each load-bearing in ordinary software, assembled into a path that none of them was built to permit.

What the hijacked function then did is the covert channel, and an earlier chapter already established the capability: pre-authentication remote code execution, the power to run an arbitrary command on a reachable server before anyone has logged in, gated so that only the holder of a secret key could use it. The mechanism beneath that capability turns on a fact that the hooked function's own name obscures: `RSA_public_decrypt` does not decrypt traffic; it verifies signatures. Filippo Valsorda supplied the gloss the rest depends on: `RSA_public_decrypt` "is a (weirdly named) signature verification function" (Valsorda 2024, post 3kowk663sll2p), the misnomer an artifact of the symmetry in RSA's mathematics, confirmed by the function's own documentation, which lists it among "low-level signature operations" (OpenSSL, n.d., NAME). The server reaches that function when it checks the cryptographic key in a connecting client's certificate. That is the moment the backdoor seized.

The command was hidden in the key. A client connecting to a backdoored

server presents a certificate, and the backdoor read its command not from any message field but from the raw number at the heart of the certificate's RSA key, the modulus, conventionally written `N`. Valsorda laid out the layering: "The payload is extracted from the `N` value (the public key) passed to `RSA_public_decrypt`, checked against a simple fingerprint, and decrypted with a fixed ChaCha20 key before the Ed448 signature verification" (Valsorda 2024, post 3kowk2vdagg2c). Each stage in that sentence is a deliberate filter. A cheap fingerprint check rejected ordinary traffic without spending effort; a decryption with a hard-coded key turned the smuggled bytes into a command; and a signature verification, using the elliptic-curve scheme Ed448, proved the command came from the one party holding the matching private key. That last check was bound to the specific machine. The signed data included a hash of the target server's own host key, so a command captured against one server could not be replayed against another, a precaution Kaspersky read, correctly, as the adversary building the defending researcher into its threat model (Leite 2024). Only after all of it passed did the command run. Of the request types the backdoor recognized, the operative one handed its payload straight to the C library's `system()` call, which runs a line of text as a shell command: "Type 2: executes null-terminated payload with `system()`" (Weems 2024). A certificate, the object a server inspects to decide whom to trust, had been turned into the envelope that carried the instruction to betray it.

Two final touches show how completely the design anticipated inspection. The verification key the backdoor used was never stored anywhere an analyst could simply find it; it was taken apart and scattered across the program's own machine instructions, one bit at a time, and reassembled at runtime. Kaspersky's team described being briefly fooled by it, and then the resolution: "the attackers used a custom steganography technique in the x86 code to hide an arbitrary message (in this case, the public key)," recovered only by walking "456 instructions" of the binary to rebuild it (Leite 2024). And the implant did not merely slip past the records a successful intrusion would normally leave; it rewrote them. The backdoor hooked the logging machinery and, in Kaspersky's

flat description of the result, "replaces these successful connection messages with messages about failed authentication attempts" (Leite 2024). A break-in was made to read, in the server's own logs, as a failed login. Anthony Weems, who built a working trigger to prove the capability was real, confirmed the silence from the other side: "successful exploitation does not generate any `INFO` or higher log entries" (Weems 2024).

All of that is the true account of how the trigger was hidden, and it invites a conclusion that is tempting and not quite right: that the decisive concealment was locational. The malicious `build-to-host.m4` shipped only in the release tarballs and never entered the git repository, so anyone reading the public source would have seen nothing. The fact is real and load-bearing. Red Hat stated it within a day: "the Git distribution lacks the M4 macro that triggers the build of the malicious code" (Red Hat 2024).[1] And the gap drew no suspicion because shipping a release tarball that differs from the repository is ordinary practice. As Sam James explained in the FAQ that organized the early response, "the release tarballs upstream publishes don't have the same code that GitHub has. This is common in C projects so that downstream consumers don't need to remember how to run autotools and autoconf" (James 2024b). A reviewer who noticed the difference would have found in it nothing more than the normal seam between a project's source and its shipped artifact.

But the people closest to the artifact doubt that the locational trick was what mattered most. Freund, who caught the thing, said as much when pressed on whether openness had forced the operator's hand: he thought they "could have actually committed it to the repo, and nobody would have noticed" (Freund 2024c, 01:10:11). A Canonical security maintainer, Marc Deslauriers, put the contrary case in the contemporaneous record more sharply still, arguing that the hidden code would likely have escaped notice in git as easily as in the

[1]The absence is verifiable directly. As of this writing the malicious `build-to-host.m4` appears nowhere in the `tukaani-project/xz` history: no commit touches the file, and a request for its contents in the repository returns a "not found" error. The injected macro existed only in the release tarballs "Jia Tan" cut and signed, which is the seam the next chapter follows downstream from upstream release into distribution packaging.

tarball, and that the mismatch was if anything a liability for the attacker: "this was easily spotted *because* the tarball didn't match the git repo" (Corbet 2024, mdeslaur comment, 2024-03-29). Another commenter reconstructed the operator's reasoning the other way, as a deliberate bet that "a commit in git would certainly be more visible" (Corbet 2024, epa comment, 2024-03-30), though that imputes a motive the record cannot confirm: an inference, not a fact.

The tarball-versus-git gap is a real account of how the trigger was placed, and it was at most redundant insurance. The decisive concealment was social, as the previous chapter showed: the things the operator added, test files, a build macro, a release, were the authorized, reviewed, signable output of the very person the project existed to trust. The payload did not have to evade review. It arrived wearing review's own credentials. The real attack surface was never the code's location but the standing that decided whose code shipped at all, and once that standing was captured, where the bytes sat was a detail.

The admiration the craft compels is genuine, and Knuth's lecture is also where it runs out, by the field's own measure. Knuth did not treat his aesthetic as amoral. The pleasure he described was bound to purpose: he confessed to especially enjoying "writing programs which do the greatest good, in some sense" (Knuth 1974, 671). By that criterion, the standard set by the man who made the case for program-as-art in the first place, a program built to compromise the machines of everyone who installed it fails as art no matter how exquisitely it is made. The skill can be conceded without the word: the work was skilled, and the thing it was for disqualifies it from the only praise that would have counted.

The sophistication was also, in plain fact, fragile, and the fragility is part of what got it caught. Freund's own assessment of the build-time machinery was that it was "over-engineered" (Freund 2024a), elaborate past the point of usefulness. The elaboration made it unreliable. "Sometimes when I would rebuild," he recounted, "the backdoor would vanish. And then it would rebuild again, and the backdoor would reappear. And there's also some concurrency bug in the make file" (Freund and Roccia 2024). The same surplus effort produced the tell that exposed the whole operation. The payload looked up far more symbols

than its job required, and that needless work is what cost the roughly half a second of latency the book opened on, the anomaly Freund chased to its source. He came away unimpressed by exactly the part that should have been most careful: it read to him like "less experienced people using the techniques that other people built for them" (Freund 2024c, 41:15). Kaspersky reached the same double verdict, reading the campaign as a "patient, multi-year approach" that was nonetheless "somewhat clumsily executed" (Kaspersky GReAT 2024a). Sophistication and fragility are not opposites here. The over-engineering and the craft were the same thing seen from two sides, and the thing that made the payload remarkable is the thing that made it visible.

None of this was the unforeseeable novelty the early coverage sometimes implied, and the evidence that it was foreseen is specific. The 2020 catalog of malicious-package attacks an earlier chapter drew on, by Marc Ohm and colleagues, mapped the routes into the software supply chain as a tree, and one labeled branch was the concealment of malicious code in test cases (Ohm et al. 2020, 8). The fit is close but not exact: the paper's category is code that executes when the tests run, striking the maintainer's own machine, whereas the operation here used its fixtures chiefly as an inert hiding place for a blob assembled later, at build time. The same paper named the deeper structural fact that the operation turned on. Malicious code committed to a version-control repository, it observed, "is more accessible to manual or automated reviews of commits or entire repositories" than code introduced through the build and packaging path (Ohm et al. 2020, 7), which is the negative space of the concealment choice stated four years early. Ohm and his coauthors also drew the distinction that separates this incident from an accidental one, the line between a vulnerable package and a malicious one: the two "may look identical," they wrote, and "the main difference lies in the intention of the developer" (Ohm et al. 2020, 3). Heartbleed, the catastrophe the book met earlier, was vulnerable: a mistake. This was malicious: deliberate, evasive, authored.

The discipline that would have broken the trick existed too, in writing, before the trigger ever shipped, and went unenforced for the kind of project that

needed it most. The 2021 United States executive order on cybersecurity, issued after SolarWinds, listed among the baseline practices of secure development the maintenance of "provenance (i.e., origin) of software code or components" (Executive Office of the President 2021, sec. 4(e)(vi)), which is precisely the property the tarball broke: the shipped artifact no longer matched the source anyone could inspect. The companion framework from the National Institute of Standards and Technology was almost uncanny read against the incident. One of its examples instructed developers that "if the integrity or provenance of acquired binaries cannot be confirmed, build binaries from source code after verifying the source code's integrity and provenance" (National Institute of Standards and Technology 2022, 12, PW.4.1, Example 8). Build from verified source rather than trust an opaque artifact: the named, published control that the concealment defeated. The framework had not failed: the control was real and it was correct. The economic and social conditions that would have applied it to a one-volunteer compression library simply did not exist, a problem the final chapters return to.

What the artifact establishes, in the end, the formal record states without hedging. The catalog assigned the incident the weakness class CWE-506, "Embedded Malicious Code" (CVE Program 2024), and the precision matters. That category is not a flaw class like a buffer overflow or a memory error, the residue of human mistake. It is the security community's own label for code that was deliberately placed to do harm. The formal record therefore agrees, in the field's own taxonomy, that the XZ Utils backdoor was an authored operation and not an accident. An authored operation implies an author, and the author is still unnamed.

That someone was, the evidence shows, a person at a keyboard, debugging his own malware in real time, which is the detail that most complicates the admiration. The payload broke things, the operator fixed them, and the fixing left fingerprints. Between the two poisoned releases the operator worked on his own code under the cover of ordinary maintenance, and in one of the most revealing lines in the entire record, Collin recognized the cover for what it was: a

routine review comment of his own, about the dictionary size in the RISC-V test files, had been turned into an alibi. "I had mentioned the dictionary size," Collin wrote, "and that gave a good excuse to update something in the backdoor code" (Collin 2024b). Welcomed, well-meant feedback had become operational cover. The figure that remains is not a genius in the abstract but a developer maintaining a flawed implant in public, who used at least one ordinary review comment, in Collin's retrospective reading, as cover for another backdoor change. That figure is harder to romanticize, and closer to the truth.

The craft sits in a lineage, and the lineage is old. Clifford Stoll built his 1989 account around the cuckoo (Stoll 1989), the nesting parasite that leaves its egg in another bird's nest to be hatched and raised by it: hostile code smuggled into a trusted host that then feeds it and grants it privileges. The metaphor is thirty-five years old and fits exactly. The nearer comparison for the sheer target-awareness is Stuxnet, the centrifuge sabotage Kim Zetter reconstructed, a payload that hunted for one precise industrial configuration and ignored everything else (Zetter 2014), with one instructive difference. Stuxnet was built to destroy; this was built for access, a passive door that did nothing until the operator chose to open it. And the design's deepest intention is captured by a maxim recorded by Ben Buchanan and credited to the cryptographer Matthew Green, speaking of a different case: "the best backdoor is a backdoor that looks like a bug" (Buchanan 2020). A backdoor engineered to remain attributable to accident keeps its author hidden even after its mechanism is fully understood. The mechanism of this one is now understood in extraordinary detail, reconstructed down to the 456 scattered instructions. The author is not. Who that author was, the world has spent the years since trying to establish, and failing.

Part III

The Foundation

With the explosion of new developers using, but not giving back to, shared code, we are building palaces on top of crumbling infrastructure.

— Nadia Eghbal, *Roads and Bridges*

Chapter 10

The Release Train

By March 9, 2024, the operation's whole material achievement amounted to two compressed archives on a download server. Lasse Collin would later fix their identity in two flat sentences: "XZ Utils 5.6.0 and 5.6.1 release tarballs contain a backdoor. These tarballs were created and signed by *Jia Tan*" (Collin, n.d.). A backdoor in a release file compromises no one by existing. Between those archives and the millions of servers they were aimed at stood machinery that almost nobody outside the trade ever has reason to look at: the system that turns an upstream release into the operating systems the world installs. The operation's final phase was a campaign against that machinery, and the near-miss at the center of this story happened inside it. The danger was never only that malicious code existed upstream. The danger was that trust, packaging automation, and release calendars were in the process of converting it into infrastructure.

The machinery rewards a plain description. An operating system like Debian or Fedora is not written by anyone. It is assembled. A distribution gathers thousands of independent projects (the kernel, the login server, the compression libraries), builds each into an installable package, and ships the whole as one coherent, vouched-for system; the projects that write the code are upstream, the distributions that package and deliver it are downstream, and the words sim-

ply describe the direction the software flows.

The unit that moves between them is old. In the years before there was any standard platform for hosting code, Nadia Eghbal observes, "most open source code was published as a 'tarball' . . . on some self-hosted, stand-alone website" (Eghbal 2020, 22), and the release tarball, the bundled snapshot at the center of the concealment story, is still the thing an upstream hands to the world. Two sociologists of infrastructure, Geoffrey Bowker and Susan Leigh Star, argued that understanding systems like this means foregrounding their "normally invisible Lilliputian threads" and granting them "causal prominence in many areas usually attributed to heroic actors" (Bowker and Star 1999, 34). The people recede here, and the plumbing decides.

The plumbing is also old in a second sense. Infrastructure, Star wrote, "wrestles with the inertia of the installed base and inherits strengths and limitations from that base. Optical fibers run along old railroad lines" (Star 1999, 382). `xz` is an old railroad line: wired into the foundations years ago, inherited by every new build because the base beneath already carried it, and inherited along with its limitations, which by 2024 included a thinly supported maintainer and a release process nobody else checked.

What a distribution ships is not one product but a graded series of them. The Linux kernel's own release model sorts its output into four categories, "Prepatch (or '-rc') kernels, Mainline, Stable, and Long Term Stable" (Linux Foundation 2020, 17), and the stable stream is the trunk the whole industry grows from: "These stable updates are the base from which most distributor kernels are made" (Corbet and Kroah-Hartman 2016, 7). Distributions extend the same gradient to everything they ship. At the fast end are the rolling and unstable lanes, where a new upstream release can land within days and the users are developers and enthusiasts who accept the risk. Behind them sit testing lanes, where the next numbered release is staged and shaken out. At the slow end are the stable and enterprise releases, the ones running banks and hospitals and the machines nobody wants surprised, which change reluctantly and on a published calendar. New code enters at the fast end and earns its way toward the slow end. Which

lane a piece of code has reached is, for most practical purposes, what its danger means, and the entire XZ Utils incident is a story about lanes.

Seen from the user's side, the apparatus is a convenience so dependable it disappears. Seen from an attacker's side, it is something else. A 2020 study of malicious packages stated the inversion in one sentence: "From an attacker's point of view, package repositories represent a reliable and scalable malware distribution channel" (Ohm et al. 2020, 16). The industry's own self-descriptions make the same point without meaning to, calling artifact repositories the "App Stores" of modern software development and "critical points of trust in every developer's workflow" (Alpha-Omega 2025a, 15). Institutional doctrine even has a name for the move. A supply chain attack, in the definition of the European Union Agency for Cybersecurity (ENISA), "is a combination of at least two attacks. The first attack is on a supplier that is then used to attack the target" (European Union Agency for Cybersecurity (ENISA) 2021, 6). The definition imagines a supplier compromised from outside. What happened here was stranger and cheaper. There was no break-in. The operator had spent two years becoming the supplier, and the campaign began already inside the trust boundary the definition assumes must be breached.

The power of a trusted delivery channel had a recent demonstration. NotPetya, the self-propagating catastrophe an earlier chapter set beside this one, entered the world through an update mechanism: the attackers, in Andy Greenberg's reconstruction, "hijacked the company's update servers" of M.E.Doc, a Ukrainian tax-accounting package, so that machines installing a routine update received the payload with the vendor's own blessing (Greenberg 2019). Ben Buchanan drew the structural lesson: "It was the software's reach, rather than its function, that made MeDoc an ideal vehicle for their attack" (Buchanan 2020). The comparison must be held in proportion: M.E.Doc was a commercial product whose servers were seized from outside, and the XZ Utils payload never detonated anywhere. But the geometry rhymes. An unglamorous, single-purpose piece of software becomes strategically decisive because of how many systems sit downstream of it, and a compression library wired beneath the world's Linux

systems offered reach of exactly that kind.

The reach had a precise and almost accidental shape, traced in an earlier chapter. Upstream OpenSSH, the project that writes the login server, does not use the compression library at all. Andres Freund put the chain in three clauses in the disclosure email: "openssh does not directly use liblzma. However debian and several other distributions patch openssh to support systemd notification, and libsystemd does depend on lzma" (Freund 2024b). In plain terms: several large distributions modify the login server they ship so that systemd, the program that supervises services on most modern Linux systems, can be told when the server has finished starting; that modification links sshd against the libsystemd helper library; and libsystemd, for its own unrelated reasons, can read compressed data, so it pulls in liblzma. No upstream developer designed a path from a compression library into the login machinery. The path was assembled downstream, by packaging decisions, which is to say that the attack surface itself was a product of the release train. Freund, retelling the discovery later, kept the point and its limit together: "So only that indirect loading of the library would actually cause the backdoor to be active. So it's very indirect" (Freund and Roccia 2024).

The indirection was a bound as well as a route, and one distribution proves it. Gentoo, which builds packages from source on the user's own machine, never carried the patch: "In Gentoo, we don't patch net-misc/openssh with systemd-notify support which means liblzma, at least in the normal case, doesn't get loaded into the sshd process," the Gentoo developer Sam James explained in the project's incident bug (James 2024a, comment 3). Same upstream code, same poisoned releases, different train: no path to the login server at all. Exposure was a property of distribution mechanics, not of the code alone.

A release that nobody packages reaches nobody, and the operation did not leave the packaging to chance. The operator tagged and released 5.6.0 in late February (Cox 2024, 2024-02-24), then updated the backdoor files and released 5.6.1 in early March (Cox 2024, 2024-03-09). "In the following weeks," Dan Goodin summarized in the week of the disclosure, "Tan or others

appealed to developers of Ubuntu, Red Hat, and Debian to merge the updates into their OSes" (Goodin 2024). The "or others" is the record's honesty: who sat at which keyboard remains unresolved. The timing, by the reading of the software-engineering study that later reconstructed the campaign, was synchronized to the calendar: the operator "waited until a month before the official RedHat release and close to Debian release to push his attack into the repository during a period of high project activity" (Przymus and Durieux 2025, 95), a reconstruction rather than a documented intention, but one consistent with the dated record of what happened next.

The main stage was Debian, and the position was undefended in a specific, institutional way. Every Debian package has a named maintainer responsible for shepherding upstream releases into the archive, and the `xz` package's maintainer had drifted away from it years earlier; the package survived on the goodwill of a developer who was not its maintainer, Sebastian Andrzej Siewior, working through a procedure whose name says it plainly: the Non-Maintainer Upload, or NMU. "I NMU maintained it for the last few years," Siewior wrote that March, adding that he was considering taking the package over officially (Jansen 2024, msg #32, 2024-03-27). The condition should sound familiar. Upstream, one volunteer in Finland with little backup; downstream, an absent maintainer and a stand-in doing unowned work. The same vacancy, stacked twice in the same supply chain, and the campaign had already exploited the first instance.

Into that vacancy, in late March, came a request. A Debian bug titled "xz-utils: New upstream version available," asking the distribution to import the backdoored `5.6.1`, was filed under the name "Hans Jansen" (Jansen 2024), a name that had surfaced once before, in mid-2023, contributing groundwork for the very optimization that later hid the backdoor's hook (Boehs 2024). Evan Boehs, tracing the accounts afterward, found the request had been opened the same week the "Hans Jansen" Debian account was created, and that the account had seeded "a few similar 'update' requests in various low-traffic repositories to build credibility, after asking for this one" (Boehs 2024). A supporting chorus

materialized: "Several other, suspicious, anonymous name+number accounts with little former activity also push for its inclusion, including `misoeater91` and `krygorin4545`. `krygorin4545`'s PGP key was made 2 days before joining the discussion" (Boehs 2024), a cryptographic identity minted for the occasion. It was the sock-puppet method of the pressure campaign two years earlier, pointed now at a distribution instead of a maintainer.

The push met friction, and the friction is the immune system, caught in the record. Thorsten Glaser, a Debian developer, objected on procedural instinct: "Very much *not* a fan of NMUs doing large changes such as new upstream versions" (Jansen 2024, msg #27, 2024-03-26). In the same message he asked the question underneath the whole affair, addressed to the absent maintainer: "what's up with the maintenance of `xz-utils`? . . . are you active? Are you well?" (Jansen 2024, msg #27, 2024-03-26). Ordinary suspicion of an irregular upload, and ordinary concern for a colleague: nothing in the objection required knowing about the backdoor. The procedure itself was generating resistance, which is what procedures are for.

The train was already moving. A non-maintainer upload importing `5.6.1` had been prepared the day before Glaser wrote, its changelog announcing "Non-maintainer upload" and "Import 5.6.1" (Jansen 2024, msg #5, 2024-03-25), and on March 27, 2024, the bug closed with the version's arrival in `unstable`, Debian's fast lane: "We believe that the bug you reported is fixed in the latest version of `xz-utils`, which is due to be installed in the Debian FTP archive" (Jansen 2024, msg #42, 2024-03-27). The upload that carried it was not the work of any sock puppet. It was Siewior's, and its changelog announced, alongside the import, "Takeover maintenance of the package" (Jansen 2024, msg #42, 2024-03-27): a conscientious developer formally adopting an orphaned package, fixing real reported issues, doing exactly what the project would have wanted done, and carrying the backdoor into the archive in the same gesture. Forty-eight hours later Freund's email reversed it. The release train converts trust into infrastructure, and it cannot tell, because it was never built to tell, whether the trust has been captured.

Debian was not the only channel being worked. *The Intercept* reported a Red Hat employee describing how "Jia Tan" had personally lobbied him to help get the compromised `xz` releases into Fedora (Mazurov 2024); a Fedora contributor, in Boehs's reconstruction, remembered the sales pitch as "great new features" (Boehs 2024). "Jia Tan also attempted to get it into Ubuntu days before the beta freeze," Boehs records (Boehs 2024), the deadline after which Ubuntu's next release stops accepting new versions. Kevin Beaumont noted a further bid pointing at the kernel itself: "a request was opened to make the threat actor a Linux kernel module maintainer for XZ Embedded" (Beaumont 2024). And the quiet result of all of it, before the catch: "These changes were committed to Github back in February, and made their way into test releases of Debian, Fedora and Kali Linux. Nobody noticed the problem" (Beaumont 2024).

When the catch came, the first descriptions reached for totality. "With a library this widely used, the severity of this vulnerability poses a threat to the entire Linux ecosystem," the Kali Linux project wrote (Kali Linux 2024), and, as a statement about the library's reach, that was not wrong. But the dated record the distributions published over the following days draws a much more precise shape. The phrase "every Linux system" has to be bounded by version, by distribution, by build path, and above all by lane.

Debian's security advisory did the bounding in its first breath: "Right now no Debian stable versions are known to be affected. Compromised packages were part of the Debian testing, unstable and experimental distributions, with versions ranging from `5.5.1alpha-0.1` (uploaded on 2024-02-01), up to and including `5.6.1-1`" (Bonaccorso 2024). Two facts sit in that sentence. The compromise had been flowing into Debian's fast lanes for nearly two months, since the first of February: the earlier uploads had entered by the ordinary route, and the late-March campaign was a race to put the updated `5.6.1` in behind them, while `5.6.0`'s `valgrind` noise was drawing exactly the attention the operation could not afford. "The race is on," Russ Cox's timeline notes on March 4, "to fix this before the Linux distributions dig too deeply" (Cox 2024, 2024-03-04). And the compromise never touched the stable releases that run production

machines. The fix was a reversion to the known-good upstream `5.4.5`, shipped under the version string `5.6.1+really5.4.5-1` (Bonaccorso 2024): Debian's house idiom for the same defensive craft openSUSE used in the same week, a number that sorts as an upgrade so the update machinery will accept it, with the `+really` confessing what is actually inside.

Red Hat's advisory drew the same line at its own boundary: "No versions of Red Hat Enterprise Linux (RHEL) are affected by this CVE" (Red Hat 2024). On the community side, "the packages are only present in Fedora 40 and Fedora Rawhide within the Red Hat community ecosystem" (Red Hat 2024), and the advisory paused to explain what Rawhide is, "the development distribution of Fedora Linux," the always-moving stream that becomes the next numbered release (Red Hat 2024). Red Hat's own retrospective made the channel boundary explicit: "While no Fedora stable builds were affected, Fedora 40 beta nightly builds, the leading edge of Linux innovation, were affected" (Freire 2024). The payload had reached the edge and only the edge.

The rest of the map repeats the pattern with different dates. openSUSE: "Our rolling release distribution openSUSE Tumbleweed and openSUSE MicroOS included this version between March 7 and March 28" (Meissner 2024), a three-week window in the fast lane, while the same vendor's fixed releases never inherited it because "SUSE Linux Enterprise and openSUSE Leap are built in isolation from openSUSE Tumbleweed" (Meissner 2024), a statement that is at once an accurate technical fact and a vendor reassuring its customers. Kali Linux, the security-testing distribution built on Debian: "The impact of this vulnerability affected Kali between March 26th to March 29th" (Kali Linux 2024), roughly seventy-two hours, and its remedy arrived down the same rails as the compromise had: "It has already been patched in Debian, and therefore, Kali Linux" (Kali Linux 2024). The train that delivered the backdoor delivered the fix. Even on Docker Hub, the public registry of prebuilt system images from which containerized deployments are assembled, the payload arrived only from Debian's fast lanes: it "was only found in development tracks of Debian, not any releases," as a maintainer of Debian's official images later put

it (Haruyama 2025, comment 2025-08-13). And the formal record bounds the upstream side to exactly two releases: in the CVE catalog, `xz` is affected at `5.6.0` and `5.6.1`, and everything else is unaffected by default (CVE Program 2024).

Two things were true at once. "The backdoor . . . affected mostly a development version, so it was not widely deployed when Andres found the backdoor," the security researcher Thomas Roccia told the DEF CON audience (Roccia 2024, 6:24); "Do I need to panic? No," Beaumont advised the enterprises checking their estates (Beaumont 2024). And: "Malicious updates made to a ubiquitous tool were a few weeks away from going mainstream," ran the subheading on Goodin's report (Goodin 2024), a judgment Luca Boccassi, a Debian and `systemd` maintainer, made specific on disclosure day: "This was caught before it got in any stable release of any distribution, it's only in development/testing. The only exception is SUSE Tumbleweed, because it's rolling. Just a few weeks of delay and it would have been part of the new Fedora 40 release and the new Ubuntu LTS 24.04 release" (Corbet 2024, bluca comment, 2024-03-29). For calibration, the going rate for detection in one 2015–2019 dataset of malicious packages was an average of 209 days from publication to public report, with a worst case of 1,216 days for a package quietly taken over after abandonment (Ohm et al. 2020, 11). This one was caught in weeks, before it had reached the slowest lanes. A genuine emergency, and a bounded one: the urgency and the bound come from the same machinery.

The map of where the packages went still overstates the danger, because having the affected version present was not the same as the exploit taking effect. The payload was choosy about where it would even build itself. "The attack appears to specifically target `amd64` systems running `glibc` on Debian or Red Hat derived distributions, although other systems may also be vulnerable," the Przymus and Durieux study records (Przymus and Durieux 2025, 93), and James had described the mechanism in the first days: the injection script "looks for `.deb` and `.rpm` specific files/environment variables in the build environment" (James 2024a, comment 3), arming itself only while being assembled into the

package formats Debian- and Red Hat-style distributions ship. After that it still needed the `systemd`-patched `sshd` at runtime. Three conditions, stacked.

The stacking produced cases of presence without fire. Red Hat's advisory, in a correction issued a day after its first read, determined that Fedora 40 beta "does contain two affected versions of xz libraries" yet "does not appear to be affected by the actual malware exploit" (Red Hat 2024): the poisoned bottle on the shelf, the cork never drawn, and the day-later correction itself a small instance of the rolling, self-revising response of those weeks. Gentoo shipped the affected `xz-utils` inside a `stage3` archive, the base-system bundle from which a Gentoo machine is first installed, built on March 24 (James 2024a, comment 12), and was still not impacted, because its build path never armed the payload and its `sshd` never loaded the library (James 2024a, comment 3). The version's presence is a fact about distribution. The exploit's firing is a fact about build conditions. The two came apart everywhere outside the targeted path.

Canonical, Ubuntu's publisher, drew the deepest version of that distinction, and no other vendor went so far. Rather than revert a package, it distrusted its own factory: it decided "to remove and rebuild all binary packages that had been built for Noble Numbat after the `CVE-2024-3094` code was committed to `xz-utils` (February 26th), on newly provisioned build environments" (Zemczak 2024), the February date being Canonical's own fixing of the poisoned commit. The reasoning: any binary compiled in an environment that might have contained the backdoored `xz` at build time was suspect, whether or not that binary contained the library, because a poisoned build tool can taint what it builds. The stated aim was affirmative: "confidence that no binary in our builds could have been affected by this emerging threat" (Zemczak 2024). Noble Numbat, the working name of the Ubuntu 24.04 LTS release then weeks away, was itself still a pre-release; the stable channel was never in question; uncertainty about build integrity alone justified throwing away the builder and starting clean.

Getting the thing out again was harder than letting it in had been, and the difficulty lived in the same machinery. The Debian bug demanding a response was titled, plainly, "revert to version that does not contain changes by bad actor,"

and its author, the developer Joey Hess, stated in it the problem that made a simple downgrade insufficient: "Reverting the backdoored version to a previous version is not sufficient to know that Jia Tan has not hidden other backdoors in it. Version 5.4.5 still contains the majority of those commits" (Hess 2024, msg #5). The damage was epistemic, not just technical. Once a trusted contributor stands exposed as hostile, no version number restores confidence, because every commit that passed through his hands, two and a half years of them, is now a question, a contamination problem a later chapter returns to.

Even the mechanical half was fraught. Reverting that far back, the Debian developer Aurélien Jarno warned, "will break packages that use new symbols introduced since then"; his quick survey of what would break named `dpkg`, `erofs-utils`, and `kmod`, and he added the operational understatement: "Having `dpkg` in that list means that such downgrade has to be planned carefully" (Hess 2024, msg #10). Symbols, here, are the named entry points a shared library offers; programs built against newer ones will not run with an older library. And `dpkg` is the tool that unpacks and installs every Debian package. Rip out `xz` carelessly and the system could lose the ability to install anything, including the fix. That is the inertia of the installed base, rendered as a bug comment: the library could not simply be removed, because the means of removing things depended on it. The condition is general and it compounds, as the Census II report observed of aging packages whose support thins while their deployment does not: they "become more likely to break with each passing day without the guarantee of support on-hand to provide fixes" (Nagle, Wilkerson, et al. 2020, 30).

The same thread holds the response's most concrete act of repair. Within a day, Hess had built an exit: "I have prepared a git repository that is a fork of `xz` from the point I identified before the attacker(s) did anything to it. In my fork, I have renamed `liblzma` to `liblzmaunscathed`. That allows it to be installed alongside current `dpkg` without breaking `dpkg` with an old version of `liblzma`. . . . The goal is not to take over from `xz` upstream, but to get the possibly backdoored code off of production systems ASAP" (Hess 2024, msg #62). One

volunteer, overnight, reconstructed a clean line of the library from before the operator's first touch and renamed it so it could stand beside the installed base without breaking it: a stopgap, offered as a stopgap, and a measure of what the distributed immune response could improvise while the institutions were still drafting advisories.

The full revert lost anyway, and the reasons it lost are the machinery's gravity made visible. Siewior, weighing how far back to go, named what a deep rollback would cost: the `5.4.x` series had threaded decompression he wanted to keep, the `5.6.x` series had a faster decompressor, and, decisively, "I want to stay on an official upstream release which is also used by other distros" (Hess 2024, msg #142). Features, performance, and alignment with what everyone else ships: the gravity that converts upstream code into infrastructure is exactly the gravity that resists un-converting it. Meanwhile every distribution encoded its remedy differently. Debian shipped `5.6.1+really5.4.5-1`; Alpine kept the `5.6.1` label and switched to building from git archives instead of release tarballs; Gentoo deleted its masked `5.6.1` package outright; and users comparing notes got lost in the strings, as one admitted in the Gentoo bug: "First, I saw `5.6.1` and did not pay attention to the next part..." (James 2024a, comment 36). The United States government's own advisory pointed at yet another target, recommending a downgrade to `5.4.6`, a version uncoordinated with the fixes the distributions actually shipped (Cybersecurity and Infrastructure Security Agency 2024a). After the train had run, a version number alone could no longer tell anyone whether they were safe.

Two further facts bound the story, one about the attack and one about the cure. The first: the path was fragile, and ordinary maintenance nearly closed it. On February 29, 2024, Cox's timeline records, a contributor known as `teknoraver` "sends pull request to stop linking `liblzma` into `libsystemd`. It appears that this would have defeated the attack" (Cox 2024, 2024-02-29): a routine dependency-slimming change, nothing to do with security, that would have severed the only road from the compression library into the login server. Beaumont, who examined the same sequence, noted that "the fix for this was already

in train before the XZ issue was highlighted" but "hadn't yet rolled out into a release of systemd" (Beaumont 2024), and he ventured a reading of the operator's late hurry: "I believe there's a good chance the threat actor realised this, and began rapidly accelerated development and deployment" (Beaumont 2024), the public bug reports and distribution lobbying of March as a race against a closing window. Cox passes the hypothesis along explicitly as speculation, and speculation is what it remains: a plausible reconstruction of tempo, unproven, and a question that returns with the hunt for the operator. What is documented is the contingency itself. The corridor the operation needed was being walled up by accident, by maintenance, while the campaign to ship through it accelerated.

The second fact runs the other way. The friction that makes the stable lanes slow is real, and it is usually told as a defect. Nicole Perlroth, reporting on critical-infrastructure patching, recorded that "Automated patches were still big no-nos inside critical infrastructure networks," updates needing high-level approval and narrow maintenance windows, with even urgent fixes deferred for fear of disruption (Perlroth 2021). She was explaining why known holes stay open for months. But the same gating ran the other way in March 2024: the conservatism that slows every fix is also what had kept the backdoor out of the lanes that matter. The friction that slows the cure slowed the disease. The train's slowness is not a virtue or a vice; it is a property, and that month it cut in the defenders' favor.

What, then, does the machinery actually verify? At every coupling of the train there are signatures: the maintainer signs the release tarball, the distribution signs its packages, the mirrors serve signed indexes. Signing, as a large-scale study of the practice defines it, uses public-key cryptography to "bind an identity (e.g., a package maintainer's private key) to an artifact (e.g., a version of a package)," so that anyone "can verify whether the artifact was indeed produced by the maintainer" (Schorlemmer et al. 2024). The backdoored tarballs passed that test perfectly. They were produced by the authorized signer; the signer was the operator. And the chain checks in one direction only, a structural habit a study of industry practitioners summarized as "a unidirectional

use of Software signing, establishing trust for outputs while placing lower expectations on the provenance of inputs" (Kalu et al. 2025, 93). Each link signs what it ships and trusts what it receives: a supply chain of signed outputs that are nobody's verified inputs.

The closed-source world had already demonstrated the consequence at scale. The SolarWinds implant, an earlier chapter's comparison, rode inside what its discoverers described as "a SolarWinds digitally-signed component of the Orion software framework that contains a backdoor" (FireEye 2020): the malicious update verified flawlessly, because the compromise sat upstream of the signature. The disanalogy is real, a vendor's compiled binary in a commercial update channel against a build-time injection into an open-source release tarball, but the lesson is identical. A signature authenticates an artifact and the identity behind it. It says nothing about the good faith of that identity. Where the signer is the attacker, the signature is the attack's passport.

The law has since internalized the threat with striking precision. The European Union's Cyber Resilience Act, defining the severe incidents manufacturers must report, gives as its example "a situation where an attacker has successfully introduced malicious code into the release channel via which the manufacturer releases security updates to users" (European Parliament and Council of the European Union 2024, rec. 68), which is very nearly a definition of the XZ Utils operation; the regulation was not written about this incident, but it names exactly this seam. The same act mandates the software bill of materials, or SBOM, an ingredients list for software, "in a commonly used and machine-readable format covering at the very least the top-level dependencies of the products" (European Parliament and Council of the European Union 2024, Annex I, Part II(1)). The floor in that phrase is the limit: top-level dependencies are the ones a product declares directly. The link that mattered was not `sshd` declaring `liblzma`; it was a transitive, distribution-specific path through a patch and `libsystemd`, below the floor the mandate names.

The U.S. guidance that defined the SBOM grasped the deeper seam too: the data can be collected from source or from the build, and the two diverge, since "a

compiler may pull in a slightly different version of a component than what was expected from the source" (National Telecommunications and Information Administration 2021, 14), so consumers "should try to obtain it from the instance of the build" (National Telecommunications and Information Administration 2021, 18). That recommendation points at precisely the gap the operation exploited, the distance between the inspectable source and the shipped artifact. But the honest middle ground runs in both directions: a build-instance inventory of the poisoned release would have faithfully recorded a correctly named component, from a trusted project, signed by an authorized maintainer. An SBOM is an inventory, not a verdict. It would not have caught this, and it is not therefore worthless; it answers "what is in the box," not "should the box have been trusted."

And the train, it turns out, does not fully un-ship. In 2025, more than a year after the reverts, researchers at the firm Binarly went looking for the backdoor on Docker Hub. The backdoored Debian packages of March 2024 had been captured into base images, and the base images had been built upon: "other images have been built on top of these infected base images, making them transitively infected" (Binarly REsearch 2025). They counted "more than 35 images that ship with the backdoor," and bounded their own number in the same breath: they had scanned only a small portion, only Debian-based images, only to the second order (Binarly REsearch 2025). With "nearly 12 million repositories" on the registry, an exhaustive scan "would be infeasible" (Binarly REsearch 2025). The figure is a floor, not a census, and the affected images sit in development tracks, not in anything a stable system pulls by default (Haruyama 2025, comment 2025-08-13). But the installed base does not run backward. What the release machinery ships, it cannot fully recall; it can only ship something newer on top.

That is the problem the catch preempted, and the era's other great dependency fire shows what it looks like realized. When the Log4j vulnerability detonated in 2021, the U.S. Cyber Safety Review Board found that "there is no comprehensive 'customer list' for Log4j, or even a list of where it is integrated

as a sub-system" (Cyber Safety Review Board 2022, iv); the library was "routinely embedded in other software components, often unknown at later levels of integration or system operation" (Cyber Safety Review Board 2022, 11). The two cases sit at different levels: Log4j was an accidental flaw whose crisis was a post-disclosure inventory problem, while the XZ Utils backdoor was an authored operation concealed before disclosure. But had `5.6.1` ridden the spring release calendar into Fedora 40 and Ubuntu 24.04 LTS, then sat there quietly for a year, the world would have faced Log4j's question, where is it, with a payload built by an adversary instead of a bug, and the honest answer would have been the same: no one keeps that list.

Step back from the dates and the version strings and the machinery resolves into a single image. Every gate the payload passed was working as designed. The maintainer's signature on the tarball was valid. The bug asking Debian for the new version followed the form. The upload that imported it was a model act of package stewardship. The lanes carried the code at exactly their rated speeds, fast at the edge, slow toward the center, and the slowness bought the time in which one engineer's curiosity could matter. The train neither failed nor saved anyone; it moved what the upstream gave it, as it does every day, for thousands of projects, almost always to everyone's benefit. Its one assumption, repeated at every coupling, is that the upstream is sound: that behind every release tarball stands a project, and behind every project stands someone fit to be trusted with all of it. For XZ Utils, the someone had been one unpaid volunteer in Finland, and then, by handover, his attacker. How many of the other thousands of upstreams look like that, what the cathedral under the train actually is and who keeps it standing, is the next question, and it has been waiting since 1991.

Chapter 11

The Cathedral Nobody Built

Pull back far enough from the release train and the ground it runs across comes into view. Beneath the distributions and their lanes, beneath the login servers and the helper libraries, stands the accumulated work of several decades: the kernel, the compilers, the compression and cryptography and parsing libraries, the thousands of small components every modern computer assumes the way a building assumes its foundation. Seen from a distance it has the proportions of a cathedral: vast, old, load-bearing, the product of generations of labor. The resemblance fails at exactly one point. A cathedral has an architect, a plan, a patron, a ledger somewhere recording who is responsible for the roof. This structure has none of those things. Bowker and Star, the sociologists of infrastructure the previous chapter consulted, wrote the honest caption for it: "In the past 100 years, people in all lines of work have jointly constructed an incredible, interlocking set of categories, standards, and means for interoperating infrastructural technologies. We hardly know what we have built. No one is in control of infrastructure; no one has the power centrally to change it" (Bowker and Star 1999, 319).

What such a structure is like to live under, they also described. Standards and infrastructures, "however imbricated in our lives, are ordinarily invisible," and "they may become more visible, especially when they break down or be-

come objects of contention" (Bowker and Star 1999, 2–3). For the better part of two decades `xz` was what they were describing: a piece of the floor, walked on by everything, looked at by no one, visible to the world for exactly one weekend in March 2024, when it nearly broke. Nicole Perlroth, after a decade of reporting on the consequences, put the general condition in one sentence: "In our brave new world, these unglamorous open-source protocols have become critical infrastructure and we barely bothered to notice" (Perlroth 2021).

The people who tried to map the structure reached for a wartime analogy. The 2020 Census II report, the Linux Foundation and Harvard effort to identify the most widely used free and open-source components, observed that critical components "may not always be the most remarkable or the most visible," and recalled that when Allied planners in the Second World War went looking for the target whose loss would propagate furthest through the German war machine, they settled on ball-bearing factories: commonplace products, crucial to nearly everything built around them (Nagle, Wilkerson, et al. 2020, 11). The analogy and its martial register are the report's; the structural claim is sound, and the report built its whole method around finding what it called the "hidden keystones" of the ecosystem, the small packages no audit lists because nothing depends on them directly and everything depends on them transitively (Nagle, Wilkerson, et al. 2020, 19).

The keystone position is not rare, and it is not obscure to attackers. A single open-source package "may be required by several thousands of open source software projects," a 2020 review of supply-chain attacks observed, which is exactly what makes the layer an efficient target (Ohm et al. 2020, 2); and the chains run deep as well as wide, most of them three or four layers, by one 2025 measurement of Ubuntu's package graph (Deng et al. 2025, 6210). `liblzma` beneath `libsystemd` beneath `sshd` is what a hidden keystone looks like from above. From below, as the ball-bearing analogy does not say, the factory had a staff of one.

How the structure came to exist resists its own legend, and the best vaccine against the legend is the founding document. On August 25, 1991, Linus

Torvalds, then a twenty-one-year-old student in Helsinki, posted to the Usenet newsgroup `comp.os.minix`: "Hello everybody out there using minix - I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu) for 386(486) AT clones" (Torvalds 1991). The post's own postscript bounded the ambition further: "It is NOT protable (uses 386 task switching etc), and it probably never will support anything other than AT-harddisks, as that's all I have :-(" (Torvalds 1991). The misspelling and the emoticon are the document's. Read straight, the announcement records contingency, not destiny: the author of the future substrate did not plan the substrate, and on the evidence of his own forecast did not believe in it.

The next day, answering a question in the same thread, he added the sentence that mattered most and sounded least like it: "It won't be ready for distribution for a couple of months. Even then it probably won't be able to do much more than minix, and much less in some respects. It will be free though (probably under gnu-license or similar)" (Torvalds 1991). Two hedges, "probably" and "or similar," and inside them the licensing instinct on which everything afterward depends. The volunteer fingerprints stayed on the movement's very definition: when "open source" later acquired a formal standard, it was not issued by a vendor or a standards body; the Open Source Definition "was originally derived from the Debian Free Software Guidelines (DFSG)," one volunteer distribution's internal rules for what it would agree to ship (Open Source Initiative 2007). Debian wrote the definition of openness. Debian was also, years later, one of the distributions whose development channels carried the backdoor closest to stable release.

What the hobby became is best measured in its own units. "The original 0.01 kernel was a mere 10,000 lines of code; now it grows by more than that every few days," the kernel's twenty-fifth-anniversary report noted in 2016 (Corbet and Kroah-Hartman 2016, 17). By the Linux Foundation's 2020 accounting, the 88 files and 10,239 lines of `linux-0.01.tar.Z`, which ran on a single hardware architecture, had become 69,325 files and more than 28 million lines running on over 30 (Linux Foundation 2020, 4), deployed by then in "products where

security and safety-critical considerations are essential, from medical devices, to autonomous vehicles, and to spacecraft" (Linux Foundation 2020, 21). By the time the XZ Utils operation unfolded, that substrate was no longer only inside servers and consumer devices; it sat beneath cloud fleets, phones, and the compute stacks being assembled for AI. Torvalds, in the memoir, called the result "the largest collaborative project in the history of the world" (Torvalds and Diamond 2001, 225), a sentence written in 2001 about a project that has only grown since. And in the same pages he predicted the invisibility: "And where is Linux itself, and open source generally, in all this? You won't even know. It will be inside those Sony machines. You'll never see it, you'll never know it, but it's there, making it all run" (Torvalds and Diamond 2001, 223). Bowker and Star's theory of invisible infrastructure, stated in the first person, in advance, by the man whose bedroom project was becoming the leading instance.

The shape of that growth had a name before the danger did. Jonathan Zittrain, a legal scholar writing in 2008 about open platforms in general, described a recurring pattern: it "begins with a generative platform that invites contributions from anyone who cares to make them," the contributions "start among amateurs, who participate more for fun and whimsy than for profit," and in the last stage "the generative features that invite contribution and that worked so well to propel the first stage of innovation begin to invite trouble and reconsideration, as the power of openness to third-party contribution destabilizes its first set of gains" (Zittrain 2008, 18). As the pattern ran, free code displaced the purchased kind: "The phenomenon of open source software, distributed at no cost over the Internet, has displaced many of those earlier software purchases," Russ Cox observed in 2019 (Cox 2019, 37), and the furniture of the purchase relationship, the contracts, the warranties, the someone to call, did not survive the substitution. Eghbal compressed the whole trajectory into a sentence: "Many open source projects are experiencing a difficult transition from selfless creative pursuit to critical public infrastructure" (Eghbal 2016, 65). The dependency grew faster than anything that might have supported the people underneath it.

"The community" is the phrase that stands where that support should be,

and it earns its quotation marks. The picture the phrase conjures, crowds of capable volunteers swarming over shared code, describes some projects, the kernel above all. It does not describe the layer where the risk lives. Eghbal, sorting widely used projects by who actually shows up, found that many of the most depended-upon ones are what she calls stadiums: "Stadiums are projects with low contributor growth and high user growth. . . . As a result, they tend to be powered by one or a few developers" (Eghbal 2020, 63). One performer, or a few, down on the field; a vast crowd in the stands, consuming the performance without ever joining it. And the stadium libraries sit "below the application layer, yet are still widely depended upon" (Eghbal 2020, 105), beneath the products users can see, which is why the stands never empty and the field never fills.

xz was the textbook case, and not by accident. The software-engineering study that later reconstructed the campaign stated the targeting logic plainly: "XZ Utils was appealing to attackers due to its status as a low-traffic repository managed by a single developer with a small community (around 10 active members on the project's IRC channel)" (Przymus and Durieux 2025, 93). Ten people in the chat channel of a library embedded in major Linux distributions as basic plumbing. The condition an earlier chapter traced through Lasse Collin's years of solo maintenance was not a weakness the operator stumbled over; it was the selection criterion, the property that made this library, among all libraries, worth two years of patient work. A bus factor of one was the specification.

Nor is the thinness confined to the basement of the long tail. The person at the apex of the model said so himself, in an interview circulating after the disclosure: "It is worth really pointing out how unusual the kernel is as an open source project. A lot of open source projects, even very central ones, are basically run by one or two or three people" (Mastery Learning 2024, 6:26). The kernel is the exception because it industrialized: it spread "the responsibility for code review and integration across 100 or more maintainers," as the anniversary report put it, so that no one person's limits would bound the project (Corbet and Kroah-Hartman 2016, 16). The solution exists, is well understood, and was

applied where the labor and the money were. In the long tail, it never arrived.

The measurements agree with him. A 2024 study commissioned by Germany's Sovereign Tech Fund, examining 3,707 active repositories, reported its headline finding without cushioning: "The majority of open-source projects are in a state of decline, following a similar trajectory: a decline in maintenance and activity over time" (Ellis and Bollampalli 2024, 19). One of the maintainers its authors interviewed said what the regression tables cannot: "It is almost scary that we as a community…are relying on a few people to maintain such core pieces of infrastructure" (Ellis and Bollampalli 2024, 21). The official record agrees in a government's voice: reviewing the Log4j emergency, the U.S. Cyber Safety Review Board pointed to "the security risks unique to the thinly-resourced, volunteer-based open source community," a community "not adequately resourced to ensure that code is developed pursuant to industry-recognized secure coding practices and audited by experts" (Cyber Safety Review Board 2022, v). The board was diagnosing the conditions for accident, neglect breeding bugs, which is not how the backdoor entered `xz`; but the under-resourcing it names is the same soil. Even Eric Raymond, the movement's most confident early theorist, had conceded the category by 2019, in a phrase Eghbal preserves: there are "Load-Bearing Internet People," each one "a person who maintains the software for a critical Internet service or library, and has to do it without organizational support or a budget backing him up" (Eghbal 2020, 198).

On the weekend of the disclosure, the condition described itself from inside. In the Debian thread demanding that the backdoored versions be purged, one developer asked the question that outlives the incident: "Will we want to keep in the archive an unmaintained low-level library - low-level as in, susceptible of getting pulled as a dependency in lots of places - and rely on it for components such as `dpkg`?" (Hess 2024, msg #40). The question was genuine, and the thread records no settled answer. In the discussion under LWN's first report of the backdoor, a longtime Debian developer named the general case: "We have an enormous maintenance problem, and I'm not sure what slowing down and writing less code with more quality looks like in the face of that maintenance

problem" (Corbet 2024, rra comment, 2024-03-30). These are not the voices of a community discovering a gap. They are the voices of people who had known about the gap for years and had no lever long enough to close it.

The deeper trouble with "the community" is not that it is small. It is that the phrase implies an entity with hands: something that could notice a gap, allocate against it, relieve an unsupported maintainer. No such entity exists. Fogel's handbook, the practitioner's manual earlier chapters consulted, says it from the inside: "even the assumption that free software projects can be 'run' is a stretch. A free software project can be started, and it can be influenced by interested parties. But its assets cannot be made the property of any single owner," and as long as anyone anywhere cares to continue it, "it can never be unilaterally shut down. Everyone has infinite power; everyone has no power" (Fogel 2020, vi). What stands in place of management is the norm Moody recorded at the source, a movement with no formal hierarchy for handing out important tasks: "A kind of self-selection takes place instead: Anyone who cares enough about developing a particular program is welcome to try" (Moody 2001). Earlier chapters showed what that norm yields when the person who cares enough is an operative. The same norm, read structurally: nobody assigned `xz` to Collin, so nobody existed to notice that the assignment had become unsustainable, and nobody's job was to relieve him.

The corporations that built on the structure discovered the absence early, as a contracting problem. Siobhán O'Mahony, the management scholar who studied how open-source projects came to incorporate, recorded a Fortune 100 executive in the late 1990s confronting the Apache Project with the question: "How do I make a deal with a Web page?" (O'Mahony 2005, 396). The question was exact. "Communities are not legal actors," O'Mahony observed; they are "initiated and managed by a distributed group of individuals who do not share a common employer" (O'Mahony 2005, 395), and so they can hold no assets, sign no contracts, shield no volunteers. The nonprofit foundations that now dot the landscape were bolted on to supply the missing legal personhood, created "despite the fact that such formal structures are an anathema to the hacker ethos

of technical autonomy and meritocratic decision making" (O'Mahony 2005, 393). A foundation, in origin, is the adapter that lets a corporation make a deal with a web page. Whether the adapter can also maintain the web page is a separate question.

The correction can overshoot, and it matters that it not. The commons is not a formless crowd. Gabriella Coleman, the anthropologist who spent years inside the free-software world, found a built order, not a swarm: hackers, she wrote, are "committed to productive freedom," a term that "designates the institutions, legal devices, and moral codes that hackers have built in order to autonomously improve on their peers' work, refine their technical skills, and extend craftlike engineering traditions" (Coleman 2013, 3), with projects like Debian developing "complex codes for collaboration along with other ethical precepts that help guide technical production" (Coleman 2013, 209). The earlier chapters depended on exactly that orderliness: the operation succeeded by following Debian's procedures, not by evading them. "The community" is a myth only as a singular noun implying an authority. As a plural description of real institutions with rules, roles, histories, and values, it is accurate. The institutions are simply not resourced in proportion to what leans on them.

There is a precise name for what this structure is, and it comes with a literature. Elinor Ostrom, the political economist who would later share the Nobel Memorial Prize in Economic Sciences for the work, spent her career on commons: shared-resource arrangements governed, in her summary, by "institutions resembling neither the state nor the market," which have sustained their resources "with reasonable degrees of success over long periods of time" (Ostrom 1990, 1). At the heart of every commons problem she placed one temptation: "Whenever one person cannot be excluded from the benefits that others provide, each person is motivated not to contribute to the joint effort, but to free-ride on the efforts of others" (Ostrom 1990, 6). Open-source code is a good no one can be excluded from, so the temptation she names is the standing condition of the entire ecosystem; her own 1990 enumeration of shared-resource systems, written for irrigation canals and fisheries, already included "mainframe

computers" (Ostrom 1990, 30). Yochai Benkler, the legal scholar who carried the framework to the internet, called free software "the quintessential instance of commons-based peer production," many individuals contributing to a common project "with a variety of motivations" and sharing the result without anyone "asserting rights to exclude" (Benkler 2006, 63). Under every industrial-era assumption about volunteer projects, he noted, "this was a model that could not succeed. But it did" (Benkler 2006, 66).

The word "commons" has to be used honestly, though, because it hides a distinction the whole argument turns on. The classic tragedy of a commons is depletion: the pasture overgrazed, the fishery emptied, the resource subtracted by use. Code does not work that way, and Raymond built his optimism on the difference: in open source, use does not deplete the resource but improves it, as users fold their fixes back in; "the grass grows taller when it's grazed upon" (Raymond 2001, 125). On the code, he is right: `xz` was not worn down by the billions of systems that ran it. What use subtracts sits underneath the code, on the side of the problem Ostrom's framework calls provision, the producing and sustaining of the shared thing rather than the consuming of it (Ostrom 1990, 49). A maintainer's attention is depletable. The unpaid hours are depletable. Trust, as the preceding chapters showed in detail, is depletable, and was in fact the resource the operation consumed. The tragedy this commons risks is not the exhaustion of its grass. It is the exhaustion of its soil.

The oldest voice in the record had said so at the start. Stoll, in 1989, defending the era's "wealth of public-domain software" against those who scorned it, located the actual vulnerability in two sentences: "Viruses and logic bombs poison this communal well. People stop trusting public software, and eventually the sources of public software dry up" (Stoll 1989). Not the code, the well: what depletes is the willingness to share and to depend. And Ostrom, whose career was spent documenting commons that work, was equally clear that commons fail: "The models . . . are not wrong," she wrote of the tragedy theorists; where the enabling conditions are absent, the collapse arrives on schedule (Ostrom 1990, 183). A commons is governable, not self-correcting. Fogel drew the prac-

tical conclusion for this one: software now belongs with the "goods that everyone needs but no one needs to own," the roads and sewers and electric grids, and "Just as we expect road workers to be paid, we should expect software developers to be paid as well" (Fogel 2020, 67). What the XZ Utils operation consumed was precisely the part of the commons nobody had priced.

Why so little of this was visible in time has a history, and the history runs through the most influential essay the movement produced. Raymond's *The Cathedral and the Bazaar*, the founding self-description an earlier chapter quoted on reputation, set two images against each other: the cathedral, software raised the careful, closed, hierarchical way, and the bazaar, the style he found in Linux and could barely believe, "release early and often, delegate everything you can, be open to the point of promiscuity," a community resembling "a great babbling bazaar of differing agendas and approaches . . . out of which a coherent and stable system could seemingly emerge only by a succession of miracles" (Raymond 2001, 21–22). The essay's most consequential sentence was its security claim. The bazaar finds flaws through sheer multiplicity of attention, a principle Raymond put "less formally" as "Given enough eyeballs, all bugs are shallow," and named on the spot: "Linus's Law" (Raymond 2001, 30). With enough people reading the code, every flaw is obvious to someone.

The law was doubted almost as soon as it was venerated. Michael Cusumano, introducing the field's own scholarly anthology in 2005, asked: "how many 'eyeballs' actually view an average piece of open source code? Not as many as Eric Raymond would have us believe!" (Cusumano 2005, xii). The Census II authors, fifteen years later, treated the limit as settled: the maxim had brought real security where contributor bases were large, while "vulnerabilities in other widely-used projects with smaller contributor bases, like OpenSSL, can slip by unnoticed" (Nagle, Wilkerson, et al. 2020, 9). And Eghbal identified the mechanism that turns the shortfall toxic: the belief in many eyes "has created the opposite problem: people mistakenly believing that more people are reviewing open source software than actually are, when

in reality nobody is taking responsibility" (Eghbal 2016, 36). The law does not merely fall short in the long tail. Believed where it fails, it manufactures a diffusion of responsibility, a standing assumption that someone else is watching. That assumption is a door, and it is the door the operator walked through.

Heartbleed, the 2014 catastrophe in exactly such a project, forced the concession from the law's author: of OpenSSL in those years, Raymond told Perlroth, "there weren't any eyeballs" (Perlroth 2021). The XZ Utils operation then ran the experiment under adversarial conditions and returned the same result with a sharper edge. For more than two years the operator worked in the open, in a library at the base of every major Linux distribution, and the promised eyes never arrived: the commit history drew almost no qualified review, the decisive components shipped where reviewing eyes do not point, in the release tarballs rather than the repository, and the few eyes that did exist had been cultivated into trust by the operator himself. Even the refined form of the law that Raymond recounted to Moody, Torvalds's own emendation that enough eyeballs would at least leave every bug "characterized" (Moody 2001), found no purchase: for two years the backdoor was neither fixed nor characterized nor suspected. What ended it was not eyeballs. It was one engineer, off to the side, chasing half a second his benchmarks could not explain. Linus's Law is not false; where the eyeballs are real, it works as advertised. It is a law about the well-watched core, quietly assumed to govern the unwatched long tail, and the assumption was the vulnerability.

Raymond's metaphor deserves one more turn, because the decades have inverted it. The bazaar, left running for thirty years, did not stay a bazaar. Its stalls fused into load-bearing masonry; the improvised became the foundational; and what the world now stands on has the mass, the permanence, and the indispensability of the cathedral in his title, with none of a cathedral's apparatus: no architect, no blueprint, no guild of masons, no endowment for the roof. A cathedral nobody built, and therefore a cathedral nobody is paid to keep standing.

Institutions did eventually rise around the structure, and honesty about them requires both halves. Heartbleed produced the first wave: in 2014 the Linux Foundation founded the Core Infrastructure Initiative (CII), whose members provided funding and support for open-source projects "critical to global information infrastructure" (Nagle, Wilkerson, et al. 2020, 6), and the CII's census work later passed into the Open Source Security Foundation, the OpenSSF of earlier chapters (Nagle, Wheeler, et al. 2020, 4). By 2023 the United States government had adopted the framing outright: open-source software, CISA declared, is "a public good," one "supported by diverse and wide-ranging communities—which are composed of individual maintainers, non-profit software foundations, and corporate stewards" (Cybersecurity and Infrastructure Security Agency 2023, 3). The sentence's grammar sets the individual maintainer beside the foundation and the corporate steward, three coequal pillars. The resources behind the three words are not coequal, and nothing in the official framing says so.

The most direct attempt to fund the gap is Alpha-Omega, an OpenSSF project "established in February 2022 with a $5 million grant made jointly by Google and Microsoft" to pursue, in its own words, "direct maintainer engagement and expert analysis" (Alpha-Omega 2022, 3). The money is real and the work is real. In 2023 its grants "helped staff security teams within the Python Software Foundation, the Eclipse Foundation, the Rust Foundation, and OpenJS" (Alpha-Omega 2024, 3); by March 2026 the Linux Foundation could cite "over 70 grants totalling over $20M across major ecosystems, package registries, and individual projects" (Linux Foundation 2026). The fund's standing mission spans the whole spectrum, "from the largest global projects to the smallest but essential components maintained by individuals" (Alpha-Omega, n.d.). That last clause names the `xz` layer exactly. It is the promise to weigh.

Weighed, it shows a resolution limit, and the fund deserves credit for stating the limit in its own voice. "There is no shortage of critical projects," its 2023 report explains, "and we aren't convinced there's a way to quantifiably measure the criticality of projects below a certain level of granularity. Is Node.js more

or less critical to the open source ecosystem than Python? Is GCC more or less critical than React?" (Alpha-Omega 2024, 7). Consider what the hard cases are: Node.js, Python, GCC, React, every one a famous, foundation-backed project. The granularity at which the fund's sight dissolves sits well above the layer where a solo-maintained compression library lives. The same report is candid about the remainder, hoping for solutions "that will scale to the huge bodies of open source that we cannot address directly" (Alpha-Omega 2024, 8). And in October 2023 it stopped work on the Omega Toolchain, the automated effort that was its one at-scale attempt on the long tail (Alpha-Omega 2024, 15); the year's goal review stated the reasoning bluntly: "The long tail is a hard problem with both technical and human challenges. We are not a software engineering organization" (Alpha-Omega 2024, 12). Five months later, the long tail produced the XZ Utils disclosure.

Set the two records side by side. The leading fund: criticality cannot be measured "below a certain level of granularity." The reconstructed targeting logic: a low-traffic repository, a single developer, ten people in the channel (Przymus and Durieux 2025, 93). The operator selected precisely the layer the funder says it cannot reliably measure or address directly at scale. The thinness of that layer was documented, not hypothetical; in the 2020 contributor survey, barely a quarter of the projects represented had a security policy of any kind in place (Nagle, Wheeler, et al. 2020, 60). And the institutional response to the human side of the problem was real and characteristically scaled: the kernel community's 2023 maintainers' gathering gave a session to an outside psychologist on reducing stress and burnout by building psychological safety (Chance 2023), a sincere gesture, pitched at the individual's resilience rather than at the conditions producing the strain. The foundations are not a failure. They are an immune system that grew where the body could see itself, in the organs with names, budgets, and dashboards, and the operator attacked through the tissue below the resolution of its sight.

Two over-readings lie in wait, and the record blocks both. The first is the fall from a golden age, the story in which the commons was once whole and

recently broke. Raymond, of all people, dismissed it to Moody decades ago: "There's a certain school of historical interpretation that views today's free software as a return to a pre-proprietary golden age. That golden age never existed. I was there and I know" (Moody 2001). The commons was assembled contingently, by argument, accident, and specific decisions, and its present thinness is likewise the product of decisions, which is what makes it changeable; Ostrom spent her career insisting on exactly that pivot, away from treating participants as prisoners of "remorseless tragedies" and toward changing the conditions that produce them (Ostrom 1990, 7).

The second over-reading is that openness itself is the vulnerability, that closed and funded software would have been safe. Zalewski answered it in a reply beneath the same disclosure-weekend essay an earlier chapter drew on: "Supply-chain attacks on closed source happened before (e.g., Solarwinds), and persisted for longer, so this doesn't really portray OSS as worse" (Zalewski 2024, lcamtuf comment, 2024-03-30). The SolarWinds compromise ran inside commercial, well-resourced, digitally signed software from "as early as Spring 2020" to its December disclosure (FireEye 2020), roughly nine months that no bazaar can be blamed for. The argument is his, but the dates are the record's, and the comparison survives every discount applied to it.

The most careful calibration came from inside the commons, from the engineer who caught the thing. Freund, asked afterward about the exposure, bounded it rather than dramatizing it: "there aren't that many projects with that degree of exposure. . . . it was like I think 10 projects or something. And those are much more scrutinized and are going to be more scrutinized from now on than in the past," and prior attempts to introduce backdoors, he added, had been caught earlier in the process; "So I think it's not all everything is lost kind of situation either" (Freund and Roccia 2024). The estimate is his, rough and offered as such, and the register is the one the record supports. The commons is thin where it should be thick, misdescribed by its own mythology, governable but not self-correcting; and it is at the same time the system that produced the disclosure, the reverse-engineering, the forty-eight-hour response, and the

repair. Both facts are load-bearing. Neither cancels the other.

Zittrain compressed the whole construction into one sentence: "The generative Internet was crucially funded and cultivated by people and institutions acting outside traditional markets, and then carried to ubiquity by commercial forces" (Zittrain 2008, 174). Built outside the market; carried to ubiquity by it. Between those two clauses sits an account that has never been settled: who maintains what commerce carried off, and who pays for the maintenance. Firms build as well as take, as Zittrain also noted, since "classical, profit-maximizing firms like Red Hat and IBM can find it worthwhile to contribute to generative technologies like GNU/Linux" (Zittrain 2008, 64), and the kernel is the standing demonstration, deep benches, distributed review, institutional attention. One floor down, `liblzma` ran on the unpaid hours of one man in Finland. The same economy that staffs the top of the substrate leaves its lowest courses to volunteers. Why the money pools in one layer and never reaches the one below, who is actually subsidizing whom, and what the word "open" is now being made to mean in the fight over it: that is the next question.

Chapter 12

The Asymmetry

By 2023, open source had won so completely that even victory could feel like extraction. Asked by the Tidelift survey what sustaining open source feels like now, one maintainer replied: “If you asked me this 20 years ago I might have something inspiring to say, but now that open source has ‘won,’ it feels like we’re just giving away our work to companies who profit from it” (Tidelift 2023, 36). The complaint is not abstract. Log4j, the small Java logging library at the center of the 2021 emergency, was run by volunteers for nothing while, in the words of the technology journalist Patrick Howell O’Neill, quoted in a 2024 Sovereign Tech Fund report, “million- and billion-dollar companies rely on it and profit on it every single day” (Ellis and Bollampalli 2024, 8). The felt reality is a one-way flow: value pours out of the commons, and very little flows back.

Put in numbers, the imbalance is vast. A 2024 Harvard Business School working paper estimated the supply-side value of all the open-source software in use, the cost of writing it once, at $4.15 billion, against a demand-side value, the replacement value firms would face if they had to build that software internally, of $8.8 trillion (Hoffmann et al. 2024). Researchers measuring the public funding of open source read that gap as evidence of a “risk of underproduction” (Osborne et al. 2024, 3). Three orders of magnitude separate what the world takes from the commons and what it returns. Noah Kantrowitz, of the Python

Software Foundation, drew the conclusion a maintainer reaches: the imbalance between producers and consumers "has become so ingrained that for a company to re-pay (in either time or money) even a small fraction of the value they derive from the Commons is almost unthinkable" (Eghbal 2016, 75). And because software liability is weak, "the costs of insecurity are largely externalized and displaced onto the public" (Ellis and Bollampalli 2024, 29): the firms that profit from a fragile dependency are not the ones who pay when it breaks.

The story is true, and it is also, at the core of the structure, wrong. The simple version holds that the companies built on open source merely freeload. They do not. The Linux kernel, the most depended-upon piece of the substrate, is overwhelmingly paid work. The kernel's twenty-fifth-anniversary report, surveying development through 2016, found that "even if one assumes that all of the 'unknown' contributors are working on their own time, well over 80 percent of all kernel development is demonstrably done by developers who are being paid for their work" (Corbet and Kroah-Hartman 2016, 12). The paid share was not static but rising: unpaid contribution had fallen from 14.6% in 2012 to 11.8% in 2014 to 7.7% by the period the report covered (Corbet and Kroah-Hartman 2016, 12). And the money came from no single master. "Corporate participation in the process is crucial, but no single company dominates kernel development," the report observed, so that "no company can drive development in directions that hurt the others" (Corbet and Kroah-Hartman 2016, 12).

The breadth is the point. Across 2007 to 2019, the Linux Foundation counted 780,048 commits accepted into the kernel from 1,730 organizations, with the top twenty accounting for 68% and "a long tail of companies that only made a single commit" (Linux Foundation 2020, 13). The roster churns: firms that were strong contributors in 2007 have since been acquired and dropped away, others arrived late, and the foundation reads the turnover as a virtue, since "the diversity of contributors has been a strength and continues to provide resilience to the project" (Linux Foundation 2020, 14). Across open source as a whole, the researchers behind the Harvard estimate note that "the private sector has been the largest direct and indirect funder of OSS development to date, which

is unusual among public goods" (Osborne et al. 2024, 3). Corporate money is not absent from the commons. It is the dominant input.

The funding is not only historical; it is visible and current. In March 2026 the Linux Foundation announced "$12.5 million in total grants from Anthropic, AWS, GitHub, Google, Google DeepMind, Microsoft, and OpenAI to strengthen the security of the open source software ecosystem" (Linux Foundation 2026), building on the Alpha-Omega security fund, whose grants are "made possible by generous and significant donations from Amazon Web Services, Google, and Microsoft" (Alpha-Omega 2024, 3). The security researcher Michał Zalewski, writing during the disclosure weekend, rejected the freeload diagnosis outright. The pundits "pointing fingers at the supposedly exploitative relationship between Big Tech and the open source community," he wrote, "claim that the lack of adequate compensation is the source of all malaise. I don't buy this." Many major projects, he added, "are supported to a significant extent. Countless prominent OSS developers are on Big Tech payroll; quite a few projects receive hefty grants" (Zalewski 2024). Linus Torvalds had named the mechanism a quarter century earlier, and approved of it: "what better way of getting some of the less interesting work accomplished, boring stuff like maintenance and support, than doing it inside companies?" (Torvalds and Diamond 2001, 163).

What the simple story gets wrong is the core; what it sees correctly is the edge. The money is real, and it lands on the legible center while thinning across everything around it. Nadia Eghbal, the most precise chronicler of maintainer labor, mapped the shape years before XZ Utils. Projects "function well on a community basis if they are on the extremes of size," she wrote: small ones that need little upkeep, or "very large projects that have found significant corporate support (as in the example of Linux)" (Eghbal 2016, 63). The danger sits between the poles. "Many projects are trapped somewhere in the middle: large enough to require significant maintenance, but not quite so large that corporations are clamoring to offer support. These are the stories that go unnoticed and untold" (Eghbal 2016, 63). XZ Utils, the compression project whose `liblzma` library

every major Linux distribution leaned on, was a story in the middle. And the middle is mostly small: "47% of the packages have 0 or 1 functions," the 2020 Census II report observed of `npm`, the package registry for the JavaScript world, "and the average npm package has 112 physical lines of code" (Nagle, Wilkerson, et al. 2020, 22), the scale at which a library nobody notices can still sit beneath everything.

The mechanism that funds the center is a hiring pipeline, and it runs in only one direction. For the kernel, the anniversary report observed, demonstrated skill converts into salary: "Kernel developers are in short supply, so anybody who demonstrates an ability to get code into the mainline tends not to have trouble finding job offers. . . . As a result, volunteer developers tend not to stay that way for long" (Corbet and Kroah-Hartman 2016, 12). The economists Josh Lerner and Jean Tirole had named the underlying logic in 2005: contribution pays off later, in jobs and reputation, through what they called the "signaling incentive," and the signal is loudest where the audience is largest (Lerner and Tirole 2005, 58). Work "related to the operating systems and programming languages, whose natural audience is the community of programmers" carries "strong signaling incentives," they wrote, while the unglamorous tasks, the ones aimed at "the much less sophisticated end user," carry "lower signaling incentives" (Lerner and Tirole 2005, 60–61). `liblzma` was the second kind: critical, boring, invisible plumbing with no audience to perform for. Collin remained what he had long been, by his own public account: an unpaid hobby maintainer, carrying a dependency the rest of the system treated as settled.

Set the two measurements side by side and the asymmetry is the distance between them. The kernel runs above 80% paid; the broad population of open-source contributors does not. The 2020 FOSS Contributor Survey, which polled people rather than projects, found that "over half (51.65%)" of respondents were paid for their work, which means just under half were not (Nagle, Wheeler, et al. 2020, 14). The two numbers measure different worlds, and the survey says so itself: its data describe "people, not projects," and "[s]ome projects may not have anyone paid to contribute to them — even if they are critical and even if some of

the contributors are being paid to work on other projects" (Nagle, Wheeler, et al. 2020, 4). That caveat is the whole asymmetry in miniature. Money clusters where it can see a return, and the most critical maintainers are the hardest to see. The apparent counter-evidence runs the same way. An analysis of 2017 GitHub data, the 2020 Census II report noted, found that many of the most active contributors worked under their "Microsoft, Google, IBM, or Intel employee email addresses," contrary to the "popular image . . . of 'the overworked and underpaid programmer' " (Nagle, Wilkerson, et al. 2020, 25). The top committers of the most-used packages are disproportionately on a payroll, which looks like a refutation of the unpaid-maintainer thesis and is in fact the asymmetry seen from the other side: the subsidy concentrates on the heavily used, legible packages the report can rank, while the critical dependency below the ranking, `liblzma` among them, draws no such employment. The developer Feross Aboukhadijeh named the paradox: "[R]eliable, error-free transitive dependencies are invisible. Therefore, the maintainers are invisible, too. And, the better these maintainers do their job, the more invisible they are" (Eghbal 2020, 190).

None of this is new. The clearest precedent ran a decade before XZ Utils, in a library even more widely deployed. After the 2014 Heartbleed vulnerability exposed the private keys of much of the encrypted web, the researchers who measured its reach delivered a verdict that reads now like a forecast: "Despite the fact that OpenSSL is critical to the secure operation of the majority of websites, it receives negligible support," and "a code review would have uncovered the vulnerability" (Durumeric et al. 2014, 486). Their closing line named the open problem directly: "Our community needs to determine effective support models for these core open-source projects" (Durumeric et al. 2014, 486). The support, at the time, amounted to almost nothing. OpenSSL, the cryptographic library standing behind hospital chains, Amazon, the FBI, and the Pentagon, was maintained by a single engineer on an annual budget of $2,000, most of it small donations, the reporter Nicole Perlroth found (Perlroth 2021). Critical-everywhere code, one unpaid maintainer, no money: the support pattern visible before Heartbleed in 2014 and Log4j in 2021 is the same one that left XZ Utils

exposed in 2024, even though the failures themselves were different. The industrially subsidized kernel sits beside that pattern as the exception that proves the rule.

The asymmetry keeps producing the same failure because it is not negligence. It is rational. A company deciding how much care to spend on a dependency is running a calculation, and Russ Cox, who leads Google's Go programming language, wrote it down in 2019: the cost of a bad dependency is "the sum, over all possible bad outcomes, of the cost of each bad outcome multiplied by its probability of happening" (Cox 2019, 38). For a hobby project the expected cost is near zero; for production software, where "servers may go down, sensitive data may be divulged, customers may be harmed," it is high, and the rational response is to concentrate scrutiny on the dependencies that are both legible and high-stakes (Cox 2019, 38). Companies fund what they treat as an asset. They are "mostly interested in open source code itself as a 'product' or commodity," Eghbal notes, valuing "code quality, influencing the project's roadmap, and brand association" (Eghbal 2020, 187), none of which a small, invisible dependency offers. The trouble is that the diligence is unaffordable even to the person prescribing it. Cox admitted as much: the careful inspection he recommends for every new dependency is something "I have done only a subset of them for a subset of my own dependencies" (Cox 2019, 43). If the expert who designed the procedure cannot run it at scale, the gap is not a lapse. It is built into the economics.

The institutions created to close the gap demonstrate it in their own operating rules. Alpha-Omega, the security fund underwritten by the largest technology companies, explains its selection logic plainly: "we look for points of leverage and 'shovel readiness,' " which leads to "ecosystems like Rust and foundations like the Eclipse Foundation," because "these organizations already have relationships with security talent and the ability to hire and manage their work" (Alpha-Omega 2022, 9). A solo maintainer has no organization to receive a grant, no security team to staff, nothing to make shovel-ready. The fund concedes a floor to its own sight, the level of granularity below which, as an earlier

chapter quoted it admitting, it cannot reliably measure criticality at all. That level is precisely where `liblzma` lived, a small library reachable beneath `sshd`. And when the money flows, it flows to the visible: of the roughly $2.84 million Alpha-Omega granted across eight organizations in 2023, the single largest grant, $620,000, went to the Linux kernel (Alpha-Omega 2024, 14), one of the most industrially subsidized projects in open source. The logic is sound, the recipients are real, and not one dollar reached a one-maintainer dependency like `liblzma`.

The neglect is not even unique to open source. Zalewski, who rejected the freeload story, pointed to its mirror image inside well-funded companies: "even with Big Tech staffing and money, if you have an internal library that almost never needs any attention, the 'ownership' of that code becomes pretty theoretical too. It's hard to build a rewarding career on being very familiar with some boring, old dependency that's just taken for granted by everyone else" (Zalewski 2024). The problem is attention and career value, not only money, which is why a 2025 interview study of closed-source firms found the opposite pattern from open source: where major supply-chain failures like SolarWinds "had no significant impact" on open-source signing adoption, they did move industry, because of "the substantial incentives, such as commercial and reputation losses, that drive greater adoption in the non-open-source industry" (Kalu et al. 2025, 94). The legible, commercially exposed core gets defended; the diffuse, unowned periphery does not. None of this makes the asymmetry acceptable. Rational is not the same as adequate, or fair, or safe.

The asymmetry is not only a matter of money. It is written into the instruments built to secure the supply chain. The flagship U.S. measure was the secure-software attestation form federal agencies required vendors to sign until the Office of Management and Budget rescinded the mandate on January 23, 2026 (Office of Management and Budget 2026, 1). Even at full force, the form carved open source out of its own scope. The form, it states, "does not cover . . . Open source software that is freely and directly obtained directly by a Federal agency," nor "Third-party open source and proprietary components that are

incorporated into the software end product used by the agency," nor "Software that is freely obtained and publicly available" (Cybersecurity and Infrastructure Security Agency 2024b, sec. I, p. 5). A freely obtained, publicly available, third-party open-source component incorporated into countless products is exactly what XZ Utils' `liblzma` library is: it falls into three of the carved-out categories at once. And the form had to be signed by "the Chief Executive Officer . . . or their designee, who must be an employee of the software producer and have the authority to bind the corporation" (Cybersecurity and Infrastructure Security Agency 2024b, instructions p. 4). The unit of accountability was the firm. A volunteer maintaining a critical dependency had no CEO, no corporation, and nothing to bind. The carve-out was not an oversight: the original 2022 mandate had already conceded there was no "software producer" to self-attest for open source, and routed it instead to a paid third-party assessor (Office of Management and Budget 2022, sec. II(1)(d)). The door narrowed from there, and then the mandate was withdrawn.

The same grammar runs through the rest of the apparatus. The NIST Secure Software Development Framework, the federal government's reference standard, addresses itself to two audiences, "software producers" and "software acquirers," both of them organizations bound by procurement (National Institute of Standards and Technology 2022, iii). The unpaid maintainer of a transitive dependency is neither. The framework knew the gap: in February 2022, as the operation against `xz` was already months underway, it listed applying itself "in the context of open-source software" among the topics that "future work may expand on" (National Institute of Standards and Technology 2022, 1). The instrument built to secure the software supply chain had not yet reached the part of the supply chain then being compromised. Europe's regulators drew the same line. In the EU's recommendations on supply-chain threats, open source appears only as something a commercial supplier must attest to, "ensuring and attesting to . . . the integrity and origin of open source software used within any portion of a product" (European Union Agency for Cybersecurity (ENISA) 2021, 28), never as labor to be sustained. The one real attempt to push the cost back

onto the beneficiary is in the EU Cyber Resilience Act (CRA), which requires that a manufacturer who finds and fixes a flaw in an open-source component "report the vulnerability to the person or entity . . . maintaining the component" and "share the relevant code" upstream (European Parliament and Council of the European Union 2024, art. 13(6)). It is a genuine redistribution of effort, and it is also more work arriving in the maintainer's inbox, triggered only when a firm happens to look.

Underneath the money and the policy sits a question about a word. Who gets to call a thing "open," and what does the claim oblige? The dispute is not new, and it is not vague, because "open source" has a published, checkable definition. "Open source doesn't just mean access to the source code," the Open Source Definition begins, before listing the ten criteria a license must meet (Open Source Initiative 2007, Introduction). The standard makes corporate claims on the word falsifiable rather than rhetorical. Torvalds had used it that way in 2001, dismantling Sun Microsystems' claim that its Jini system was open: "It was open source in the sense that you could read the source, but if you wanted to make any modifications or make it part of your infrastructure, you had to license it from Sun" (Torvalds and Diamond 2001, 151). Readable source is not open source if use and modification are gated. The move was old. The AI boom merely raised the stakes.

The current test case is Llama, the family of large language models Meta began releasing in 2023 under a license it calls open. At the center is a real grant: users get a "non-exclusive, worldwide, non-transferable and royalty-free limited license" to "use, reproduce, distribute, copy, create derivative works of, and make modifications to" the model (Meta 2024b, sec. 1(a)). The word that does the work is "limited." Any company whose products exceeded "700 million monthly active users" on the release date "must request a license from Meta, which Meta may grant . . . in its sole discretion" (Meta 2024b, sec. 2), which is openness offered to everyone except the handful of competitors large enough to matter. A separate, incorporated acceptable-use policy bars whole fields of endeavor: "Military, warfare, nuclear industries or applications, espionage" (Meta

2024a, sec. 2(a)). And every product built on the model must display a "Built with Llama" notice (Meta 2024b, sec. 1(b)(i)), the one trademark obligation the license imposes, since "[a]ll goodwill arising out of your use of the Mark will inure to the benefit of Meta" (Meta 2024b, sec. 5(a)).

Measured against the published standard, the claim fails where the standard is most explicit. A license "must not discriminate against any person or group of persons," which the user gate does (Open Source Initiative 2007, point 5), and "must not restrict anyone from making use of the program in a specific field of endeavor," which the use policy does (Open Source Initiative 2007, point 6). The Open Source Initiative, the nonprofit that maintains the definition, had reached exactly that verdict about the model's earlier version and stated it without hedging: Meta "has created the misunderstanding that LLaMa 2 is 'open source' – it is not," confusing open source with "resources available to some users under some conditions" (Maffulli 2023). The grounds it named, the user gate and the field bar, carry into Llama 3.1 unchanged. None of this makes the license worthless. It is not simply closed: Meta grants users ownership of the "derivative works and modifications" they make (Meta 2024b, sec. 5(b)). The shape is the point, permissive at the center and controlled at the strategic edges of scale, brand, and field of use. That is a precise description of strategic openness. It is not a description of open source.

That openness might be a posture rather than a principle is the oldest theme in the story. In 1976 a young Bill Gates, furious that hobbyists were copying his software, wrote an open letter that framed the whole future argument in two sentences: "Hardware must be paid for, but software is something to share. Who cares if the people who worked on it get paid?" (Gates 1976). He meant it as a reductio; the commons spent fifty years answering it in earnest. And when the movement needed a name that business could accept, it chose one for its marketing value. "Open source" was coined at a 1998 strategy meeting as a deliberate rebrand of "free software," in Eric Raymond's candid account "a marketing campaign" requiring "spin, image-building, and rebranding" to make it work (Raymond 2001, 175). The biographer Sam Williams recorded the design

intent: the new label let companies "examine the free software phenomenon on a technological, rather than ethical, basis" (Williams 2010, 114). Michael Tiemann, the founder of the first free-software business, later put the lesson plainly: "it turns out that brand trumps almost everything" (Moody 2001). Eghbal saw where the logic led. Large companies embrace open source "to build trust and influence" (Eghbal 2016, 114), she wrote in 2016, and warned that digital infrastructure might come to exist in "a series of 'walled gardens,' each well-supported and technically open, but effectively championed by one company" with the resources to control it (Eghbal 2016, 117). The fight over Llama is that prophecy arriving on schedule.

The asymmetry is real and harmful, and the funding logic that produces it is rational; corporate openness is strategic, not charitable, and the word "open" was a brand before it was a principle. All of that holds at once. The man at the top of the structure conceded the core of it. Asked about the kernel's elaborate, well-funded development process, Torvalds allowed that "what we do is not necessarily something that can translate to 99% of all the open source projects" (Mastery Learning 2024, 7:20). The kernel is the exception, not the model, and the distance between the exception and everything else is the asymmetry itself. The open question is the one the 2020 Census II report asked in its flat phrasing: "Are those who benefit most from FOSS projects doing their 'fair share' to support the communities behind them?" (Nagle, Wilkerson, et al. 2020, 14). The money reaches the legible core and stops. Who actually does the work in the long tail, who pays for it, and what it costs the people who carry it without pay are the questions the asymmetry leaves open.

Chapter 13

The Maintainer Economy

In 2020 the webcomic xkcd published a drawing that the people who run the internet recognized at once as a portrait of themselves. Numbered 2347 and titled "Dependency," it shows a tower of neatly stacked blocks labeled "all modern digital infrastructure," the whole structure balanced on a single load-bearing piece near the bottom, drawn so thin it looks like an accident waiting to happen. An arrow identifies that piece as "a project some random person in Nebraska has been thanklessly maintaining since 2003" (Munroe 2020). It circulated as a joke. It is more accurate as documentary. The thin block is not a metaphor for fragility in general. It is a specific, common, and largely unpaid arrangement: one person, working alone and unrecognized, holding up infrastructure the rest of the economy treats as a settled fact.

That arrangement has a name once you decide to look at it, and the name is labor. The infrastructure scholars Geoffrey Bowker and Susan Leigh Star noted that few people study the maintenance of complex systems "as a kind of work practice, with its attendant financial, skill, and moral dimensions" (Bowker and Star 1999, 5), and Star, writing on the ethnography of infrastructure, gave the people who perform that work their precise description. In any system there are workers "whose work goes unnoticed or is not formally recognized," she observed, the cleaners and janitors and parents. And she warned that "leaving

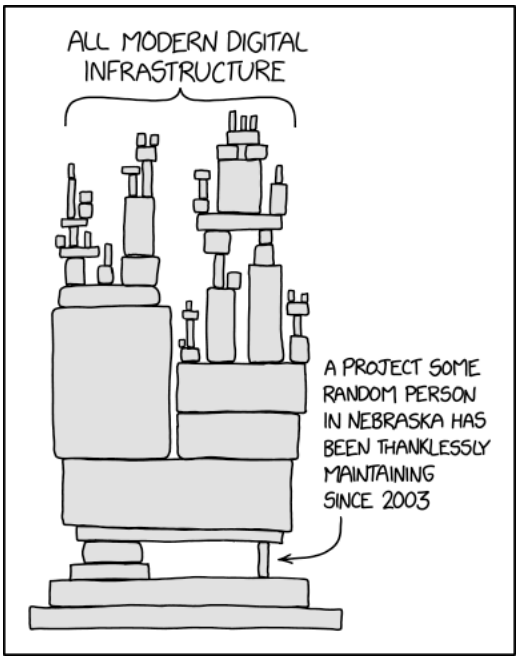

Figure 13.1: "Dependency" (www.xkcd.com/2347/). Source: (Munroe 2020). Used under CC BY-NC 2.5 (www.xkcd.com/license.html).

out what are locally perceived as 'nonpeople' can mean a nonworking system" (Star 1999, 386). The maintainer of a critical library is one of those nonpeople, invisible precisely until the system stops working without him. Nadia Eghbal, who has written the most exact account of this economy, put the same fact in the language of the digital commons. Open-source code generates "trillions of dollars in economic value," she wrote, while many of the people who produce it "are not directly paid for their work," so that, absent other rewards, "maintaining code for general public use quickly becomes an unpaid job you can't quit" (Eghbal 2020, 8).

The condition is visible even at the funded center of the commons, where it has been turned into a running joke. The Linux kernel records who is responsible for each part of itself in a plain-text file called MAINTAINERS; in the catch-all entry covering everything no one else has claimed, the status line read "Buried

alive in email" in the 1996 file and, by 2020, "Buried alive in reporters" (Linux Foundation 2020, 8). Even there, in the best-resourced open-source project on Earth, the work concentrates in a thin layer: roughly one-third of the contributors in a given development cycle submit exactly one patch and are not heard from again (Corbet and Kroah-Hartman 2016, 10). The kernel can absorb that because it is paid for. Much of the long tail cannot. What the discovery of the backdoor forced into view, and what the researcher Evan Boehs named directly in its aftermath, was that the field had been spending its attention in the wrong place. "We must stop focusing on fuzzing and static analysis," Boehs wrote, "and instead turn our attention to the human costs of open source" (Boehs 2024). The point overstates the case, since the automated tools mattered to the catch, but it locates the weak layer correctly. It was never only the code.

The intuitive remedy is money, and the evidence that money alone is not the lever is unusually consistent. The 2020 FOSS Contributor Survey, a study of free and open-source software developers run jointly by the Linux Foundation and Harvard, asked thousands of contributors what motivated them and found that nonmonetary reasons dominated: adding a needed feature, the enjoyment of learning, the satisfaction of creative work. "Being paid to develop FOSS was the most likely motivation to rank in an individual's bottom three motivations," the survey reported, "even for those who reported receiving payment for their contributions" (Nagle, Wheeler, et al. 2020, 4). This was neither new nor fragile. A 2005 study of free and open-source developers had already concluded that "enjoyment-based intrinsic motivation . . . is the strongest and most pervasive driver" (Lakhani and Wolf 2005, 3). The later survey was not a discovery so much as a confirmation of a two-decade pattern. The mechanism behind the numbers is older still. Eric Raymond described open source as a gift culture in which "social status is determined not by what you control but by what you give away" (Raymond 2001, 81), a reputation economy whose currency is the esteem of peers, not cash. Linus Torvalds, who built the largest project in that economy, stated the principle plainly: "In a society where survival is more or less assured, money is not the greatest of motivators" (Torvalds and Diamond

2001, 227).

None of which means maintainers do not want to be paid, and the argument collapses into sentimentality the moment it forgets the difference. Torvalds, who grew wealthy when his collaborators' companies went public, was scornful of the suggestion that he had renounced money: "I've always hated that self-effacing monk image because it's so uncool. It's a boring image. And it's untrue" (Torvalds and Diamond 2001, 190). The maintainers themselves are blunt about it. When the Tidelift survey tested "the regularly cited theory that most maintainers prefer to work on open source as an unpaid hobby," it found the opposite: "77% of the maintainers who are not paid would prefer to get paid" (Tidelift 2023, 4). Wanting to be paid and being motivated by pay are different things, and the maintainer economy has to hold both at once. The work is not done for money, and the people doing it would still rather not do it for free.

If money is not the lever and money is still wanted, the failure lies in the channels that carry it. They reward the wrong thing. The dominant models all flow toward what can be seen: recurring donations through GitHub Sponsors, the platform where users send maintainers small monthly sums; subscription income split among maintainers by the company Tidelift; grants from foundations; and one-time cash bounties for reported bugs. The maintainers see the bias clearly. "I feel the visibility of big projects are amplified," one told the Tidelift survey, "but the work of small but often essential maintainers is not recognized. #frustrated" (Tidelift 2023, 38). Each model reproduces the bias in its own way. Donations raise what Mike McQuaid, lead developer of the Homebrew project, called "sticker money," enough to print swag but "not enough money to quit one's job and work on open source full-time" (Eghbal 2020, 193). Bounties pay for a discrete, visible artifact rather than ongoing care: in the Internet Bug Bounty program, which pays cash for flaws found in widely used code, 61% of those who earned a bounty had exactly one successful submission. "These are not hackers that are necessarily going to have a long-term investment in the ongoing health and maintenance of the project," the analysis noted (Ellis and Bollampalli 2024, 35). The optimistic model of the era, Kevin Kelly's

"1,000 True Fans," in which a creator needs only a few thousand devoted supporters to make a living (Eghbal 2020, 201), works for a visible creator and not for an invisible dependency. Even micro-donations, researchers found, have a "limited" effect on the projects that receive them (Osborne et al. 2024, 3).

The one model that does deliver real money tends to deliver it invisibly. A maintainer in the Tidelift survey named the arrangement "benevolent employment," the practice of companies that "subside OSS by employing maintainers for non-related work" and let the open-source maintenance happen on the side (Tidelift 2023, 38). The open-source developer Karl Fogel, describing how free software actually gets funded, observed that much of it is underwritten by employers who donate their staff's time and bandwidth "albeit unknowingly," organizations that "may or may not be institutionally aware of" what they are paying for (Fogel 2020, 67). The labor is real, and the subsidy is real, but it is illegible on every balance sheet, contingent on an employer's unrelated priorities, and impossible to count on. It is the visible model's mirror image: support that arrives by accident rather than by design.

You can watch the bias operate in real time. When Andres Freund's disclosure made the backdoor public, the spontaneous gesture of gratitude in the comment threads was a reader announcing, "I donated $1000 to Debian for your work. Let's all do something nice for Debian, please?" (Corbet 2024, Cyberax comment, 2024-03-29). The money moved toward the visible hero and the visible institution, ad hoc and gratitude-driven, in precisely the pattern the funding models systematize. It did not move toward the unsupported maintainer of the library that had actually been attacked.

The question of who pays for software that the world uses for free is older than open source, and it was first posed by the person least sympathetic to the answer. In 1976 a young Bill Gates, angry that hobbyists were copying his company's software, published an open letter that framed the whole future argument in one line: "Who can afford to do professional work for nothing?" (Gates 1976). He even priced his own underpayment, complaining that royalties had made "the time spent on Altair BASIC worth less than $2 an hour"

(Gates 1976). Gates meant the question as a reductio, an impossibility no rational person would accept. The commons spent the next fifty years answering it in earnest, and the answer was Lasse Collin and the thousands of maintainers beneath most of the digital economy, who did exactly the professional work for nothing that Gates said no one could afford. The instructive difference is the exit. Gates could price his underpayment and walk away from it to build a company. The maintainer of a critical dependency cannot.

Collin's situation was not an exception to the maintainer economy. It was its clearest expression. He had carried `xz` and its `liblzma` compression library, a component used across Linux systems to unpack data, for years and without pay. Even receiving support came with friction. "Due to the Finnish money collection act, I do not accept donations from Finns or people living in Finland," his project page read (Collin, n.d.); he set up an account to receive donations only after the disclosure brought new attention to the project. His position was not idiosyncratic, either. A 2025 study of critical open-source projects described the prevailing condition as one of maintainers "who often experience burnout and lack motivation but continue out of a sense of duty" (Przymus and Durieux 2025, 91), and the census of the most-used packages keeps surfacing the same shape. Its authors built a list of the "keystones of the FOSS ecosystem" (Nagle, Wilkerson, et al. 2020, 24) and found them carried by one or two people each, like the few thousand lines of the JavaScript utility `inherits`, a single committer beneath an enormous share of the software world.

The pattern has a documented prehistory. A decade before XZ Utils, the encryption library OpenSSL stood behind most of the secure web while running, Eghbal recounts, on barely enough money "to pay the salary of one developer," so that "two-thirds of the Web relied on encryption software maintained by just one full-time employee" (Eghbal 2016, 12). After the Heartbleed flaw exposed that arrangement in 2014, Steve Marquess, who had tried to fund the project, put the condition in a single line: "The mystery is not that a few overworked volunteers missed this bug; the mystery is why it hasn't happened more often" (Eghbal 2016, 13). Marquess also named the felt texture of the work, the strain

of carrying code that is invisible until it fails: "It takes nerves of steel to work for many years on hundreds of thousands of lines of very complex code . . . knowing that you'll be ignored and unappreciated until something goes wrong" (Eghbal 2016, 14). The condition reaches back further than that. In 1996 Eric Allman went looking for companies to fund Sendmail, the program then carrying most of the world's email, and found that his modest proposal "couldn't float" (Moody 2001). Every arrangement that does hold together, Eghbal concluded after talking to maintainers who had found a way to pay themselves, turns out to rest on luck: "Everybody is getting lucky. Luck lasts for a couple of months, maybe a year or two, and then it runs out" (Eghbal 2016, 106).

The maintainer economy has an optimistic answer to all of this, and the operation against XZ Utils was built to exploit it. In a widely read 2019 essay on managing software dependencies, Russ Cox offered a reassurance the ecosystem broadly shared: "if a widely used package loses its maintainer, an interested user is likely to step forward" (Cox 2019, 39). Something close to that expectation played out around XZ Utils, except that the interested user who stepped forward was the operator who used the name "Jia Tan." The assumption that popularity guarantees a successor holds only if the volunteer who appears can be trusted, and nothing in the model checks. The pool such volunteers are drawn from is, in any case, thin and short-lived. "A generation for free-software programming is very short," Matthias Ettrich, who founded the KDE desktop environment, told the writer Glyn Moody; "it's between half a year and maybe two years for those more dedicated" (Moody 2001). And the work a project like XZ Utils actually needs is the opposite of what a large, churning crowd supplies: not a stream of small drive-by contributions but the sustained, high-skill stewardship of a single coarse-grained component, the kind of investment that, by its nature, only a few people can ever make (Benkler 2006, 101). A thin pool of trusted hands, no succession plan, and a published expectation that a stranger would arrive to help. The security researcher Kevin Beaumont, surveying the incident, observed that this kind of maintainer-account takeover "actually happens all the time" and that open-source developers are "largely unpaid, and face significant

amounts of online abuse from users of the software" (Beaumont 2024). The handover was not a freak event. It followed the system's ordinary assumptions.

What maintainers say they need, when asked, is not first of all money. Among those who had quit or considered quitting, the Tidelift survey found that requests for experienced help outranked requests for pay, with 64% choosing one or both of the help options, which "makes getting experienced help even more important to maintainers than getting paid" (Tidelift 2023, 32). The scarcest resource is a second trusted person, someone to share the load, review the patches, and carry the project when the first maintainer flags. That is the gap the operation found, and it is precisely the gap that funding aimed at the project, rather than at succession, does not fill.

Some models do work, and honesty requires saying so. The clearest successes share one structural feature: they pay a specific person to do specific unglamorous work, and they can do so because an organization exists to receive the money and manage the role. "One of the most impactful ways to change the security culture of an entire community is to make it someone's job," the Alpha-Omega security fund wrote of its strategy (Alpha-Omega 2024, 8), and its showcase case is real and named: Alpha-Omega money let the Python Software Foundation hire Seth Larson to drive its security work (Alpha-Omega 2024, 6). Django runs a comparable arrangement, a paid fellowship that does the patch review and release management no one does for free, described by one of its developers as "a direct answer" to chronic contributor churn (Eghbal 2020, 204). A Django security volunteer described what the paid help bought: "when I know that there are some people paid to do it, I trust them to be more patient than I am" (Ellis and Bollampalli 2024, 26). But the same report names the catch in the next breath: "for thinly resourced projects, developing a security team with paid support may not be possible" (Ellis and Bollampalli 2024, 26). The model that rescues Django rescues it because Django has a foundation able to receive the grant, recruit for the role, and manage it. The solo maintainer who is the entire project has no such organization, which is exactly why the model that works cannot reach him. There is, as Eghbal puts it, "no one-size-fits-all

solution," because open source no longer has one shape (Eghbal 2020, 185).

There is also a trap inside the remedy. Making maintenance legible enough to fund makes it legible enough to cut. Star saw the dilemma in the hospitals she studied: leave the work tacit and it vanishes into the wallpaper, but "make it explicit, and it will become a target for . . . cost accounting" (Star 1999, 386). A fix that turns the maintainer into a budget line solves the funding problem by manufacturing the accounting problem, and the freedom from that ledger was part of what made the commons productive in the first place.

The strongest evidence for the argument comes from the institution best placed to refute it. Alpha-Omega is one of the best-funded open-source security initiatives in the world, underwritten by the largest technology companies. In December 2022, fifteen months before the backdoor surfaced in exactly such a project, it conceded the precise gap: "we haven't proven out a model for funding security work on single-maintainer projects" (Alpha-Omega 2022, 17). The admission only sharpened with time. The fund later wrote that "this remains a human-scale problem and many maintainers of the long-tail projects are already working nights and weekends," and that "adding money isn't a quick fix" (Alpha-Omega 2025a, 18); that the binding constraint is "the scarcity of dedicated time" rather than funding (Alpha-Omega 2025a, 13); and that the danger in any remedy is "merely shifting more unpaid work onto their shoulders" (Alpha-Omega 2025a, 3). The same caution comes from the center of the subsidized world. Greg Kroah-Hartman, one of the Linux kernel's lead maintainers, said of a 2026 round of security grants that "grant funding alone is not going to help" with the overload then arriving (Linux Foundation 2026). Even the skeptics who reject the labor framing land in the same place from the other side. The researcher Michał Zalewski argued that small foundational libraries fail not for want of cash but for want of anything engaging left to do on them: "even if you wave some cash around, it's hard to build a sustainable community around watching paint dry" (Zalewski 2024). The money exists. The will exists. What does not exist is a fit between the shape of the funding and the shape of the work.

The law has now caught up to part of the diagnosis. A 2021 federal report

stated the premise from the security side in two flat sentences: "Software that is not maintained is at greatest risk of being exploitable. Older software is at a greater risk of not being maintained" (National Telecommunications and Information Administration 2021, 18). The risk concentrates exactly where maintenance is thinnest, in the long tail the funding routes past. The European Union's Cyber Resilience Act went further and invented a legal category for the bodies that fund the commons, the "open-source software steward," defined as "a legal person, other than a manufacturer," that sustains open-source projects and "ensures the viability of those products" (European Parliament and Council of the European Union 2024, art. 3(14)). But the same regulation leaves the individual contributor outside that category: it "does not apply to natural or legal persons who contribute with source code to products with digital elements qualifying as free and open-source software that are not under their responsibility" (European Parliament and Council of the European Union 2024, rec. 18). The maintainer who is the whole project remains hard for the law to see. The instinct to fund maintenance is in the record, too. The U.S. Cyber Safety Review Board, reviewing the Log4j emergency of 2021, proposed that "funding the maintenance of critical open source software could drive a more sustainable model for security at scale" (Cyber Safety Review Board 2022, 26). Whether any such fund can reach criticality rather than visibility is the question the policy chapter takes up.

The deepest difficulty is that the same condition that left the commons unfunded is the one that made it work. David Heinemeier Hansson, who created the Ruby on Rails framework, put it sharply: open source has been "such an incredible force for quality and community exactly because it's not been defined in market terms. In market terms, most open source projects should never have had a chance" (Eghbal 2016, 60). A repair that simply imports the market, that prices the maintainer's labor and books it against a return, risks dissolving the nonmarket culture that produced the work in the first place. Freund, who caught the backdoor and spent the following months turning over the conditions that produced it, reached the modest version of the conclusion in a later

interview: "we need to do something more about supporting some projects that are very crucial" (Freund 2024c, 01:21:31). Something more, and something that fits. The maintainer economy did not fail for want of money. It failed because the money, when it came, could not find the person.

Part IV

The Aftermath

How many other Jia Tans do you think are paid to insert backdoors in open-source projects?

— Thomas Roccia, *The XZ Backdoor Story* (DEF CON 32)

Chapter 14

The Hunt

In April 2024 the threat-intelligence firm Mandiant published a report that read, in places, like a dossier on a person. Its subject was a Russian military hacking group the company had tracked for more than a decade, and the document announced a kind of promotion. "Given the active and persistent threat to governments and critical infrastructure operators globally," it stated, "Mandiant has decided to graduate the group into APT44" (Roncone et al. 2024, 1). The new label resolved to an institution with an address. APT44, "commonly known as Sandworm," was "a Russian Federation backed threat group attributed by multiple governments to Unit 74455, the Main Centre for Special Technologies (GTsST) within the Main Directorate of the General Staff of the Armed Forces of the Russian Federation (GU), commonly known as the Main Intelligence Directorate (GRU)," and Mandiant fixed the unit's founding in 2009 from "publicly available images of the unit's anniversary insignia" (Roncone et al. 2024, 2). APT is the security industry's shorthand for an advanced persistent threat: a well-resourced intruder that picks a target, gets inside, and stays hidden for years. The GRU is Russia's military-intelligence service. The report is what attribution looks like when it works, a sprawling cluster of activity converted, by a deliberate and gated institutional act, into a numbered unit of a named state, down to the insignia on the wall.

Attribution does not always arrive at the moment of disclosure; more often it accretes. When the SolarWinds compromise became public in December 2020, the firm then called FireEye could say only that "we are tracking the actors behind this campaign as UNC2452" (FireEye 2020), a placeholder for an actor it could not yet name. Sixteen months later the placeholder resolved. Mandiant had "gathered sufficient evidence to assess that the activity tracked as UNC2452 . . . is attributable to APT29" (Mandiant 2022), and its finding "matches attribution statements previously made by the U.S. Government that the SolarWinds supply chain compromise was conducted by APT29, a Russia-based espionage group assessed to be sponsored by the Russian Foreign Intelligence Service (SVR)" (Mandiant 2022), Russia's civilian foreign-intelligence service. The machinery that turns an artifact into a name demonstrably exists, and it works on exactly the kind of patient, supply-chain-minded adversary the xz operation resembled. It produced no name for xz. No public attribution links APT44, Sandworm, APT29, the SVR, or any other named actor to the backdoor in xz: the groups above are a measure of what successful attribution can do, not a claim about who did this. Two years on, the operator is known only by the name on the commits, "Jia Tan," which is not a name so much as the absence of one.

What makes the blank strange is that the record is not thin. The XZ Utils incident is among the most exhaustively documented intrusions in the history of open-source software. Every change the operator made sits in a public version history; every message survives in mailing-list archives; the late moves are timestamped to the minute. A real-time chat channel logged the operator's connection, a network lookup captured a connecting address, the release archives carry cryptographic signatures, and the backdoor's trigger mechanism has since been published and dissected line by line. By any ordinary intuition, that much evidence ought to converge on a person. It does not. The sharpest way to feel why is to set it beside a case that ran the other way.

For most of Bitcoin's first decade its users believed the currency was anonymous, and they were wrong, and the demonstration of how wrong became its

own genre of investigation. Andy Greenberg's account of that work, *Tracers in the Dark*, describes investigators "tracing a cryptocurrency that had once seemed untraceable" (Greenberg 2022) to crack one criminal case after another. The property they exploited had been hiding in plain sight. As the computer scientist Sarah Meiklejohn put it in the 2013 paper that opened the field, "Bitcoin has the unintuitive property that while the ownership of money is implicitly anonymous, its flow is globally visible" (Meiklejohn et al. 2013, 127). The public ledger recorded every transaction permanently; in Greenberg's phrase, "everyone was a witness to every payment" (Greenberg 2022). Substitute "code" for "money" and the sentence describes the operator with unsettling precision. The flow of the work, every commit, every email, every signing event, was globally visible. The ownership behind it was not. So why does one fully public record name people and the other does not?

The answer is not in the records but at their ends. Crypto investigators could trace value through the ledger, but the ledger alone never produced a person. Greenberg describes the realization that broke the cases open: the chains of transactions "eventually" led "to exchanges like Mt. Gox and Bitstamp, where they seemed to be cashed out for traditional currency. For an academic researcher, this was a dead end. But anyone with the subpoena power of law enforcement, Meiklejohn realized, could very likely force those exchanges to hand over information about the accounts behind those transactions" (Greenberg 2022). Meiklejohn's own paper had stated the point dryly: "an agency with subpoena power would be well placed to identify who is paying money to whom" (Meiklejohn et al. 2013, 128). De-anonymization rode on a chokepoint in the real world. Sooner or later, value had to convert into ordinary money, and that conversion happened at a regulated exchange that collected identities, where a subpoena could force a name. The operator never had to cash out. The currency of the operation was trust, and it was spent in place: there was no exchange at the end of the trail, no account to subpoena, no real-world gate where the pseudonym had to anchor itself to a person.

Even fully exploited, the public record yields an actor, not an identity. Meik-

lejohn's method could collapse a scatter of separate keys into a single node by "transitively" tainting an entire cluster once one member was labeled, producing "a condensed graph, in which nodes represent entire users and services rather than individual public keys" (Meiklejohn et al. 2013, 128). The same logic lets the XZ Utils record be read as coordinated activity: the helpful contributor "Jia Tan," the impatient strangers who pressed for a faster handover, the account that later pushed the poisoned release downstream, all clustering into a single operational node. But a node is not a name. Clustering can show coordinated personas; it cannot tell you whether they were one person, several people, or a contributor later co-opted. That last step needs a label from outside the record, and against this operator no outside label ever came. Tigran Gambaryan, the federal investigator at the center of Greenberg's biggest cases, drew the line as a working rule: against an adversary a state protects, "you might get a name-and-shame out of it," but "you're not going to get a body" (Greenberg 2022). The crypto criminals were reachable because the money chain ended at an identity. State-shielded actors are not: "the real problem remains rogue countries like Russia and North Korea," Greenberg writes, "countries whose governments allow their citizens to defy global law enforcement action even when their activities are fully visible on the blockchain" (Greenberg 2022). Brian Krebs, the security journalist, made the same point about XZ Utils from the other end. His own investigative tools, he wrote, "can't hold a candle to the tools available to our government and three-letter agencies. And when even they can't see much of a trail with all their spy tools and off-books access, that's telling. Thing is, those 3-letters probably are not going to tell us what they find (or don't find)" (Krebs 2024, post 112198143331078347). Three cases sit on one spectrum: in crypto, a name and a body; with APT44, a name and no body; with xz, neither.

What the open record offered, then, was not identity but behavior, what Meiklejohn called "idioms of use," the habitual tells that gradually erode a pseudonym (Meiklejohn et al. 2013, 128). The discipline of the hunt is to read those tells for exactly what they establish and no more. The most striking of them is an absence. Krebs, who spent hours on the question, found that the

email addresses "used for a couple of years at least by the parties involved have absolutely *zero* trace in any kind data breach or database beyond Github/Gitlab, and maybe Tukaani and Debian and a few mailing lists" (Krebs 2024, post 112197305365490518). Ordinary identities leak: over years of use, an address turns up in some breach corpus, the haul of stolen data that circulates after a hack. These did not. The explanation Krebs reached for was the discipline of a professional. "Very few people do opsec well," he wrote, using the security world's shorthand for operational security, the practice of leaving no linkable trace, "and for every year you're operating under the same name, nick, number, email, etc you dramatically increase the risk of screwing up that opsec. And almost everyone does, eventually" (Krebs 2024, post 112197305365490518). The operator did not. From the clean absence Krebs drew an inference, and he marked it as one: "to find it multiple times suggests we're dealing with an operation that was set up carefully from the beginning. And that almost certainly means a group project (state-sponsored)" (Krebs 2024, post 112197305365490518). The reasoning is sound and the conclusion is hedged, and the hedge is the point. "Almost certainly" is a probability, not a verdict, and it names no one.

The other artifacts behave the same way: each is a real lead, and each stops short of a person. An independent study of the commit timestamps concluded, as Evan Boehs summarized it, that "the perpetrator worked 'Office Hours' in a UTC+02/03 timezone," and noted that "they worked through the Lunar New Year, and did not work on some notable Eastern European holidays, including Christmas and New Year" (Boehs 2024). That pattern points toward a place and away from another, but it is a pattern in metadata, not a location. The single piece of geographic data was thinner still. The only public geolocation for "Jia Tan," Kaspersky found, was "a Singaporean VPN exit node," the server through which the operator's traffic emerged, masking wherever it actually began, and the vendor immediately cautioned that the choice of exit could itself be staging: "if constructing a fictitious identity, using that particular exit node would definitely be a selected resource," and "our pDNS confirms this IP as a Witopia VPN

exit" (Kaspersky GReAT 2024a), a commercial service identified through passive DNS, the historical record of which addresses have served which domains. A lead that confirms the operator used a VPN is a lead that confirms there is no location to read. Even the backdoor's cryptography, the most legible artifact of all, names no author. Reverse-engineering the payload, the security researcher Anthony Weems found that "the backdoor uses a hardcoded ED448 public key for signature validation and decrypting the payload," and that to make it run at all "we can replace this key with our own" (Weems 2024). Ed448 is a public-key signature scheme: a secret key signs, and a matching public key verifies. The backdoor carried only the public half, which meant it would accept commands from one party alone, the holder of the matching private key; that proved control of the backdoor, not a name. The encryption around the payload reused the public key's bytes, so any onlooker could "decrypt any exploit attempt" (Weems 2024) and watch the door from outside, but watching is not opening. The operation built itself a private door with exactly one keyholder, left the door in plain view, and the key names no one.

The permanence that undid the crypto criminals was present here too, and produced nothing, because what the permanent record preserves are constructed artifacts that anchor to nothing real. It also turns the attribution problem inward, onto the code itself. The release the world initially fell back to, version `5.4.5`, "seems to be still signed by the adversary," one responder noted on disclosure day (Hess 2024, msg #72): the operator's signature, the very mechanism meant to vouch for a release, sat on the safe harbor. Nearly a year later the cleaned and trusted line still carried hundreds of the operator's commits, because excising every contribution would have meant rewriting the project's history (Hess 2024, msg #200).

The deepest version of the problem is that a deliberately planted flaw can be indistinguishable from an honest mistake. The developer Joey Hess raised it within hours of the disclosure, arguing that the operator was "well placed to insert a buffer overflow" a crafted file could turn into "arbitrary code execution," and that "the impact of such a security hole could be much more stealthy and

bad than the known backdoor" because it could be denied as a bug (Hess 2024, msg #5). Lasse Collin's later line-by-line review found no such second payload (Collin 2024b), and Hess's worst case stayed a worst case; but the structural point holds, that a sufficiently subtle bug escapes attribution entirely by being deniable. The forensic responders ran into the same wall from the integrity side. The released archives matched their manifests, with "no evidence of tarballs being modified post-upload," Sam James reported, and yet that proof of integrity bought no confidence, because the question was no longer whether the files were altered but whether the person who signed them could be trusted: "I am still considering whether we should bring back the last release signed by Lasse Collin, rather than Jia Tan, to be safe" (James 2024a, comment 8). A signature can prove provenance. It cannot certify intent.

The strongest corrective to amateur forensics comes from the one person who corresponded with the operator for two years. Collin, the maintainer who had been deceived, went back through the malicious commits and published his findings, and his review repeatedly cools the readings that the public was most eager to make. On the question of forged identity, the foundation of any attribution theory, he was flat: "while the commits weren't signed, I didn't spot any signs of committer fraud" (Collin 2024b). The operator had not needed to forge anything, because a trusted co-maintainer's work is accepted on the strength of the relationship, not the cryptography. On the commit timezones that observers read as leaks of the operator's real location, Collin noted that "there are mundane explanations for these," routine `git` operations that rewrite the recorded time, adding that of the techniques that could produce the anomaly he "never did the last one but I don't know if Jia did" (Collin 2024b). And on one of the louder theories, that the operator had shortened the project's `SECURITY.md` disclosure window as tradecraft, to buy a backdoor more time before any flaw went public, Collin's account inverts the story. Simplifying the file had been his own initiative, and on the window itself it was Collin who wanted it shorter: "I felt 21–30 days would be appropriate but Jia wanted to keep 90" (Collin 2024b). The artifact read as the operator's cunning was, in the record,

the maintainer's housekeeping. The one glimpse of the operator's character cuts the other way and is the more chilling for it. Pressed about a harmless metadata oddity in a release archive, the kind of slip that invites an easy excuse, the operator would not take it. "The simplest explanation would have been that the tarball was created with `make dist` instead of `make mydist`," Collin wrote, naming the project's ordinary packaging command and the hardened variant it was supposed to use, "however, Jia didn't admit that it was such a simple error even though it would have been the easiest way to make me stop asking further questions" (Collin 2024b). It was discipline so complete that it refused even the convenient lie.

Set against that discipline, the leading theories are best handled the way the most careful analysts handled them: as a labeled set of possibilities, ordered by the strength of the evidence, never as a conclusion. Kaspersky, after laying out the artifacts, said as much itself: "reflecting on these data points still leads us to shaky ground. Until more details are publicized, we are left with speculation" (Kaspersky GReAT 2024a). The vendor then offered not one account but three. The operation might have been a small team that planted a malicious actor in the maintainer's seat; or a single individual running every persona, the pressure accounts and the helpful contributor alike; or, in the version that most unsettles the tidy narrative, a real contributor named in the record as "Jia Tan" who "legitimately earned access" and was only later manipulated or co-opted, with the harassing strangers possibly a coincidence rather than a coordinated screen (Kaspersky GReAT 2024a). A vendor with the full artifact set could not decide between a team and a lone operator, or between a planted agent and a turned volunteer. That irresolution is the finding.

What the artifacts do support, and only at the level of capability, is a register: sophisticated, professional, patient, and probably backed by a state. The reverse-engineering vendors reached for that language while refusing the next step. Akamai called the tradecraft state-grade and stopped: "such long-term operations are usually the realm of state-sponsored threat actors, but specific attribution does not currently exist" (Akamai Security Intelligence Group 2024).

The clearest articulation came from the security researcher Michał Zalewski, who argued from the operation's defining trait, its multi-year patience, that "if this timeline is correct, it's not the modus operandi of a hobbyist," since anyone with the skill and the patience to do this "can easily land a job that would set you for life without risking any prison time" (Zalewski 2024). His conclusion was a graded inference: "all signs point to this being a professional, for-pay operation," he wrote, "and it wouldn't be surprising if it was paid for by a state actor" (Zalewski 2024). The provenance of that last phrase is itself a small lesson in attribution discipline. Zalewski had first written "foreign government," then changed it; asked why, he explained in the comments that "given that it's a pretty weak hunch, and that it's tangential to the broader point, I stealth-edited the article to just say 'state actor' " (Zalewski 2024, lcamtuf comment, 2024-03-30). A confident-sounding claim, downgraded in public, with the reasoning shown. It is the same calibration Mandiant performs formally with its tiers of confidence, here done in the open by a single analyst.

The "probably state" reading rests on a recognizable kind of tradecraft, and the recognition has a history. The journalist Nicole Perlroth recorded the taxonomy of the veteran U.S. offensive-cyber figure James Gosler, for whom the apex of the field was occupied by "the Tier V and VI nation-states who spent years and billions of dollars finding mission-critical zero-days, developing them into exploits, and . . . inserting them into the global supply chain" (Perlroth 2021). Supply-chain insertion, in that account, is the signature of the very top tier, and the `xz` operation is structurally a supply-chain insertion that demanded exactly the years and the patience the tier implies. But the inference must be held at the level it lives on. Gosler was describing his era and never connected the taxonomy to `xz`; the resemblance is a reading, not a finding. The operation's footprint was also both vast and slow to map, which is part of why the reading took the shape it did. Surveying the incident days after the disclosure, the security researcher Kevin Beaumont noted that the "changes made by the threat actor on Github span multiple years," and that "several days in, despite global focus," he had not "seen anybody who has finished reverse engineering

it" (Beaumont 2024). And the discipline against over-reading craft into a culprit is itself a documented lesson. Greenberg, recounting the long hunt for the GRU's Sandworm, names "the attribution problem" as an ordinary condition of network operations rather than a coyness, a fog of "proxies, misdirection, and sheer overwhelming geographic uncertainty," and quotes the analyst Rob Lee's warning that "the people who develop it are not always the same people who use it" (Greenberg 2019). The patient social engineering, the discipline, and the bespoke cryptography license a register, not a name. As Thomas Roccia, who reconstructed the operation for a DEF CON audience, framed it, this was "an undercover operation that lasted almost three years, which is very impressive in terms of technical details, but also the social-engineering aspect is very impressive as well" (Roccia 2024, 1:06). Impressive enough to read as professional; not legible enough to name.

The most basic uncertainty was acknowledged from the first days. As Dan Goodin wrote in the week of the disclosure, "it's unknown if there was ever a real-world person behind this username or if Jia Tan is a completely fabricated individual" (Goodin 2024), and an Electronic Frontier Foundation (EFF) analyst told *The Intercept* that there was no "indication yet whether this was state sponsored, a hacking group, a rogue developer, or any combination of the above" (Mazurov 2024). Two further years of work have not closed that sentence.

If any thread invites a verdict, it is the alias that did the most concrete damage. An account presenting as "Hans Jansen," writing from `hans-jansen162@outlook.com`, did two operation-critical things from a single email address. It had authored the indirect-function commits, a legitimate-looking optimization that lets a program pick the fastest version of a routine at startup, which the payload later used as its way into the build. And in March 2024, four days before the disclosure, the same account asked Debian to ship the backdoored release, calling it "my package": "I am looking for a sponsor for my package 'xz-utils'," it wrote, naming "Version : 5.6.1-0.1" and the project's own address as the upstream contact (Jansen 2024, msg #5, 2024-03-25). Named observers read the account as part of the operation rather than a coincidence.

Russ Cox noted that "Hans Jansen" surfaces in 2024 only to promote the poisoned release and otherwise has essentially no internet presence, "a likely sock puppet" (Cox 2024); Krebs found the address absent from any breach corpus, as with the other personas; *The Intercept* grouped it among accounts that looked like deliberate persona management (Mazurov 2024). The pattern that links the personas is mechanical. As Krebs put it, accounts betray a shared hand through "multiple accounts that match some kind of naming convention, provider, etc. over the years" (Krebs 2024, post 112197610429038229), and the operation's personas do: a recurring shape of a word plus digits at a free email provider, across "Jigar Kumar" and "Dennis Ens," the strangers who pressured Collin to hand over control, and "Hans Jansen." The reach may have extended past XZ Utils itself: a Linux kernel developer noted that the same actor was listed as a co-maintainer of the kernel's XZ Embedded implementation (Corbet 2024, sergey.senozhatsky comment, 2024-04-01). That shows access and role, not identity. All of this is documented behavior and attributed judgment, and the discipline is to keep it there. Even the FAQ that became the incident's reference work marked the limit: yes, "Hans Jansen" introduced the function-resolution mechanism and later pressed Debian to update, "but this is quite a common thing for eager users to do, so it's not necessarily nefarious" (James 2024b). The clustering shows coordination. It does not show identity, and the relationship between "Hans Jansen" and "Jia Tan," one person or several, planted or turned, stays exactly as unresolved as Kaspersky's three hypotheses leave it.

That the attribution stays open is not a failure to find an ending. It is the finding. The European Union's own cybersecurity agency, surveying supply-chain attacks before the XZ Utils backdoor existed, stated the general case plainly: "attribution of attackers is very hard, prone to error, imprecise and politically challenging, but not impossible" (European Union Agency for Cybersecurity (ENISA) 2021, 14). The agency's own numbers make non-attribution the ordinary outcome rather than the exception. Of twenty-four incidents it analyzed, ten were never attributed to a particular group, roughly 40%, and it attributed

even that much resolution mostly to time, since most of the unattributed cases were recent (European Union Agency for Cybersecurity (ENISA) 2021, 25). Engineered ambiguity, in the literature of these operations, is a result and not a reporting gap. Thomas Rid, the historian of disinformation, describes attribution as "underdetermined by observable evidence," a "judgment call" rather than a deduction (Rid 2020): the evidence does not compel a name, and supplying one anyway would be a different act from reporting one. The serious analyses converge on the same restraint from independent directions. The peer-reviewed software-engineering study of the attack declined to name anyone at all, telling its readers that "since our work focuses on the impact of the attack on software engineering practices rather than specific individuals, we have anonymized the actors involved" (Przymus and Durieux 2025, 92, fn 1). Unnamed is not one writer's omission. It is the steady state of the field.

There is a reason the medium produces that steady state, and it is older than this operation. Open-source collaboration binds people who never identify one another through any embodied, hard-to-fake channel. Karl Fogel, in the standard manual on running such projects, calls it "psychologically odd, because it involves tight cooperation between human beings who almost never get to identify each other by the most natural, intuitive methods: facial recognition first of all, but also sound of voice, posture, etc." (Fogel 2020, 98). A consistent screen name is the only face anyone has, and the culture's own etiquette blesses it: "use your real name for all interactions," Fogel advises, "or if for some reason you prefer pseudonymity, then make up a name and use it consistently" (Fogel 2020, 99). By that standard a steady, well-behaved alias is not a warning sign. It is good conduct, and "Jia Tan" satisfied the norm exactly. The operator's pseudonymous consistency was camouflage indistinguishable from the ordinary, which is why no one was positioned to read it as a tell. Clifford Stoll's hunt through the same kind of network a generation earlier ended, in 1989, with German authorities charging five named people with espionage and a trail that led to the KGB; that era could still walk the wire to a person. Stoll had already imagined a version of this adversary: a spy able to work cheaply

from inside his own country, with little personal risk and few diplomatic costs (Stoll 1989). The operator is that figure realized and the courtroom subtracted.

In the end the people closest to the incident did not need a name, and said so. Andres Freund, who caught the backdoor, found that the operator's identity was beside the point of the response: whether the malicious code came from a compromised account or a long con, he had not looked further, because "it didn't really play a role for the reaction" (Freund 2024c, 50:48). The responders treated attribution as out of scope on principle. "We do not want to speculate on the people behind this project," the incident FAQ stated; "this is not a productive use of our time, and law enforcement will be able to handle identifying those responsible. They are likely patching their systems too" (James 2024b), a line that assumes, quietly, a competent adversary reading the same disclosure as everyone else. Krebs's conclusion was that even the agencies with the deepest tools would likely "not tell us what they find (or don't find)" (Krebs 2024, post 112198143331078347). And the most direct datum of all is the maintainer's, recorded without speculation in the commit that ripped the backdoor out: "the maintainer who added the backdoor has disappeared" (Collin 2024a). The operator who spent two years earning a place at the center of the world's software walked away from it and left no one to charge. Roccia, whose job is to attribute, allowed that "I'm not sure if we will have the answer someday" (Freund and Roccia 2024). That an adversary can operate this deep inside critical infrastructure and never be named is not a loose end in the story. It is the trust argument in its sharpest form: the systems that hold up the digital world authenticate work, not people, and a patient operator who never breaks the etiquette can ride that gap as far as it goes. The openness is a finding about the limits of what the public record can know, not a license to fill the silence with a culprit, and the honest reading leaves room for the operations that were never caught at all. As one developer wrote into the disclosure thread on its first day, "it would be hubris to assume this is the first or only attempt to subvert an upstream so far" (Corbet 2024, alex comment, 2024-03-29).

Chapter 15

Trust as Infrastructure

Working infrastructure is the kind of thing nobody looks at. The sociologist Susan Leigh Star described it as "transparent to use, in the sense that it does not have to be reinvented each time or assembled for each task, but invisibly supports those tasks" (Star 1999, 381). A road, the power grid, the plumbing inside a wall: each is noticed only when it stops working. "The normally invisible quality of working infrastructure becomes visible when it breaks," Star wrote, "the server is down, the bridge washes out, there is a power blackout" (Star 1999, 382). Trust is infrastructure in exactly this sense. An encrypted login, a signed software release, a maintainer's authority to commit code on behalf of millions of strangers: each is relied on continuously and re-examined almost never, and each becomes visible only at the moment it fails. As Nadia Eghbal put the everyday version, "Most of us take opening a software application for granted, the way we take turning on the lights for granted. We don't think about the human capital necessary to make that happen" (Eghbal 2016, 9).

The half-second of unexplained latency that Andres Freund chased across a test machine was that kind of breakdown. `sshd`, the program that handles encrypted remote logins on most of the world's servers, was running a fraction slower than it should, and the fraction led back to a backdoor hidden in a compression library widely shipped across Linux systems and, on the vulnerable

path, loaded into `sshd`. What the slowdown exposed was not a single defect but an entire layer the digital economy depends on and seldom audits. The legal scholar Jonathan Zittrain had already named that layer, placing above the network's physical and software layers a "social layer, where new behaviors and interactions among people are enabled by the technologies underneath" (Zittrain 2008, 67). The move that the XZ Utils story forces, from the code to the people, is a move into a layer charted long before the incident.

That the social layer is where attacks land was also known beforehand, and measured. Surveying supply-chain compromises in 2021, the European Union Agency for Cybersecurity (ENISA) found that "Around 62% of the attacks on customers took advantage of their trust in their supplier" (European Union Agency for Cybersecurity (ENISA) 2021, 3): the dominant route in was the trust relationship, not a flaw in anyone's code, and the figure comes from a dataset assembled three years before the backdoor existed. Russ Cox had stated the underlying condition even earlier. Writing in 2019 about the modern habit of building software out of hundreds of borrowed packages, he observed that "Developers trust more code with less justification for doing so" (Cox 2019, 38). What holds such cooperation together, Yochai Benkler observed in his study of peer production, is "some combination of technical architecture, social norms, legal rules, and a technically backed hierarchy that is validated by social norms" (Benkler 2006, 104): even the technical safeguards are, in the end, licensed by the community's trust. The mechanisms by which that trust is extended run in a sequence. An alias becomes a contributor; a contributor becomes a maintainer; a maintainer's authority becomes a signed release; a release becomes risk on machines its author will never see; and, when something finally breaks, public disclosure runs the chain in reverse. Each link is a trust relationship, and the XZ Utils record exposes every one.

The chain holds together on reputation, and reputation came to bear more weight than it was ever built for. The older supports fell away first. "The commercial and legal support for trusting software sources was replaced by reputational support," Cox wrote of the shift from bought software to the borrowed

kind (Cox 2019, 37). Reputation is cheaper than contracts and faster than audits, and it can also be manufactured. The political economist Elinor Ostrom, who spent a career studying communities that governed shared resources for generations without collapse, found that reputation and good faith were necessary but never sufficient on their own. She concluded that "reputation and shared norms are insufficient by themselves to produce stable cooperative behavior over the long run. If they had been sufficient, appropriators could have avoided investing resources in monitoring and sanctioning activities" (Ostrom 1990, 93–94). The long-lived commons she studied paid for monitoring and graduated penalties; the open-source commons, by and large, did not, substituting the maintainer's accumulated reputation for any system that watched the maintainer. That absence was less negligence than ancestry. Most of these projects, the anthropologist Gabriella Coleman noted, begin "without formal procedures of governance" and run instead on "the technical judgments of a small group of participants," the informal technocracy that the internet pioneer David Clark distilled in the maxim "rough consensus and running code" (Coleman 2013, 125). Formal controls, where they arrive at all, are grafted on later; the trust comes first, because at the start the trust is the project. What ran in monitoring's place was an assumption of good faith that nobody had to earn twice. The early internet, Zittrain wrote, had outpaced safer designs by assuming that every user was contributing a "goodwill subsidy: people would not behave destructively even when there were no easy ways to monitor or stop them" (Zittrain 2008, 9). The operation drew on exactly that subsidy, goodwill the community extended without an easy way to verify it or take it back. A peer-reviewed survey of these attacks, published before the operation began, located the danger in precisely that gap, in a project's "numerous trust boundaries" rather than in its code (Ohm et al. 2020, 6).

The first link is identity, and in this world an identity is thinner than it looks. It is a username, an account, a signing key: a handle whose behavior is entirely public and whose person is never verified, what researchers studying cryptocurrency called "pseudo-anonymous" (Meiklejohn et al. 2013, 127).

The people behind these handles rarely meet. When the 2020 FOSS Contributor Survey, a broad census of people who work on free and open-source software, asked how often respondents had met their project partners face to face, the most common answer, given by 47.03%, was "Never" (Nagle, Wheeler, et al. 2020, 54). The medium itself strips away the cues, face, voice, bearing, by which people ordinarily size one another up, leaving a consistent screen name as the only face anyone has.

The verification that does exist verifies less than it appears to. Even projects that encourage contributors to cryptographically sign their commits, the same survey noted, do so only to establish who, "by real name or username," is proposing a change (Nagle, Wheeler, et al. 2020, 64), and about half of projects required nothing at all. A signature of that kind binds a key to an account; it cannot bind an account to good faith. Hardening the identity layer further runs into a wall that the disclosure-day discussion mapped within days. Real-world identity, as one developer put it, does not establish trust in the first place: "It's not about whether they can be trusted, but about whether they can be held accountable if they do something bad" (Corbet 2024, draco comment, 2024-04-04). And accountability is exactly what a state-grade adversary can arrange around. As the developer Matthew Garrett argued in the same thread, against state-level attackers "we should assume that they're going to be able to produce ID that's good enough to pass any viable non-government checks," which leaves a regime that "deters legitimate contributors without preventing the worst case failures" (Corbet 2024, mjg59 comment, 2024-04-01). The control that would screen out the operator screens out the volunteers instead.

What the identity layer cannot supply, reputation does. Authority in these projects is earned by visible work over time and held by consent, not by contract or by ownership. Linus Torvalds described the arrangement from the inside, plainly: people trust his version of the Linux kernel "because they've seen me work for nine years on it," and "the only reason they do is that so far I've been trustworthy" (Torvalds and Diamond 2001, 189). He named the concentration too: "I control the Linux kernel, the foundation of it all, because, so far,

everybody connected with Linux trusts me more than they trust anyone else" (Torvalds and Diamond 2001, 168). For the kernel, that single point of trust is backed by a deep bench of co-maintainers; for legitimate XZ Utils maintainership before the handoff, the bench had effectively narrowed to Lasse Collin alone. The difference is not in the trust mechanism, which is the same across the commons, but in how much backup stands behind it. Benkler named that structure: "Torvalds's authority is persuasive, not legal or technical" (Benkler 2006, 105). Coleman called the expectation around it "meritocratic trust," the assumption that those entrusted with authority "act in good technical faith and not for personal interest" (Coleman 2013, 127). This reputational authority is also the one asset in an open project that cannot be defended by copying it. Code can be forked, mirrored, and restored from a thousand backups; the social capital cannot. As Karl Fogel observed in the standard manual on running these projects, "attention, credibility, and influence in the project very much are: they are by definition not copyable, and therefore not forkable" (Fogel 2020, 136). The one thing that cannot be backed up is therefore the one thing worth attacking, and the operation attacked it not by stealing a reputation but by building one. The community treats attributed credit as something close to sacred (Lerner and Tirole 2005, 61); the operator, working patiently as "Jia Tan," simply earned the credit, satisfying the machinery that converts visible work into trust from the one direction no one watches.

Reputation cashed out as a privilege: the authority to cut and sign a release. Where that privilege is well defended, it leaves a record. In the Linux kernel, every patch accumulates a chain of `Signed-off-by` lines, each maintainer adding one as a change moves upward, so that, in the project's own description, "the sequence of signoff lines can be used to establish the path by which each change got into the kernel" (Corbet and Kroah-Hartman 2016, 14); by 2020 each commit carried two such tags on average, one for each level of the hierarchy it had passed through (Linux Foundation 2020, 15). The XZ Utils backdoor never touched that kind of record, because it did not travel through the reviewed `git` history. It rode in the release tarball, the packaged archive of source code that a

maintainer ships to the world and that, by long habit, almost no one compares against the public repository.

The cryptographic control that the policy world reaches for would not have helped, because it answers a different question. A draft federal guideline on software inventories notes that "integrity and authenticity are most often supported through signatures and public key infrastructure" (Cybersecurity and Infrastructure Security Agency 2025, 12): a signature proves that an artifact arrived unaltered from whoever signed it. It says nothing about whether that signer is honest. Practitioners know the limit. In one 2025 study of the software industry, engineers reported that "the identity of individuals signing open-source dependencies does not inherently establish trust" (Kalu et al. 2025, 88), and one put the everyday reality more bluntly: "everybody on the team does sign their commits, but we don't really verify it" (Kalu et al. 2025, 88). Once the operator had legitimately become a signing maintainer of XZ Utils, every signature check on the poisoned releases would have passed, because the signatures were genuine. And the control is not merely bypassable; it is broken in ordinary use. In 2024, researchers audited four of the public registries developers fetch their dependencies from. On three of them, signatures failed to verify at rates of 24.0%, 53.1%, and 76.1%. Only Docker Hub, whose tooling refuses a bad signature at upload, recorded none (Schorlemmer et al. 2024).

The clearest measure of what the signature carried is what it took to withdraw it. Collin's own OpenPGP key, the cryptographic identity at the root of the project, had been certified by a signature from "Jia Tan," one key vouching for another in the web of mutual endorsement that key-based trust runs on. Restoring the project meant unpicking that endorsement by hand. The key, Sam James recorded after the cleanup, "is the same before but: 1) renewed; 2) dropped Jia's signature from it" (James 2024a, comment 46). The sequence the operation had climbed, identity to key to signature to release authority, had to be dismantled link by link.

From the signed release, trust runs outward, and it runs transitively: to rely on one thing is to rely, unawares, on everything it relies on. Zittrain had de-

scribed the shape of the problem in 2008 using web pages. "To visit a Web site is not only to be asked to trust the Web site operator," he wrote, but also to trust "every third party" whose content the page automatically pulls in, "and every fourth party" who in turn supplies that third party (Zittrain 2008, 56). Software dependencies are that pattern made load-bearing. Every system that loaded `liblzma`, the compression library at issue, trusted the maintainer of `xz`, and every system downstream of those trusted them in turn, none of them positioned to check. Clifford Stoll had watched a smaller version of it in the 1980s: "Often, these networked computers had been arranged to trust each other. . . . The hacker exploited that trust to enter a half dozen computers" (Stoll 1989).

What made the modern version so efficient was that the scarce defense was not code but attention, and attention is unevenly distributed. The security researcher Kevin Beaumont put his finger on the asymmetry: the operators "didn't need to backdoor systemd, which has a rich development community paying attention." Instead "they relied on liblzma loading XZ, which is much further down the chain, where nobody was paying attention" (Beaumont 2024). The attack routed around the watched project and into the unwatched dependency that the watched one happened to load. Invisibility, in this layer, costs nothing in power. Geoffrey Bowker and Star, writing about the classifications buried in working infrastructure, made the point directly: "Classifications are powerful technologies. Embedded in working infrastructures they become relatively invisible without losing any of that power" (Bowker and Star 1999, 319). An unwatched compression library held precisely the control over `sshd` that a watched one would have; what it had shed was scrutiny, not power. The exposure is structural and worsening: "The old reasons for trusting dependencies are becoming less valid at exactly the same time there are more dependencies than ever," Cox warned (Cox 2019, 43). In a supply chain built this way, as Andy Greenberg wrote of a different catastrophe, "distance is no defense" (Greenberg 2019).

The chain that carried the attack outward is the same chain that carried the alarm back, and that symmetry is the part of the story most easily lost. The

openness that let a stranger accrue authority is also what let the backdoor be caught and understood. An Electronic Frontier Foundation (EFF) analyst told *The Intercept* that "the ability for the engineer to discover this backdoor before it was shipped was only possible due to the open nature of the project" (Mazurov 2024). Once Freund's alarm existed, the social layer reversed direction within hours. Another engineer, Florian Weimer, "first extracted the injected code in isolation" from the binary that Freund had only seen whole (Freund 2024b), and the distributed network of distributions routed the warning upward to the U.S. Cybersecurity and Infrastructure Security Agency (CISA). "CISA was notified by a distribution," Freund noted (Freund 2024b).

Public disclosure is itself a trust mechanism, deliberately built, and it comes with its own assumptions. The convention security researchers follow, coordinated disclosure, assumes a good-faith maintainer who can be warned privately and given time to fix a flaw before it goes public. In the Log4j emergency two years earlier, the U.S. Cyber Safety Review Board found, the researcher who reported the bug "acted responsibly by following a sound coordinated disclosure process" with the project (Cyber Safety Review Board 2022, 16). XZ Utils could not follow that script, because the listed security contact belonged to the operator. The project's reporting address, the address to which a discoverer was supposed to send a private warning, pointed to "Jia Tan." "I couldn't really . . . report it to the security contact, which was Jia at that point," Freund said afterward, "because that was clearly not gonna be helpful for anybody" (Freund 2024c, 48:04). With no trustworthy upstream to coordinate with, disclosure went straight to the public, immediately. That informal norm is now being written into law: the EU Cyber Resilience Act requires manufacturers to maintain coordinated vulnerability disclosure policies, including a channel for reporting "where requested anonymously" (European Parliament and Council of the European Union 2024, rec. 76).

And the loop, when it finally closed, closed on trust rather than on proof. Seven months after the disclosure, when a rebuilt release reappeared in Debian, a developer asked the only question that mattered: "Do we trust these newer

versions now?" (Hess 2024, msg #167). The answer turned not on a cryptographic guarantee but on a human judgment: "Yes. We started with 5.6.2 which was audited by upstream after the malicious party left" (Hess 2024, msg #172). The key had been renewed, the operator's certifying signature excised, and an audit by the returned maintainer had been judged trustworthy enough to build on. That a community could argue its way back to confidence was not new. Recounting an earlier governance crisis in Debian, Coleman found that the public fight over a perceived breach of trust "was also the very mechanism by which trust was rebuilt again" (Coleman 2013, 155): in these communities the argument is often the repair. The institutions that study these attacks named both halves of the lesson in a single breath after a similar social-engineering attempt was caught against a JavaScript project weeks later. Such attacks are "difficult to detect or protect against programmatically as they prey on a violation of trust," the Open Source Security Foundation (OpenSSF) and the OpenJS Foundation wrote, and yet that one had been "successfully averted by the OpenJS community" (Bender Ginn and Arasaratnam 2024). The social layer is the vulnerability and the immune response at once.

None of this is unique to open source, and pretending otherwise mistakes the lesson. Trust is the under-defended layer everywhere software is made. As Torvalds observed after the incident, the problem is not peculiar to the open-source kind: "it's true even in proprietary source: you depend on the trust in the company, but also within the company you depend on trusting your employees, and that trust can be violated" (Mastery Learning 2024, 0:55). Beaumont drew the same distinction: "all development has a risk of insider account abuse, and that includes open source" (Beaumont 2024). The proprietary world had already furnished the proof. When investigators at the security firm FireEye dissected the SolarWinds compromise, a state operation that rode a signed update from a trusted vendor into thousands of corporate and government networks, they located its edge not in a novel exploit but in tradecraft "focusing on evasion and leveraging inherent trust" (FireEye 2020). There the mechanism was a customer's trust in a signed vendor update rather than an ecosystem's

trust in an upstream maintainer, but the under-defended surface was the same. The failure is more precisely trusted-account misuse than a stolen password: the account is real, the authority genuine, and what gives way is the assumption underneath them. Nor is the problem likely to stay singular. Surveying the incident, Kaspersky relayed OpenSSF's assessment that the XZ Utils effort was "highly likely not an isolated incident," similar social-engineering attempts having already surfaced in other projects (Kaspersky GReAT 2024a).

The hard problem is not naming the danger but detecting it. Trust is extended by default, and, as Torvalds put it, "How to figure out when it's being violated is an open problem" (Mastery Learning 2024, 1:13), which is the monitoring Ostrom had warned a commons cannot do without. The half-second was detection by accident, not by design.

The answers now emerging point at behavior rather than at identity. Freund, reflecting afterward, found he had come to value "soft" boundaries, the kind that "aren't intuitively hard to get over" but "add the friction for noise," prized less for stopping an intruder than for raising the odds that an intrusion is noticed (Freund 2024c, 01:28:17). Torvalds foresaw "a lot of work being put into some kind of trust model" that flags "a new person," or "a person that is acting differently from before" (Mastery Learning 2024, 4:26). But a model that works by suspecting the newcomer cuts against the welcoming openness that the operation exploited and that the commons needs in order to function, and behavioral profiling asks ordinary, privacy-valuing contributors to pay a cost a disciplined operator will simply absorb. That tension, between defending the trust layer and preserving the participation that makes it worth defending, is the one the next round of fixes has to resolve.

The most thorough answer on offer refuses the premise of the whole arrangement. The Zero Trust model that the United States has directed its own agencies to adopt "eliminates implicit trust in any one element, node, or service and instead requires continuous verification" (Executive Office of the President 2021, sec. 10(k)). A federal network can be ordered to trust nothing by default. A commons cannot: the goodwill extended to an unproven contributor is not

a flaw in the system but the thing that lets new contributors exist at all, and a project that verified everyone continuously would no longer be the kind of project anyone volunteers for. Some of the people the system runs on reject even the vocabulary of the fix. The "supply chain" framing that the policy response takes for granted is, one developer wrote into the disclosure thread, "a completely unrealistic model of free software development that assumes a 'supply chain' and an avenue for contractual obligations that just does not exist, cannot exist and is deeply undesired by all of the people this industry runs on, those who publish their code online because it brings them joy" (Corbet 2024, atnot comment, 2024-04-05). The trust that holds up the digital world is real infrastructure, and it is the infrastructure least defended, hardest to monitor, and most resistant to the obvious repairs. What could actually defend it, without dismantling the openness that makes it worth defending, is the question the arriving wave of policy has to answer.

Chapter 16

What Holds It Up

Two years after Andres Freund's disclosure, the response to the XZ Utils backdoor had hardened into a stack of instruments a reader can recognize: a European regulation whose main obligations were still phasing in, a set of American secure-development frameworks whose procurement mandate had been rescinded, a push to standardize the list of every ingredient in a piece of software, and a handful of funds pointed at the people who keep the code alive. Each addresses something real. None, on its own, reaches the condition that produced the backdoor, which lived in the social layer of trust: the goodwill extended to a stranger, the authority handed to a maintainer, the release no one audited. A serious response would have to reach that layer, and the labor layer beneath it, without destroying the openness that makes the commons work in the first place. There are no clean fixes. The security researcher Kevin Beaumont said as much in the first week: "There are no easy fixes.. we should just try to reduce the risk and calmly work some solutions" (Beaumont 2024). What is left is to judge the responses on the table by a single question: which layer does each one actually touch, the law, the code and its metadata, market incentives, foundation capacity, maintainer labor, or social trust?

That judgment has to be made before the evidence to settle it exists. The responses are arriving faster than anyone can measure whether they work; the

impacts of public funding on open source, one survey of the field concedes, "remain poorly understood, with a lack of consensus on how to meaningfully measure them" (Osborne et al. 2024, 1). And each instrument is an ethical choice that carries risk rather than a villain to be unmasked. Geoffrey Bowker and Susan Leigh Star, writing about the standards buried in every working infrastructure, put the discipline plainly: a standard "is an ethical choice, and as such it is dangerous—not bad, but dangerous" (Bowker and Star 1999, 5–6). Openness is dangerous in exactly that sense, and so is every proposal to constrain it. The temptation the political economist Elinor Ostrom warned against, after a career spent watching communities govern shared resources, is to mistake a model for the world and address "proposals to governments that are conceived in their models as omnicompetent powers able to rectify the imperfections that exist in all field settings" (Ostrom 1990, 215). The omniscient fix is the move to decline. What remains is to name the criteria a real response would have to meet, and to say, instrument by instrument, what reaches the under-defended layers and what does not.

The American response is best told as an arc, because it rose and fell inside those two years. It began with the right instinct. Executive Order 14028, signed in May 2021, located precisely the category of software that matters, "software that performs functions critical to trust," the kind with "direct access to networking and computing resources" (Executive Office of the President 2021, sec. 4(a)), which describes `liblzma` wired into `sshd` almost exactly. But the same order diagnosed the problem as a property of "the development of commercial software," which it said "often lacks transparency" (Executive Office of the President 2021, sec. 4(a)). XZ Utils was the harder inverse case: public in the ordinary open-source sense, yet compromised through a release path whose most dangerous contents did not match the public git history. The order did reach for open source by name, requiring producers to attest to "the integrity and provenance of open source software used within any portion of a product" (Executive Office of the President 2021, sec. 4(e)(x)), the two properties the operation corrupted. But it placed that duty on the firm that bundles the code, never on the

unpaid maintainer who produces it, and the maintainer is where the operation happened. From the order came the Secure Software Development Framework, a catalog of practices that NIST was careful to call non-prescriptive, "not . . . a checklist to follow, but . . . a basis for planning and implementing a risk-based approach" (National Institute of Standards and Technology 2022, 3). A voluntary vocabulary, in other words, addressed to "software producers."

Then procurement gave the voluntary vocabulary teeth. A 2022 federal memorandum ruled that agencies "must only use software provided by software producers who can attest to complying with" the framework's practices (Office of Management and Budget 2022, sec. II): no signed self-attestation, no sale to the federal government. For three years that made the framework a condition of federal software purchasing. In January 2026 it was undone. A new memorandum rescinded the mandate, finding that its predecessor "imposed unproven and burdensome software accounting processes that prioritized compliance over genuine security investments" (Office of Management and Budget 2026, 1). The self-attestation form was not abolished but demoted to optional, a resource agencies "may choose to use" (Office of Management and Budget 2026, 1), and the software bill of materials, or SBOM (a machine-readable list of every component inside a piece of software), became one contractual tool agencies could ask for rather than a general federal requirement. The arc is the policy landscape's advances and reversals in miniature, and the rescission's own diagnosis, compliance bought at the expense of security, is the judgment the other instruments keep inviting, conceded by the policy that built the mandate in the first place.

The most recognizable remedy of all is the SBOM, which the order also introduced. The order reached for a homely metaphor, calling it "analogous to a list of ingredients on food packaging" (Executive Office of the President 2021, sec. 10(j)), and CISA's 2025 public-comment draft kept the same handle, an "ingredients list" for software (Cybersecurity and Infrastructure Security Agency 2025, 4). On the response side it earns the praise. Once a vulnerability is public, an inventory of what is installed where genuinely speeds the cleanup, "significantly

improving response time compared to organizations that did not have SBOMs" (Cybersecurity and Infrastructure Security Agency 2025, 12). But an ingredients list would not have caught this. Its mechanism is keyed to "a newly reported vulnerability" in a component that is "listed in the SBOM" (Cybersecurity and Infrastructure Security Agency 2025, 10), and before Freund's disclosure there was no reported vulnerability for an SBOM tool to match: the backdoor rode inside `liblzma`, a normal, trusted, correctly named dependency. A conformant inventory would have listed `xz` versions `5.6.0` and `5.6.1` as ordinary components, with valid hashes, and flagged nothing.

The standard says so itself. "SBOM will not resolve all software security and supply chain concerns" (Cybersecurity and Infrastructure Security Agency 2025, 4), the draft concedes; an earlier report had already granted that "there are no cybersecurity panaceas, and SBOM is no exception" (National Telecommunications and Information Administration 2021, 7). The design assumes good-faith error. It asks consumers to tolerate the occasional honest mistake, then draws one explicit line: "this tolerance should not apply to intentional obfuscation or willful ignorance" (National Telecommunications and Information Administration 2021, 13). Intentional obfuscation, a backdoor concealed in test files and a release tarball, is exactly the XZ Utils case, and the framework excludes it by fiat rather than by mechanism. The empirical record is blunter still. When the U.S. Cyber Safety Review Board examined the Log4j emergency, the canonical open-source catastrophe, it spoke with organizations already using SBOMs and reported that "none reported having leveraged them to identify vulnerable deployments of Log4j" (Cyber Safety Review Board 2022, 13). Knowing your ingredients is not the same as trusting the cook.

The European Union's Cyber Resilience Act is the one instrument that recognizes the structural fact the whole story turns on, that critical software is published rather than sold, kept alive by people and entities outside the market, and it answers that fact with a deliberately light hand. The regulation invents a legal category for it, the "open-source software steward," and subjects stewards to "a light-touch and tailor-made regulatory regime" (European Parliament and

Council of the European Union 2024, rec. 19), reaching the foundation layer the OpenSSF and the Linux Foundation occupy. But the steward's entire substantive duty is to "put in place and document in a verifiable manner a cybersecurity policy" (European Parliament and Council of the European Union 2024, art. 24(1)): a documentation duty, not a resourcing one, a requirement to have a policy rather than to fund a maintainer. The enforcement that can reach a commercial manufacturer, fines into the tens of millions of euros, is switched off entirely for stewards, by an explicit derogation under which "the administrative fines . . . shall not apply to . . . any infringement of this Regulation by open-source software stewards" (European Parliament and Council of the European Union 2024, art. 64(10)). The lone, unpaid contributor falls outside the regime altogether (European Parliament and Council of the European Union 2024, rec. 18).

The two readings of that softness, protective and toothless, are both available, and the choice between them is deliberate, because coercion applied to volunteers would break the openness the law is trying to defend. In force since December 2024, with reporting obligations beginning in September 2026 and full application in December 2027, the Act is still at the stage where its effect can only be guessed at. One developer caught the register exactly in the disclosure-day discussion: "I think the CRA is a step in the development of the business models that will improve the funding situation in the future but I don't think we yet know how this will work out" (Corbet 2024, kleptog comment, 2024-04-01). The law does one quiet thing worth marking. It opens a voluntary channel for reporting not only incidents but "near misses that could have resulted in such an incident" (European Parliament and Council of the European Union 2024, art. 15(2)). For an operation that was caught with half a second to spare, a regulatory category for the thing that almost happened is a small, apt grace note.

Money is the response that reaches furthest down. The funds are real and some of them are aimed squarely at the labor layer. Germany's Sovereign Tech Agency, the public financier whose 2024 study of open-source maintenance an

earlier chapter drew on, runs a Fellowship that pays "the typical work of maintainers up to €78,000 per year per individual" (Osborne et al. 2024, 7), a salary-scale grant to a named person rather than a bounty or a mandate; the OpenSSF-linked Alpha-Omega fund makes grants to critical projects; a 2026 Linux Foundation round named maintainer labor as its explicit target (Linux Foundation 2026). Where a project has an organization able to receive the money, it demonstrably works. The Rust programming language's foundation moved, on Alpha-Omega funding, from "three part-time volunteers . . . limited to reactive support" to staffed security and software-engineering roles (Alpha-Omega 2025a, 20). Money is not nothing.

But the funders name two failure modes themselves. The first is that the money routes to the legible core and misses the long tail, and the sharpest illustration is an irony the record hands over without comment: Alpha-Omega's stated mission is to protect "the most critical open source software projects and ecosystems" (Alpha-Omega 2024, 16), and its 2023 grant slate, published on February 16, 2024, six weeks before the disclosure, named Eclipse, Node.js, Rust, OpenSSL, the Linux kernel, and several others, and did not include `xz` (Alpha-Omega 2024, 14). The fund whose word was "critical" missed a library that, six weeks later, would become one of the year's defining examples of critical open-source risk. The second failure mode is durability. Grants end, and a one-time injection can leave a project more exposed than before; an evaluation of one public fund found that "over half of their funded projects struggled to secure follow-up funding" (Osborne et al. 2024, 15), a funding cliff a fellowship walks toward from its first day. Bounties, the market-shaped version of the remedy, can be worse than nothing, providing for organizations that skip the underlying work "an illusion of security" (Ellis and Bollampalli 2024, 31). The most disciplined judgment in the funding literature is also the simplest: invest in maintenance, not just in the market for bugs, because buttressing maintenance is not charity but, the report insists, an investment "in security" (Ellis and Bollampalli 2024, 37).

That is the layer every instrument keeps circling without landing on. Beau-

mont named the proportion in the same breath as his praise for the metadata work: "Let's just keep doing the good SBOM work at CISA, and stop doing stunts around Huawei and such — Huawei is a speck of dust compared to the issues around tens of thousands of unpaid developers writing the core of the world's most critical infrastructure nowadays" (Beaumont 2024). The endorsement and the limitation arrive together, from the same person, in the same sentence. The metadata is worth doing; it is also a rounding error against the human problem underneath it.

And the human layer cannot simply be told to do more. In the 2020 FOSS Contributor Survey, respondents reported that security already takes "an average of 2.27% of their total contribution time," and that contributors "do not desire to increase this significantly" (Nagle, Wheeler, et al. 2020, 5). The disinclination is cultural as much as economic. "I find the enterprise of security a soul-withering chore and a subject best left for the lawyers and process freaks," one respondent to that survey wrote; "I am an application developer" (Nagle, Wheeler, et al. 2020, 31). A separate study four years later found a core contributor to Ruby on Rails giving the same answer in three words about why volunteers avoid the work: "Because it sucks!" (Ellis and Bollampalli 2024, 24). Onto that disposition the new standards land badly. A majority of maintainers, 52%, said they "weren't aware of any of these new government and industry standards" at all (Tidelift 2023, 10), and asked why they would not comply, they ranked the two reasons in order: 38% had no time, 37% were not paid for it (Tidelift 2023, 14).

A compliance obligation dropped onto unpaid, time-starved maintainers does not add security; it adds what the same survey called "unfunded mandates" (Tidelift 2023, 8), work nobody is paying for. The cost is already measurable: "asked to comply with requirements I don't have time for" has become one of the named reasons maintainers cite for quitting (Tidelift 2023, 26). The mechanism by which a security regime exhausts the people it depends on is visible in the daily grind of vulnerability reports, where, one maintainer wrote, skewed incentives leave the maintainer "to do essentially all the work validating/disputing

findings" (Tidelift 2023, 36), much of it for alerts that turn out to be empty. The tools that generate those alerts are part of the problem: one study of a widely used dependency scanner found that 88.8% of its findings, among the vulnerabilities the authors could independently check, were false positives (Ponta et al. 2020, 3175). The reframe that the institutions themselves have begun to reach is the criterion any serious response has to meet. "Ensuring our maintainers are well supported," the OpenSSF and the OpenJS Foundation wrote after fending off a copycat attack, "is the primary deterrent we have against these social engineering attacks" (Bender Ginn and Arasaratnam 2024). Maintainer support is not welfare for the commons. It is a security control.

One finding runs underneath all of it: compliance is not security, and the policy wave largely does not reach the practice it names. Software is signed, but the signatures go unchecked. Engineers in one 2025 industry study described signing as a "check the box" technique performed to satisfy "regulatory, framework, customer or organizational requirements," not as something they believed made them safer (Kalu et al. 2025, 92). The hardest single piece of evidence is a measurement: a study of public software registries found that the SolarWinds compromise, "the subsequent executive order and NIST guidance" included, "had no discernible effect on signing adoption" (Schorlemmer et al. 2024). That compromise, plus the American policy apparatus built in its wake, moved the practice by an amount too small to see. What the frameworks did move was SBOM collection, not verification (Kalu et al. 2025, 92): the layers cheap to demonstrate advanced, the layers that require sustained checking did not. And the one control aimed squarely at this kind of attack is inert against it by construction. A 2021 European best-practice guide advised buyers to "receive assurance of suppliers and service providers that no hidden features or backdoors are knowingly included" (European Union Agency for Cybersecurity (ENISA) 2021, 27). The word doing the work is "knowingly." A supplier who is the adversary, an operator who has become the maintainer, can give that assurance in perfectly good-faith-looking form.

The technical fixes that do reach the artifact layer are real, and they close

a real seam. The disclosure provoked an immediate push to stop shipping the hand-curated release tarballs the backdoor had hidden in: "we need to stop using curated tarballs, only auto-generated from tags" (Corbet 2024, bluca comment, 2024-03-29), one developer proposed, building releases automatically from the public source instead. The rebuttal came within a day, and it is the reason the fix is partial: stop curating tarballs, another answered, and "developers will just add generated files into the SCM" (Corbet 2024, jengelh comment, 2024-03-30), relocating the unreviewed content rather than abolishing it. A third sketched the durable version, a build pipeline that takes "the tagged source code commit, generate[s] artifacts and sign[s] them so you do have the provenance" (Corbet 2024, gdamjan comment, 2024-03-30), an unbroken, checkable record from source to shipped file. The distro responders had already done it by hand. Restoring `xz` in Gentoo, Sam James "reproduced [Collin's] tarball with only minor differences in dates" and urged everyone downstream to do the same (James 2024a, comment 46), rebuilding the release from source to confirm the two matched. Dependency surgery followed in the same spirit. `systemd` worked to reduce what `libsystemd` pulls in (Przymus and Durieux 2025, 98), and in version `9.8`, released in July 2024, OpenSSH added support for notifying `systemd` through a standalone implementation that no longer needed `libsystemd` at all (OpenSSH 2024), removing the reason distributions had patched `sshd` to pull the library in, and `liblzma` with it.

Every one of these, though, makes the same assumption as commit signing: that the party who signs is the party to trust. That assumption is precisely the position the operation spent two years acquiring. The remedies reach the code, the metadata, and the procurement contract; they do not reach the layer of social trust where the backdoor was planted. Linus Torvalds, asked about the lesson, put the limit in one line: "when you have rules in place, the bad actors, they don't follow the rules" (Mastery Learning 2024, 3:25). Rules bind the honest and route around the adversary. The controls that might actually reach the trust layer, verifying the human behind a commit, flagging the newcomer whose behavior looks unusual, are the ones that cut against the welcoming openness

the commons runs on and the privacy of the ordinary contributor, who pays a cost a disciplined operator simply absorbs. A commons that verified everyone continuously would stop being a commons. That, and not any missing piece of metadata, is why the menu of responses leaves the structural condition standing. Clifford Stoll named the deeper hazard decades ago. A community like this one could be destroyed outright by an attack, he wrote, or, "worse, it could consume itself with mutual suspicion, tangle itself up in locks, security checkpoints, and surveillance; wither away by becoming so inaccessible and bureaucratic that nobody would want it anymore" (Stoll 1989). The remedy that misses the trust layer is one kind of failure. The remedy that reaches it by force is the other.

The double edge cuts even where the response was fastest. Within hours of the disclosure, GitHub suspended the operator's account and took down the XZ Utils repository. That removed the attacker, and with him the public record of every change the world's reverse engineers were at that moment racing to read; for a stretch the platform also suspended Collin's own account, restoring it only days later (Boehs 2024). Excising the actor and foreclosing the investigation were, briefly, the same act. Platform moderation reaches the hosting layer, decisively and at once, and a hosting layer is not where the trust failure lived either.

It also leaves a residue, which is the first reason the story refuses a clean ending. The distributions reverted within days; the standing advice from CISA, relayed by Akamai, was simply to "downgrade to an uncompromised version, such as 5.4.6" (Akamai Security Intelligence Group 2024), and most of the world did. But the backdoor did not leave when the patch arrived. More than a year after the disclosure, twelve official Debian base images on Docker Hub still contained it, and they remained, the researchers who found them noted, because the artifacts "persist in container registries for a very long time" (Binarly REsearch 2025). They stayed not by oversight but by a decision. Asked to pull the images, a maintainer of the Docker library explained that the team had made "an intentional choice to leave these artifacts available as a historical curiosity" (Haruyama 2025, comment 2025-08-08), reasoning that the affected builds

were old, never meant for production, and exploitable only by whoever held the backdoor's key. The judgment is defensible and the risk is narrow; it is also a deliberate human choice to keep a state-grade backdoor downloadable into 2026. A removed backdoor can stay published as long as an installed vulnerability stays installed. The Cyber Safety Review Board had used the same word for Log4j, calling it an "endemic vulnerability" likely to persist "perhaps a decade or longer" (Cyber Safety Review Board 2022, v). The true tempo of the aftermath is not a clean closure but a slow administrative wind-down, long after public attention has moved on.

The second reason is the catch itself. It hung on half a second of latency on a test machine, and the contingency was real. "It was found randomly," Torvalds observed; "random ends up being good" (Mastery Learning 2024, 3:01). That is the line to interrogate, not to adopt, because a defense that depends on luck is not a defense. But the randomness was not quite pure. The slowness Freund noticed existed partly because someone, for unrelated reasons, had been working to reduce `sshd`'s dependencies and shrink its exposed surface, an effort that plausibly forced the operator to rush and produce the very latency that gave the operation away. Freund drew the lesson himself: the episode "shows the value of doing some prospective security work to reduce your exposed code areas" (Freund and Roccia 2024). And there were, as Torvalds noted, no gates designed to catch this, and yet "they were actually really caught fairly quickly" (Mastery Learning 2024, 2:30). What worked was not a control but the distributed, public system functioning as it is built to: many hands, an open record, a stranger's unrelated diligence, and one engineer who chased an anomaly nobody had asked him to chase. Jonathan Zittrain had described the only defense a system like this really has, years before the operation began. Security in such a system, he wrote, "requires the continuing ingenuity of a few experts who want it to work well, and the broader participation of others with the goodwill to outweigh the actions of a minority determined to abuse it" (Zittrain 2008, 166). Freund was the expert; the distributions and the reverse engineers were the goodwill; "Jia Tan" was the minority. Contingency met competence. But that is a description

of an immune response, not a guarantee, and an immune response can fail; it is not the same as a system designed to stop the next one, and the honesty of that "almost" depends on keeping the two apart.

Ten days after the disclosure, Collin was reverting the operator's changes one commit at a time, with the same unhurried care that had run the project for years. One of those commits removed a piece of the backdoor and listed, among its reasons, simply: "Backdoors are bad for security" (Collin 2024a). He put his project back in order, and he maintains it still.

The question that should hang over all of it is not Collin's but one a security researcher left ringing at the end of a conference talk, after walking an audience through every move of the operation. "How many other Jia Tans," Thomas Roccia asked, "do you think are paid to insert backdoors in open-source projects?" (Roccia 2024, 41:27). These instruments are the world's first attempt at an answer, and none of them reaches the place the question comes from. The half-second held. There is no reason yet to believe the next one will.

Glossary

advanced persistent threat (APT) A well-resourced, persistent intruder, usually state-backed, that works for long-term access rather than a quick break-in. Security firms track such groups under numbered labels they assign rather than names the group chose, such as APT44 (also known as Sandworm). The XZ Utils operator fits the profile of a capable, patient adversary but, unlike a tracked APT, was never identified.

backdoor A hidden piece of code that lets someone bypass normal authentication and gain secret access to a system.

benevolent dictator The founder or lead maintainer of an open-source project who holds final say over which changes are accepted, an authority that rests on accumulated trust and reputation rather than any formal office. Linus Torvalds is the original example, with the last word over the Linux kernel; the term generalizes to any project organized around a single trusted decision-maker. The arrangement concentrates a project's direction, and its safety, in one person.

build The process of turning human-readable source code into the program a computer actually runs.

bus factor How many people would have to abruptly drop away from a project (the grim image is being hit by a bus) before it stalls because no one left understands the code. A bus factor of one means the whole project lives in a single maintainer's head, as `xz` did.

CNA CVE Numbering Authority: an organization authorized to assign CVE

identifiers for vulnerabilities within its own scope. Red Hat acted as the CNA for the XZ Utils backdoor, assigning it CVE-2024-3094.

commit A single recorded change to a project's code, with an author and a timestamp, stored in the project's history.

committer A trusted project contributor with permission to write changes directly into the project's code repository. A committer is not necessarily the sole maintainer, but has been given authority over the project's history.

configure script A build-time script, often named `configure`, that checks a machine and prepares source code to be compiled. In the backdoored `xz` releases, malicious commands were hidden so they ran during this preparation step, before the library was built.

CVE Common Vulnerabilities and Exposures: the public catalog of known security flaws, and the unique number it gives each one (the XZ Utils backdoor is CVE-2024-3094).

CVSS Common Vulnerability Scoring System: a standard scale from 0 to 10 that rates how severe a security flaw is. The XZ Utils backdoor was scored 10.0, the maximum, and rated "critical."

CWE Common Weakness Enumeration: a catalog of the kinds of software weakness, as distinct from the specific cases recorded as CVEs. The XZ Utils backdoor is classed CWE-506, embedded malicious code, the category for code planted on purpose rather than a flaw introduced by mistake.

Debian Free Software Guidelines (DFSG) Debian's criteria for what may ship as free software in its main archive, among them that a package provide real, modifiable source rather than opaque binaries. The XZ Utils payload was hidden as binary blobs disguised as test files, with no source behind them; when Lasse Collin removed the backdoor, he named this violation of the guidelines first, before the security harm.

dependency A piece of software that another program needs in order to work.

distribution (distro) A complete, packaged operating system (Debian, Fedora,

Ubuntu) assembled from thousands of open-source components.

downstream The direction a piece of software travels after its upstream project, out to the distributions that repackage it and the users who run it. The XZ Utils backdoor was caught while still moving downstream, before it reached stable releases.

dynamic linker The part of a Linux system that connects a running program to the shared libraries it needs. The XZ Utils backdoor abused this machinery to reach functions used by `sshd`.

exploit A technique or piece of code that turns a vulnerability into a working attack.

IFUNC A GNU C Library mechanism that lets a program choose between different implementations of a function while a shared library is being loaded. In the XZ Utils incident, a legitimate performance feature became one of the places the backdoor could take control.

long-term support (LTS) kernel A kernel release singled out to receive security and bug fixes for years rather than months, so distributions can build on a stable base without chasing every new version. Most everyday Linux systems run a kernel derived from one of these long-lived releases.

mainline The canonical, official line of a project's code that its lead maintainers release from, as opposed to the many side branches where work happens first. In the Linux kernel, a change reaches the mainline only after passing through a subsystem maintainer.

maintainer The person, or small group, responsible for keeping a piece of software working: reviewing changes, fixing bugs, and publishing releases. Often unpaid.

NOBUS "Nobody but us": a backdoor or vulnerability built so that only the party who planted it can trigger it, because using it requires a secret they alone hold (in the XZ Utils case, the operator's private signing key). A term from the intelligence world. The XZ Utils backdoor was NOBUS by design, which is why researchers who had the code still could not turn it against others, and why observing one attack would not let a defender

reuse it elsewhere.

non-maintainer upload (NMU) In Debian, a package upload made by someone other than the package's listed maintainer, usually to fix a problem when the maintainer is unavailable or a change is urgent. In the XZ Utils incident, pressure around an NMU helped push the backdoored upstream release toward Debian.

object file A compiled fragment of code that is not yet a complete program or library. In the backdoored `xz` releases, the hidden payload was extracted as an object file and linked into `liblzma` during the build.

open source Software whose source code is public and free to use, study, modify, and share. The phrase is also a defined standard: the Open Source Initiative's Open Source Definition sets out ten criteria a license must meet, among them no discrimination against any person or group and no restriction on the field of use. A license that lets anyone read the code but limits who may use it, or for what, is "source-available" rather than open source, a distinction the AI industry's recent claims on the word have blurred.

open-source software steward A legal category created by the EU Cyber Resilience Act for an organization, often a foundation, that sustains a free or open-source project relied on by commercial products without selling the software itself. Stewards carry lighter obligations than commercial manufacturers and, under the Act, face no fines.

package confusion attack A class of supply-chain attack that gets a victim to install malicious code by publishing it under a name close to, or easily mistaken for, a package they meant to use; typosquatting is its commonest form. The book notes the category mainly to mark what the XZ Utils attack was not: not a lookalike planted under a deceptive name, but a real, trusted library quietly corrupted from within.

package manager A tool that automatically downloads, installs, and updates the software packages a program or system depends on, pulling in their own dependencies in turn. On Linux distributions it is the machinery

(`apt` on Debian and Ubuntu, `dnf` on Fedora) that carries code from an upstream release down to a user's machine.

payload The part of an attack that does the actual damage, as opposed to the parts that deliver or conceal it.

provenance A verifiable record of where a piece of code came from and everything done to it along the way: its origin, who has owned and changed it, and when. The XZ Utils backdoor lived in the gap between a project's public source and the release tarball built from it.

release channel A distribution's lane for software at a particular stability level, such as unstable, testing, pre-release, or stable. The XZ Utils backdoor reached some unstable and pre-release channels, which is different from reaching stable systems.

release tarball The packaged archive a project publishes for people to download and build. It can differ from what is in the public code repository, which is where the XZ Utils backdoor hid.

remote code execution (RCE) The ability to make a computer run commands from another machine. In the XZ Utils case, the concern was pre-authentication remote code execution: an attacker with the right key material could make `sshd` execute a command before normal login had succeeded.

repository (repo) The version-controlled store of a project's code and its full history of commits.

rolling release A distribution that ships a continuous stream of updates rather than discrete, versioned editions, so its users run new software soon after it is published. openSUSE Tumbleweed is one; fast-moving channels like it took the backdoored `xz` quickly, while slower fixed-release editions did not.

Sandworm The common name for the Russian military-intelligence cyber group that Mandiant designates APT44 (see advanced persistent threat). The book uses it as a contrast: a state attacker that has been publicly attributed, against the XZ Utils operator, who has not.

SBOM Software Bill of Materials: an inventory of the components inside a piece of software, and how they depend on one another, meant to make its supply chain auditable. It records which components are present, not whether a trusted one has been tampered with.

self-attestation A software producer's own signed declaration that it follows specified secure-development practices, accepted by a buyer in place of independent verification. The U.S. government required it for federal software purchases from 2022 until the rule was rescinded in 2026; a producer could instead submit a third-party assessment, but in the usual case the assurance rested on a signature, not an audit.

Signed-off-by A line a developer adds to a kernel change to certify, by name, that it may legally be included; the sequence of these lines records the chain of hands a change passed through on its way in. It documents who vouched for code, the kind of trust record the XZ Utils release process, built outside the public repository, bypassed.

social engineering The practice of manipulating people, rather than breaking software, to win trust or access.

software supply chain The full chain of code, people, and tools a finished program depends on. A weakness anywhere along it, a single unmaintained library or one trusted contributor, can compromise everything built on top.

SSH / sshd Secure Shell, the standard way to log into a remote computer; `sshd` is the program that listens for those logins on the server.

SSH certificate / certificate authority (CA) An SSH certificate is an ordinary SSH key bundled with identity and validity information and signed by a separate key, a certificate authority (CA), that the server has been configured to trust. Presenting one makes the server read the CA's public key out of the certificate and verify its signature before a login can succeed. The XZ Utils backdoor used this normal step as its delivery path: a crafted certificate carried an attacker-controlled CA key whose modulus hid the encrypted command, so that the act of verifying it reached the hooked

function in `sshd`.

stable tree / stable updates The stream of fixes applied to a kernel after its initial release, maintained separately from new feature work. These stable updates are the base from which most distributions build the kernel they actually ship, one of the links in the chain that would have carried the XZ Utils backdoor downstream.

subsystem maintainer / subsystem tree A maintainer who controls one part of the kernel (networking, a driver family, a filesystem) and the code branch, the subsystem tree, through which changes to that part flow before reaching the mainline. The kernel spreads authority across more than a hundred such maintainers, the opposite of the single-maintainer arrangement that left `xz` exposed.

systemd / libsystemd `systemd` is the service manager used by many Linux distributions to start and supervise system processes. `libsystemd` is one of its libraries; in some distributions, patches caused `sshd` to load `libsystemd`, which in turn made `liblzma` reachable during SSH server startup.

test fixture A file or input used to check that software behaves correctly. In the XZ Utils backdoor, malicious code was hidden inside binary test fixtures, a place reviewers were less likely to inspect by hand.

tragedy of the commons The idea that a resource open to everyone and owned by no one is ruined by overuse, because each user gains from taking more while the cost of depletion is shared by all. Coined for grazing pastures and fisheries, the phrase is often stretched to open-source software; this book treats the collapse as a risk to be governed rather than an inevitable fate, and locates the resource that can actually be used up not in the code, which no one depletes by copying, but in the finite attention and trust of the maintainers who keep it alive.

transitive dependency A dependency reached through another dependency, rather than one a program requires directly. Such indirect dependencies are easy to overlook, and a flaw in one is as dangerous as a flaw in code

you chose yourself. The XZ Utils backdoor rode into `sshd` as a transitive dependency, by way of `libsystemd` and `liblzma`.

trusted-account misuse The use of an account or identity that a project has learned to trust in a way that turns that trust against the project. In the XZ Utils incident, the danger was not only malicious code but the authority attached to a contributor identity that had become ordinary.

typosquatting Registering a malicious package under a near-miss spelling of a popular one (`reqeusts` for `requests`), so that a mistyped or misremembered name installs the attacker's copy. The commonest form of a package confusion attack, and another thing the XZ Utils backdoor was not.

upstream The original project a piece of software comes from. Linux distributions pull code "downstream" from upstream projects, repackage it, and ship it to users; the XZ Utils backdoor was planted upstream and was weeks from flowing down into stable Debian, Fedora, and Ubuntu.

valgrind A debugging tool that watches a program while it runs and reports memory errors and other abnormal behavior. `valgrind` errors were among the odd symptoms that helped expose the XZ Utils backdoor.

VEX Vulnerability Exploitability eXchange: a machine-readable statement that a known vulnerability in a bundled component does not actually put a product at risk, because the flawed code is never reached or is not built in. It distinguishes a flaw being present from its being exploitable.

vulnerability A flaw in software that can be abused to make it behave in ways its users never intended.

wiper Malware built to destroy, overwriting files or a disk's startup records, rather than to steal data or hold it for ransom. The XZ Utils backdoor was not a wiper; the term appears where the book contrasts it with the destructive operations of state cyber actors.

xz / liblzma A widely used compression tool (`xz`) and the library beneath it (`liblzma`), installed on nearly every Linux system.

Zero Trust A security approach that gives nothing standing trust and re-

verifies every request for access, on the assumption that an attacker may already be inside the network. It is built for machine and network access and does not reach the human trust that open-source maintenance runs on.

Bibliography

Akamai Security Intelligence Group. 2024. *XZ Utils Backdoor — Everything You Need to Know, and What You Can Do.* https://www.akamai.com/blog/security-research/critical-linux-backdoor-xz-utils-discovered-what-to-know.

Alpha-Omega. 2022. *2022 Annual Report.* Open Source Security Foundation (OpenSSF). https://openssf.org/wp-content/uploads/2022/12/OpenSSF-Alpha-Omega-Annual-Report-2022.pdf.

Alpha-Omega. 2024. *2023 Annual Report.* Open Source Security Foundation (OpenSSF). https://alpha-omega.dev/wp-content/uploads/sites/22/2024/02/Alpha-Omega-Annual-Report-2023.pdf.

Alpha-Omega. 2025a. *2024 Annual Report.* Open Source Security Foundation (OpenSSF). https://alpha-omega.dev/wp-content/uploads/sites/22/2025/01/Alpha-Omega-Annual-Report-2024_012925.pdf.

Alpha-Omega. 2025b. "Grant Recipients." https://alpha-omega.dev/grants/grantrecipients/.

Alpha-Omega. n.d. "About Alpha-Omega." Accessed May 31, 2026. https://alpha-omega.dev/about/about-alpha-omega/.

Ayala, Jessy, Yu-Jye Tung, and Joshua Garcia. 2025. "A Mixed-Methods Study of Open-Source Software Maintainers on Vulnerability Management and Platform Security Features." *34th USENIX Security Symposium (USENIX Security 25)* (Seattle, WA), August, 2105–24. https://www.usenix.org/conference/usenixsecurity25/presentation/ayala.

Beaumont, Kevin. 2024. *Inside the Failed Attempt to Backdoor SSH Globally — That Got Caught by Chance*. https://doublepulsar.com/inside-the-failed-attempt-to-backdoor-ssh-globally-that-got-caught-by-chance-bbfe628fafdd.

Bender Ginn, Robin, and Omkhar Arasaratnam. 2024. *XZ Utils Cyberattack Likely Not an Isolated Incident*. Open Source Security Foundation (OpenSSF) and OpenJS Foundation. https://openssf.org/blog/2024/04/15/open-source-security-openssf-and-openjs-foundations-issue-alert-for-social-engineering-takeovers-of-open-source-projects/.

Benkler, Yochai. 2006. *The Wealth of Networks: How Social Production Transforms Markets and Freedom*. Yale University Press.

Binarly REsearch. 2025. *Persistent Risk: XZ Utils Backdoor Still Lurking in Docker Images*. Binarly. https://www.binarly.io/blog/persistent-risk-xz-utils-backdoor-still-lurking-docker.

Boehs, Evan. 2024. *Everything I Know about the XZ Backdoor*. https://boehs.org/node/everything-i-know-about-the-xz-backdoor.

Bonaccorso, Salvatore. 2024. *xz-utils Security Update*. Debian Security Advisory DSA-5649-1. Debian Project. https://lists.debian.org/debian-security-announce/2024/msg00057.html.

Bowker, Geoffrey C., and Susan Leigh Star. 1999. *Sorting Things Out: Classification and Its Consequences*. MIT Press.

Buchanan, Ben. 2020. *The Hacker and the State: Cyber Attacks and the New Normal of Geopolitics*. Harvard University Press.

Chance, Gloria. 2023. *Tips and Strategies for Reducing Stress and Burnout by Creating Psychological Safety*. https://lpc.events/event/17/sessions/153/.

Coleman, E. Gabriella. 2013. *Coding Freedom: The Ethics and Aesthetics of Hacking*. Princeton University Press.

Collin, Lasse. 2022. *Re: [Xz-Devel] XZ for Java*. https://www.mail-archive.com/xz-devel@tukaani.org/msg00567.html.

Collin, Lasse. 2024a. *Remove the Backdoor Found in 5.6.0 and 5.6.1 (CVE-2024-3094).* Git commit e93e13c8b3bec925c56e0c0b675d8000a0f7f754 in tukaani-project/xz. GitHub. https://github.com/tukaani-project/xz/commit/e93e13c8b3bec925c56e0c0b675d8000a0f7f754.

Collin, Lasse. 2024b. *XZ Utils Review Notes.* https://tukaani.org/xz-backdoor/review.html.

Collin, Lasse. n.d. *XZ Utils Backdoor.* Accessed May 31, 2025. https://tukaani.org/xz-backdoor/.

Corbet, Jonathan. 2024. *A Backdoor in xz.* LWN.net. https://lwn.net/Articles/967180/.

Corbet, Jonathan, and Greg Kroah-Hartman. 2016. *Linux Kernel Development: How Fast It Is Going, Who Is Doing It, What They Are Doing and Who Is Sponsoring the Work.* Linux Foundation. https://www.linuxfoundation.org/hubfs/lf_pub_whowriteslinux2016.pdf.

Cox, Russ. 2019. "Surviving Software Dependencies." *Communications of the ACM* 62 (9): 36–43. https://doi.org/10.1145/3347446.

Cox, Russ. 2024. *Timeline of the Xz Open Source Attack.* https://research.swtch.com/xz-timeline.

Cusumano, Michael A. 2005. "Foreword." In *Perspectives on Free and Open Source Software*, edited by Joseph Feller, Brian Fitzgerald, Scott A. Hissam, and Karim R. Lakhani. MIT Press.

CVE Program. 2024. *Xz: Malicious Code in Distributed Source.* CVE Record. CVE Program. https://www.cve.org/CVERecord?id=CVE-2024-3094.

Cyber Safety Review Board. 2022. *Review of the December 2021 Log4j Event.* U.S. Department of Homeland Security. https://www.cisa.gov/sites/default/files/2023-02/CSRB-Report-on-Log4j-PublicReport-July-11-2022-508-Compliant.pdf.

Cybersecurity and Infrastructure Security Agency. 2023. *CISA Open Source Software Security Roadmap.* U.S. Department of Homeland Security. https://www.cisa.gov/sites/default/files/2023-09/CISA-Open-Source-Software-Security-Roadmap-508c.pdf.

Cybersecurity and Infrastructure Security Agency. 2024a. *Reported Supply Chain Compromise Affecting XZ Utils Data Compression Library, CVE-2024-3094.* https://www.cisa.gov/news-events/alerts/2024/03/29/reported-supply-chain-compromise-affecting-xz-utils-data-compression-library-cve-2024-3094.

Cybersecurity and Infrastructure Security Agency. 2024b. *Secure Software Development Attestation Form.* https://www.cisa.gov/sites/default/files/2024-04/Self_Attestation_Common_Form_FINAL_508c.pdf.

Cybersecurity and Infrastructure Security Agency. 2025. *2025 Minimum Elements for a Software Bill of Materials (SBOM).* Public comment draft. U.S. Department of Homeland Security. https://www.cisa.gov/sites/default/files/2025-08/2025_CISA_SBOM_Minimum_Elements.pdf.

Deng, Peng, Lei Zhang, Yuchuan Meng, Zhemin Yang, Yuan Zhang, and Min Yang. 2025. "ChainFuzz: Exploiting Upstream Vulnerabilities in Open-Source Supply Chains." *34th USENIX Security Symposium (USENIX Security 25)* (Seattle, WA), August. https://www.usenix.org/conference/usenixsecurity25/presentation/deng.

Durumeric, Zakir, James Kasten, David Adrian, et al. 2014. "The Matter of Heartbleed." *Proceedings of the 2014 Conference on Internet Measurement Conference (IMC '14)* (New York), November 5, 475–88. https://doi.org/10.1145/2663716.2663755.

Eghbal, Nadia. 2016. *Roads and Bridges: The Unseen Labor Behind Our Digital Infrastructure.* Ford Foundation. https://www.fordfoundation.org/learning/library/research-reports/roads-and-bridges-the-unseen-labor-behind-our-digital-infrastructure/.

Eghbal, Nadia. 2020. *Working in Public: The Making and Maintenance of Open Source Software.* Stripe Press.

Ellis, Ryan, and Jaikrishna Bollampalli. 2024. *Bug Bounties and FOSS: Opportunities, Risks, and a Path Forward.* Sovereign Tech Fund. https://www.sovereign.tech/public/files/Bug-Bounties-and-FOSS-EN.pdf.

Ellis, Ryan, and Yuan Stevens. 2022. *Bounty Everything: Hackers and the Making of the Global Bug Marketplace*. Data & Society Research Institute. https://datasociety.net/library/bounty-everything-hackers-and-the-making-of-the-global-bug-marketplace/.

emaste. 2024. *Tar: Make Error Reporting More Robust and Use Correct Errno*. Pull request 2101, libarchive/libarchive. GitHub. https://github.com/libarchive/libarchive/pull/2101.

European Parliament and Council of the European Union. 2024. *Regulation (EU) 2024/2847 of the European Parliament and of the Council of 23 October 2024 on Horizontal Cybersecurity Requirements for Products with Digital Elements and Amending Regulations (EU) No 168/2013 and (EU) 2019/1020 and Directive (EU) 2020/1828 (Cyber Resilience Act)*. https://eur-lex.europa.eu/eli/reg/2024/2847/oj.

European Union Agency for Cybersecurity (ENISA). 2021. *ENISA Threat Landscape for Supply Chain Attacks*. European Union Agency for Cybersecurity (ENISA). https://doi.org/10.2824/168593.

Executive Office of the President. 2021. *Executive Order 14028 of May 12, 2021: Improving the Nation's Cybersecurity*. https://www.govinfo.gov/content/pkg/FR-2021-05-17/pdf/2021-10460.pdf.

Feller, Joseph, Brian Fitzgerald, Scott A. Hissam, and Karim R. Lakhani, eds. 2005. *Perspectives on Free and Open Source Software*. MIT Press.

FireEye. 2020. *Highly Evasive Attacker Leverages SolarWinds Supply Chain to Compromise Multiple Global Victims with SUNBURST Backdoor*. https://cloud.google.com/blog/topics/threat-intelligence/evasive-attacker-leverages-solarwinds-supply-chain-compromises-with-sunburst-backdoor.

Fogel, Karl. 2020. *Producing Open Source Software: How to Run a Successful Free Software Project*. 2nd ed. https://producingoss.com/.

Freire, Rodrigo. 2024. *Understanding Red Hat's Response to the XZ Security Incident*. Red Hat. https://www.redhat.com/en/blog/understanding-red-hats-response-xz-security-incident.

Freund, Andres. 2024a. *A Chat about the XZ Backdoor with the Guy Who Found It.* Podcast interview by Patrick Gray and Adam Boileau, episode 743. Risky Business. https://risky.biz/RB743/.

Freund, Andres. 2024b. *Backdoor in Upstream Xz/Liblzma Leading to Ssh Server Compromise.* https://www.openwall.com/lists/oss-security/2024/03/29/4.

Freund, Andres. 2024c. *Discovering the XZ Backdoor with Andres Freund.* Podcast interview by Bryan Cantrill and Adam Leventhal. Oxide and Friends.

Freund, Andres. 2024d. *I Accidentally Found a Security Issue While Benchmarking Postgres Changes.* https://mastodon.social/@AndresFreundTec/112180083704606941.

Freund, Andres, and Thomas Roccia. 2024. *Behind the Scenes of the XZ Vuln with Andres Freund and Thomas Roccia.* Podcast interview by Sherrod DeGrippo, episode 18. Microsoft Threat Intelligence Podcast.

Gates, William H. 1976. "An Open Letter to Hobbyists." *Homebrew Computer Club Newsletter* 2 (1): 2. https://commons.wikimedia.org/wiki/File:Bill_Gates_Letter_to_Hobbyists.jpg.

Ghosh, Rishab Aiyer. 2005. "Understanding Free Software Developers: Findings from the FLOSS Study." In *Perspectives on Free and Open Source Software*, edited by Joseph Feller, Brian Fitzgerald, Scott A. Hissam, and Karim R. Lakhani. MIT Press.

Glyph. 2024. *I Really Hope That This Causes an Industry-Wide Reckoning.* https://mastodon.social/@glyph/112180922900094371.

Goodin, Dan. 2024. *What We Know about the xz Utils Backdoor That Almost Infected the World.* https://arstechnica.com/security/2024/04/what-we-know-about-the-xz-utils-backdoor-that-almost-infected-the-world/.

Greenberg, Andy. 2019. *Sandworm: A New Era of Cyberwar and the Hunt for the Kremlin's Most Dangerous Hackers.* Doubleday.

Greenberg, Andy. 2022. *Tracers in the Dark: The Global Hunt for the Crime Lords of Cryptocurrency.* Doubleday.

Greenberg, Andy, and Matt Burgess. 2024. *The Mystery of "Jia Tan," the XZ Backdoor Mastermind*. WIRED. https://www.wired.com/story/jia-tan-xz-backdoor/.

Haruyama, Takahiro. 2025. *Docker Hub Debian Image Contains CVE-2024-3094 Backdoor*. Issue \#246 in debuerreotype/docker-debian-artifacts. GitHub. https://github.com/debuerreotype/docker-debian-artifacts/issues/246.

Hess, Joey. 2024. "revert to version that does not contain changes by bad actor." Bug report 1068024, package xz-utils. Debian Bug Tracking System, March 29. https://bugs.debian.org/1068024.

Hoffmann, Manuel, Frank Nagle, and Yanuo Zhou. 2024. *The Value of Open Source Software*. Working Paper 24-038. Harvard Business School Strategy Unit. https://doi.org/10.2139/ssrn.4693148.

Huang, Cheng, Nannan Wang, Ziyan Wang, et al. 2024. "DONAPI: Malicious NPM Packages Detector Using Behavior Sequence Knowledge Mapping." *33rd USENIX Security Symposium (USENIX Security 24)* (Philadelphia, PA), August, 3765–82. https://www.usenix.org/conference/usenixsecurity24/presentation/huang-cheng.

James, Sam. 2024a. "=app-arch/xz-utils-5.6.0, =app-arch/xz-utils-5.6.1: backdoor in release tarballs." Bug report 928134, Gentoo Security. Gentoo Bugzilla, March 29. https://bugs.gentoo.org/928134.

James, Sam. 2024b. *FAQ on the Xz-Utils Backdoor (CVE-2024-3094)*. https://gist.github.com/thesamesam/223949d5a074ebc3dce9ee78baad9e27.

Jansen, Hans. 2024. *xz-utils: New upstream version available*. Bug report 1067708, package xz-utils. Debian Bug Tracking System. https://bugs.debian.org/1067708.

JiaT75. 2021. *Added Error Text to Warning When Untaring with bsdtar*. Pull request 1609, libarchive/libarchive. GitHub. https://github.com/libarchive/libarchive/pull/1609.

Kali Linux. 2024. *All about the xz-utils Backdoor*. https://www.kali.org/blog/about-the-xz-backdoor/.

Kalu, Kelechi G., Tanmay Singla, Chinenye Okafor, Santiago Torres-Arias, and James C. Davis. 2025. "An Industry Interview Study of Software Signing for Supply Chain Security." *34th USENIX Security Symposium (USENIX Security 25)* (Seattle, WA), August, 81–100. https://www.usenix.org/conference/usenixsecurity25/presentation/kalu.

Karty, Rhea, and Simon Henniger. 2024. *XZ Backdoor: Times, Damned Times, and Scams*. https://rheaeve.substack.com/p/xz-backdoor-times-damned-times-and.

Kaspersky GReAT. 2024a. *Assessing the Y, and How, of the XZ Utils Incident*. https://securelist.com/xz-backdoor-story-part-2-social-engineering/112476/.

Kaspersky GReAT. 2024b. *XZ Backdoor Story: Initial Analysis*. https://securelist.com/xz-backdoor-story-part-1/112354/.

Keefe, Patrick Radden. 2021. *Empire of Pain: The Secret History of the Sackler Dynasty*. Doubleday.

Kelty, Christopher M. 2005. "Free Science." In *Perspectives on Free and Open Source Software*, edited by Joseph Feller, Brian Fitzgerald, Scott A. Hissam, and Karim R. Lakhani. MIT Press.

Kelty, Christopher M. 2008. *Two Bits: The Cultural Significance of Free Software*. Duke University Press.

Knuth, Donald E. 1974. "Computer Programming as an Art." *Communications of the ACM* 17 (12): 667–73. https://doi.org/10.1145/361604.361612.

Krebs, Brian. 2024. *Some Thoughts about Attribution in the XZ Backdoor*. https://infosec.exchange/@briankrebs/112197305365490518.

Lakhani, Karim R., and Robert G. Wolf. 2005. "Why Hackers Do What They Do: Understanding Motivation and Effort in Free/Open Source Software Projects." In *Perspectives on Free and Open Source Software*, edited by Joseph Feller, Brian Fitzgerald, Scott A. Hissam, and Karim R. Lakhani. MIT Press.

Leite, Anderson. 2024. *XZ Backdoor: Hook Analysis*. https://securelist.com/xz-backdoor-part-3-hooking-ssh/113007/.

Lerner, Josh, and Jean Tirole. 2005. "Economic Perspectives on Open Source." In *Perspectives on Free and Open Source Software*, edited by Joseph Feller, Brian Fitzgerald, Scott A. Hissam, and Karim R. Lakhani. MIT Press.

Lessig, Lawrence. 2006. *Code: And Other Laws of Cyberspace, Version 2.0*. Basic Books.

Linux Foundation. 2020. *2020 Linux Kernel History Report*. Linux Foundation. https://www.linuxfoundation.org/resources/publications/linux-kernel-history-report-2020.

Linux Foundation. 2026. https://www.linuxfoundation.org/press/linux-foundation-announces-12.5-million-in-grant-funding-from-leading-organizations-to-advance-open-source-security.

Linux Foundation. n.d. "The Linux Kernel Organization: Contributor Insights." https://insights.linuxfoundation.org/project/korg/contributors.

Maffulli, Stefano. 2023. *Meta's LLaMa License Is Not Open Source*. Open Source Initiative. https://opensource.org/blog/metas-llama-2-license-is-not-open-source.

Mandiant. 2022. *Assembling the Russian Nesting Doll: UNC2452 Merged into APT29*. https://cloud.google.com/blog/topics/threat-intelligence/unc2452-merged-into-apt29/.

Mastery Learning. 2024. *Linus Torvalds: XZ Utils Breach Raises Questions about Trust in Open Source Development*. https://www.youtube.com/watch?v=vtce4vmZIC8.

Mazurov, Nikita. 2024. *The Other Players Who Helped (Almost) Make the World's Biggest Backdoor Hack*. The Intercept. https://theintercept.com/2024/04/03/linux-hack-xz-utils-backdoor/.

Meiklejohn, Sarah, Marjori Pomarole, Grant Jordan, et al. 2013. "A Fistful of Bitcoins: Characterizing Payments Among Men with No Names." *Proceedings of the 2013 Conference on Internet Measurement Conference (IMC '13)* (New York), October 23, 127–39. https://doi.org/10.1145/2504730.2504747.

Meissner, Marcus. 2024. *openSUSE Addresses Supply Chain Attack Against xz Compression Library*. openSUSE. https://news.opensuse.org/2024/03/29/xz-backdoor/.

Meta. 2023a. *Llama 2 Acceptable Use Policy*. https://ai.meta.com/llama/use-policy.

Meta. 2023b. *Llama 2 Community License Agreement*. https://github.com/meta-llama/llama-models/blob/main/models/llama2/LICENSE.

Meta. 2024a. *Llama 3.1 Acceptable Use Policy*. https://www.llama.com/llama3_1/use-policy.

Meta. 2024b. *Llama 3.1 Community License Agreement*. https://github.com/meta-llama/llama-models/blob/main/models/llama3_1/LICENSE.

Moody, Glyn. 2001. *Rebel Code: Linux and the Open Source Revolution*. Perseus.

Munroe, Randall. 2020. "Dependency." August 17. https://xkcd.com/2347/.

Nagle, Frank, David A. Wheeler, Hila Lifshitz-Assaf, Haylee Ham, and Jennifer L. Hoffman. 2020. *Report on the 2020 FOSS Contributor Survey*. The Linux Foundation; Laboratory for Innovation Science at Harvard. https://www.linuxfoundation.org/wp-content/uploads/2020FOSSContributorSurveyReport_121020.pdf.

Nagle, Frank, Jessica Wilkerson, James Dana, and Jennifer L. Hoffman. 2020. *Vulnerabilities in the Core: Preliminary Report and Census II of Open Source Software*. The Linux Foundation; The Laboratory for Innovation Science at Harvard.

National Institute of Standards and Technology. 2022. *Secure Software Development Framework (SSDF) Version 1.1: Recommendations for Mitigating the Risk of Software Vulnerabilities (NIST SP 800-218)*. U.S. Department of Commerce. https://doi.org/10.6028/NIST.SP.800-218.

National Telecommunications and Information Administration. 2021. *The Minimum Elements for a Software Bill of Materials (SBOM)*. U.S. Department of Commerce. https://www.ntia.gov/report/2021/minimum-elements-software-bill-materials-sbom.

O'Mahony, Siobhán. 2005. "Nonprofit Foundations and Their Role in Community-Firm Software Collaboration." In *Perspectives on Free and Open Source Software*, edited by Joseph Feller, Brian Fitzgerald, Scott A. Hissam, and Karim R. Lakhani. MIT Press.

Office of Management and Budget. 2022. *Enhancing the Security of the Software Supply Chain Through Secure Software Development Practices*. https://bidenwhitehouse.archives.gov/wp-content/uploads/2022/09/M-22-18.pdf.

Office of Management and Budget. 2026. *Adopting a Risk-Based Approach to Software and Hardware Security*. https://www.whitehouse.gov/wp-content/uploads/2026/01/M-26-05-Adopting-a-Risk-based-Approach-to-Software-and-Hardware-Security.pdf.

Ohm, Marc, Henrik Plate, Arnold Sykosch, and Michael Meier. 2020. "Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks." *arXiv Preprint arXiv:2005.09535*, May 19. https://arxiv.org/abs/2005.09535.

Open Source Initiative. 2007. *The Open Source Definition*. https://opensource.org/osd.

Open Source Initiative. 2024. *The Open Source AI Definition, Version 1.0*. https://opensource.org/ai/open-source-ai-definition.

OpenSSH. 2021. "PROTOCOL.certkeys." June 5. https://github.com/openssh/openssh-portable/blob/281ea25a44bff53eefb4af7bab7aa670b1f8b6b2/PROTOCOL.certkeys.

OpenSSH. 2024. *OpenSSH 9.8/9.8p1 Release Notes*. https://www.openssh.com/txt/release-9.8.

OpenSSL. n.d. "RSA_private_encrypt, RSA_public_decrypt." Accessed June 1, 2026. https://docs.openssl.org/master/man3/RSA_private_encrypt/.

Osborne, Cailean, Paul Sharratt, Dawn Foster, and Mirko Boehm. 2024. "A Toolkit for Measuring the Impacts of Public Funding on Open Source Software Development." *arXiv Preprint arXiv:2411.06027*, November 9. https://arxiv.org/abs/2411.06027.

Ostrom, Elinor. 1990. *Governing the Commons: The Evolution of Institutions for Collective Action*. Cambridge University Press.

Perlroth, Nicole. 2021. *This Is How They Tell Me the World Ends: The Cyberweapons Arms Race*. Bloomsbury.

Ponta, Serena Elisa, Henrik Plate, and Antonino Sabetta. 2020. "Detection, Assessment and Mitigation of Vulnerabilities in Open Source Dependencies." *Empirical Software Engineering* 25 (5): 3175–215. https://doi.org/10.1007/s10664-020-09830-x.

PostgreSQL Core Team. 2020. *New PostgreSQL Core Team Members*. https://www.postgresql.org/about/news/new-postgresql-core-team-members-2103/.

Przymus, Piotr, and Thomas Durieux. 2025. "Wolves in the Repository: A Software Engineering Analysis of the XZ Utils Supply Chain Attack." *22nd IEEE/ACM International Conference on Mining Software Repositories, MSR@ICSE 2025, Ottawa, ON, Canada, April 28-29, 2025*, April 28, 91–102. https://doi.org/10.1109/MSR66628.2025.00026.

Raymond, Eric S. 2001. *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. Revised. O'Reilly.

Red Hat. 2024. *Urgent Security Alert for Fedora Linux 40 and Fedora Rawhide Users*. https://www.redhat.com/en/blog/urgent-security-alert-fedora-40-and-rawhide-users.

Rid, Thomas. 2020. *Active Measures: The Secret History of Disinformation and Political Warfare*. Farrar, Straus; Giroux.

Roccia, Thomas. 2024. *The XZ Backdoor Story: The Undercover Operation That Set the Internet on Fire*. Conference talk. DEF CON 32. https://www.youtube.com/watch?v=hwuIb-Vv2Ew.

Roncone, Gabby, Dan Black, John Wolfram, et al. 2024. *APT44: Unearthing Sandworm*. Mandiant / Google Cloud Threat Intelligence. https://services.google.com/fh/files/misc/apt44-unearthing-sandworm.pdf.

Ryan, Keegan. 2024. *Modify_ssh_rsa_pubkey.py*. https://gist.github.com/keeganryan/a6c22e1045e67c17e88a606dfdf95ae4/cc803f25c6e0260a953f63ca2acb72104c395b04.

Schorlemmer, Taylor R., Kelechi G. Kalu, Luke Chigges, et al. 2024. "Signing in Four Public Software Package Registries: Quantity, Quality, and Influencing Factors." *2024 IEEE Symposium on Security and Privacy (SP)*, May, 1160–78. https://doi.org/10.1109/SP54263.2024.00215.

Sovereign Tech Fund. n.d. *Project Documentation.* Accessed May 31, 2026. https://sovereigntechfund.de.

Spracklen, Joseph, Raveen Wijewickrama, A H M Nazmus Sakib, Anindya Maiti, Bimal Viswanath, and Murtuza Jadliwala. 2025. "We Have a Package for You! A Comprehensive Analysis of Package Hallucinations by Code Generating LLMs." *34th USENIX Security Symposium (USENIX Security 25)* (Seattle, WA), August, 3687–706. https://www.usenix.org/conference/usenixsecurity25/presentation/spracklen.

Star, Susan Leigh. 1999. "The Ethnography of Infrastructure." *American Behavioral Scientist* 43 (3): 377–91. https://doi.org/10.1177/00027649921955326.

Stoll, Clifford. 1989. *The Cuckoo's Egg: Tracking a Spy Through the Maze of Computer Espionage.* Doubleday.

Tidelift. 2023. *The 2023 Tidelift State of the Open Source Maintainer Report.* Tidelift. https://tidelift.com/open-source-maintainer-survey-2023.

Torvalds, Linus. 1991. *What Would You Like to See Most in Minix?* https://groups.google.com/g/comp.os.minix/c/dlNtH7RRrGA.

Torvalds, Linus, and David Diamond. 2001. *Just for Fun: The Story of an Accidental Revolutionary.* HarperBusiness.

Tukaani Project. 2010. *XZ Utils AUTHORS.* Source file at tag v5.0.0. https://github.com/tukaani-project/xz/blob/v5.0.0/AUTHORS.

Tukaani Project. n.d.-a. "LZMA Utils." Accessed June 7, 2026. https://tukaani.org/lzma/.

Tukaani Project. n.d.-b. *LZMA Utils AUTHORS.* Accessed June 7, 2026. https://git.tukaani.org/?p=lzma.git;a=blob_plain;f=AUTHORS;hb=HEAD.

Tukaani Project. n.d.-c. *LZMA Utils README.* Accessed June 7, 2026. https://git.tukaani.org/?p=lzma.git;a=blob_plain;f=README;hb=HEAD.

Tukaani Project. n.d.-d. "XZ Utils." Accessed June 7, 2026. https://tukaani.org/xz/.

Valsorda, Filippo. 2024. *Preliminary Analysis of the XZ Utils Backdoor Payload.* https://bsky.app/profile/filippo.abyssdomain.expert/post/3kowjkx2njy2b.

Vinsel, Lee, and Andrew L. Russell. 2020. *The Innovation Delusion: How Our Obsession with the New Has Disrupted the Work That Matters Most.* Currency.

Weber, Steven. 2004. *The Success of Open Source.* Harvard University Press.

Weems, Anthony. 2024. *Xzbot: Notes, Honeypot, and Exploit Demo for the Xz Backdoor (CVE-2024-3094).* https://github.com/amlweems/xzbot.

Wikipedia contributors. n.d. *XZ Utils Backdoor.* Accessed May 31, 2026. https://en.wikipedia.org/wiki/XZ_Utils_backdoor.

Williams, Sam. 2010. *Free as in Freedom (2.0): Richard Stallman and the Free Software Revolution.* 2nd ed. Free Software Foundation. https://static.fsf.org/nosvn/faif-2.0.pdf.

xz-devel mailing list. 2022. *Pressure-Campaign Correspondence on the Xz-Devel Mailing List.* https://www.mail-archive.com/xz-devel@tukaani.org/.

Zalewski, Michał. 2024. *Techies Vs Spies: The xz Backdoor Debate.* https://lcamtuf.substack.com/p/technologist-vs-spy-the-xz-backdoor.

Zemczak, Łukasz. 2024. *Noble Numbat Beta Delayed (xz/liblzma Security Update).* Ubuntu Community Hub. https://discourse.ubuntu.com/t/noble-numbat-beta-delayed-xz-liblzma-security-update/43827.

Zetter, Kim. 2014. *Countdown to Zero Day: Stuxnet and the Launch of the World's First Digital Weapon.* Crown.

Zittrain, Jonathan L. 2008. *The Future of the Internet — and How to Stop It.* Yale University Press.

Index

About the Author

Adrian Mastronardi has spent more than two decades building and running the engineering organizations behind software used by millions of people across five continents. He began at Globant, the Latin American software company, and went on to lead technology at the online classifieds platform OLX as it grew across Brazil, India, and dozens of other emerging markets in Africa, Asia, and Latin America. From there he moved into financial technology: RappiPay, the fintech arm of the delivery company Rappi, where, as CTO, he helped launch a credit card and digital wallet across several markets and pursue a banking license in Colombia; a fintech venture at Delivery Hero in Berlin, whose technical foundation he built; and Habi, a real-estate technology company operating in Colombia and Mexico, where he has been CTO since 2023. The constant across that work is infrastructure: the platforms, payment rails, and data systems a business runs on, almost all of it standing on open-source foundations he has built on since 1996, when he installed his first Linux system.

That long involvement with infrastructure, Linux, and open source is what drew him to the XZ Utils story, a backdoor planted in exactly the kind of open-source code his work depends on. *Half a Second* is his third book, after *Maestros involuntarios*, a reflection on entrepreneurship drawn from two decades alongside some of Latin America's most successful founders, and *Willpower Amplified*, on how AI is making temperament a new inequality. Both are freely available on his website. He writes *Al sur del Río Grande*, a weekly newsletter in Spanish on artificial intelligence, software, and the craft of building it, published on

LinkedIn and openly on his website.

`adrian@mastronardi.xyz` | `www.mastronardi.xyz`

Colophon

This book is set in Libertinus, a humanist old-style typeface that Khaled Hosny forked in 2012 from the Linux Libertine project Philipp H. Poll began in 2003. Hosny maintained Libertinus until 2020, when he handed it to Caleb Maclennan, its current maintainer. The body text and display are set in Libertinus Serif, with code, commands, and version numbers in Libertinus Mono; the title page sets the author's name in small capitals. Both faces are released under the SIL Open Font License.

The book itself was written in Markdown and typeset with pandoc and XeLaTeX on Fedora Linux. Fonts, tools, and the operating system beneath them are all free and open-source. For a book about the labor behind open-source software, no other choice felt right.