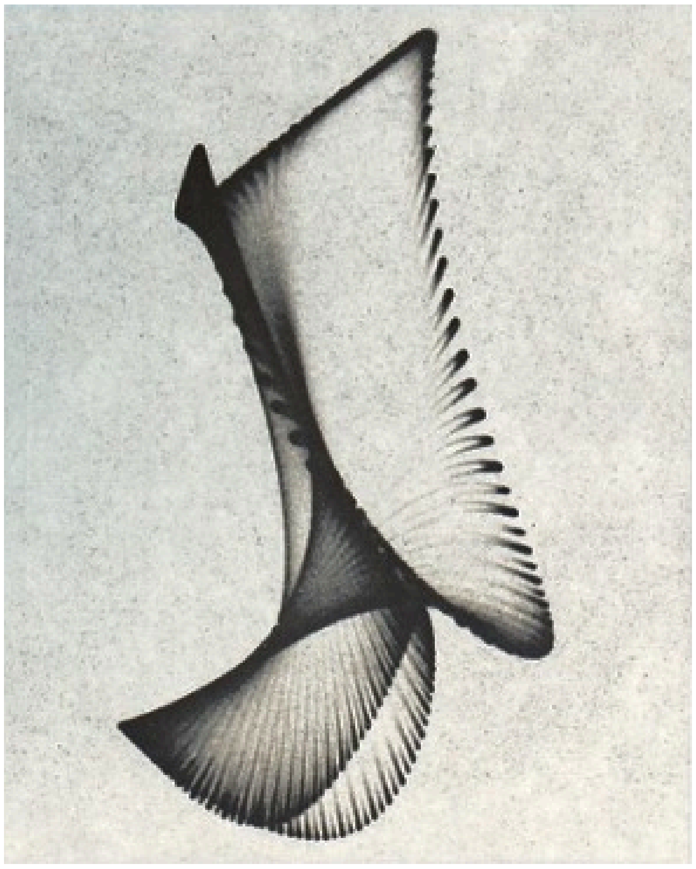
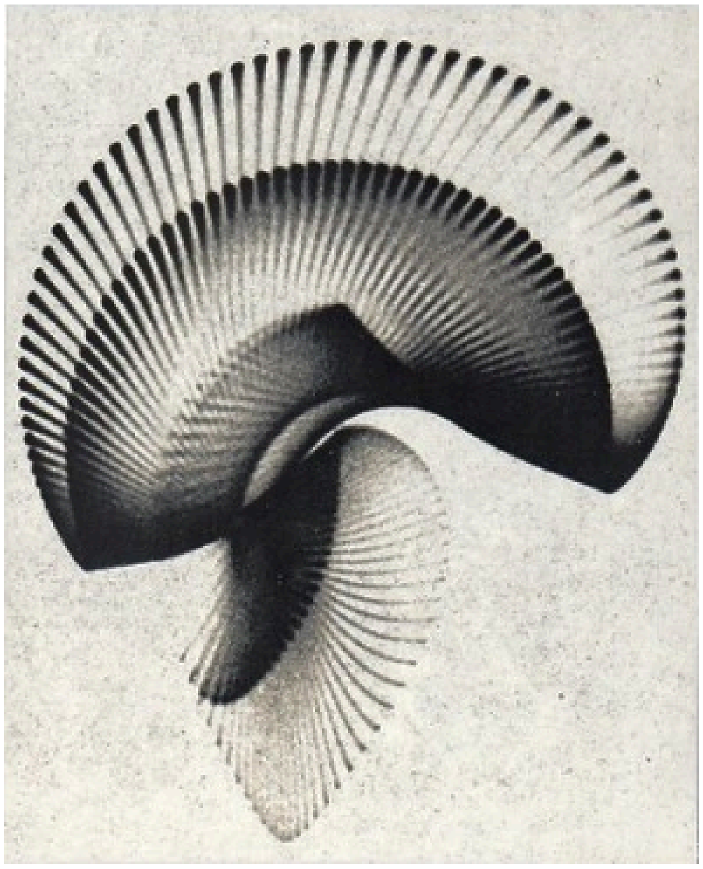
OpenSSH Backdoors


Oscillon # 16


Oscillon # 38

Ben Hawkes

Aug 23, 2024

Imagine this: an OpenSSH backdoor is discovered, maintainers rush to push out a fixed release package, security researchers trade technical details on mailing lists to analyze the backdoor code. Speculation abounds on the attribution and motives of the attacker, and the tech media pounces on the story. A near miss of epic proportions, a blow to the fabric of trust underlying open source development, a stark reminder of the risks of supply-chain attacks. Equal measures brilliant and devious.

If you've been paying attention to the security news recently, your mind probably went straight to the attack on the liblzma/xz-utils repository earlier this year, the ultimate aim of which was an OpenSSH backdoor. However the event described above isn't the xz-utils backdoor. It's a little-remembered fact that the xz-utils backdoor was actually the *second* time OpenSSH had a "near miss" with a backdoor attack. The first time was over 22 years ago, all the way back in 2002. This blog post shares the story of that backdoor, and what we can learn from an attack that happened over two decades ago.

Background

The 2002 attack was quite straightforward. Here's the original announcement: OpenSSH Security Advisory: Trojaned Distribution Files.

The OpenSSH source code was hosted on ftp.openbsd.org, and somehow it was replaced with a backdoored version. Nobody knows exactly how this happened, but the attacker managed to switch out the .tar.gz files for a few versions. There were some really good server exploits going around the hacking scene at that time though, so there's nothing particularly surprising about this. Fortunately though, the backdoor didn't last long because a developer noticed an unexpected difference in checksums. Specifically when trying to build the backdoor on FreeBSD, the "ports" package system ran the checksums automatically, and since ports already had a checksum stored when the backdoor was inserted, a mismatch occurred. If the attackers had waited until a new version was released and immediately replaced the .tar.gz and the checksum files at the same time, they would have had a lot more success.

In any case, it was a simple backdoor, essentially the simplest that you could imagine. Step one, hook the build system so that an attacker-controlled source file would be compiled and executed when the "configure" command was run by the victim. Step two, have the payload connect back to a hardcoded IP address in Australia and wait for commands to execute on the compromised machine.

We still don't know for sure who was behind the backdoor, but the prevailing opinion (at least among the OpenBSD developers I talked to) seems to be that it was just some old fashioned shenanigans. Punk kids doing punk kid things, as was perfectly normal in 2002, and then life goes on. It certainly wasn't the first time something like this had happened -- Wu-FTPd, which was the most popular ftpd in the 90s, had a very similar thing happen back in 1993 -- but it was a good indication of what was to come. It's an interesting historical event, because it shares both some similarities and differences with the modern xz-utils OpenSSH backdoor attempt, and by prying those apart we can hopefully find some useful insights.

Similarities

The obvious similarity is that both the historical and modern events targeted OpenSSH. That's for good reason. OpenSSH is clearly in the upper echelon of targets for vulnerability research because if you get a pre-authentication remote exploit working for OpenSSH you essentially have a skeleton key for the Internet and everything else comes tumbling down from that initial access. It turns out that there aren't many remote OpenSSH exploits left though, it's been about 20 years since the last really good one. So in lieu of finding a good bug, the next best thing is to insert one.

Inserting an exploitable bug (a "bugdoor"), one that's subtle enough that developers might not even notice during code review, is probably the winning move. However, it's interesting to note that in both 2002 and 2024 we got a backdoor rather than a bugdoor. That's probably because exploits are hard, and server-side exploits are really hard. Given how much work it is to be in a position to change the source code in the first place, it's not entirely surprising that attackers want to go with a reliable option. The counter-argument is that we may just never get to see any bugdoors because they never get caught (or if they do, they don't get flagged as subterfuge), so we're biased towards the events that we can actually detect.

There are other similarities. Both the 2002 and 2024 events targeted the build system, for example. This also makes sense, because build systems are a perfect mix of inscrutability and expressiveness. There's really no constraints on what you can do with most build systems. They have to be like this in order to make everything work everywhere that it's supposed to. Making something compile on Linux, MacOS, and Windows simultaneously is no easy feat. Add in support for multiple architectures and legacy versions, and well... you see where I'm going with this. The guiding design principle for build systems has been "just make it work", and so they end up being a complicated mess of directives, rules, variables, and command invocations. As long as they're working correctly, I suspect very few people are paying close attention to the contents of their build scripts, and that includes the developers/maintainers themselves. It's the ideal place to insert the first hook for a backdoor, hiding in plain sight.

The same attributes that make OpenSSH a very attractive target also make it a very difficult target, however. Everyone uses it, and so the chances that someone notices that something fishy is going on is quite high. Indeed, both attacks were found relatively soon after they were attempted. The 2002 attack was found by a developer noticing that the checksums provided didn't match the source code they had downloaded, and the 2024 attack was found by a developer after diligent exploration of a performance issue. The "many eyes" theory of open source security isn't popular right now, but it certainly seems like bigger targets have smaller margins for error.

The last similarity is that both events were perpetrated by unknown attackers, e.g. they were caught in the act, but never attributed to any specific threat actor or country. This might not seem like much, but I suspect the key observation is that our usual approaches to attribution don't work very well for supply chain attacks. The sample size is tiny, the attacker's targeting is opaque, and each event sees a high degree of customization. For an attacker this is quite appealing, since either the attack succeeds or it fails in such a way that nobody can figure out exactly who was responsible.

Differences

Despite the similarities, these two attacks are fundamentally on a different level in terms of their intent and execution. It's interesting to see how things have evolved. If you want an attack like this to be successful, everything has to go perfectly. Clearly then the xz-utils backdoor wasn't perfect, but it did do a lot of things right, and came a lot closer to succeeding than the 2002 attack. The main difference that explains this seems to be in the motivation and intent of the perpetrators.

The consensus is that the attackers in 2002 were motivated by having fun and causing mayhem, and they probably didn't mind getting caught all that much. If you're thinking in terms of bragging rights, getting caught might actually be a feature rather than a bug. In contrast, the attackers in 2024 seem to have been given a very specific task, and clearly intended to actually use the backdoor to achieve their goals. In other words, the xz-utils backdoor was designed to be an intelligence capability, whereas the 2002 attack was more "performance art" than "persistent threat".

One of the key technical differences is that the xz-utils backdoor was targeting the build *artifact* rather than the build *system*. The worst-case outcome of the 2002 attack was compromising whichever systems happened to compile OpenSSH. If the xz backdoor had been successful however, then eventually every single machine running OpenSSH on a systemd-based Linux distribution could have been compromised at any time or place of the attacker's choosing.

That word "choice" is important here. The xz-utils backdoor gave attackers optionality: they could choose to deploy the backdoor's hidden features in a targeted manner. When compared to triggering a reverse shell automatically, this greatly reduces the risk that the backdoor is detected, and allows the backdoor to be used in a surgical manner. This is a conscious decision that the attackers made, because the alternative of compromising *every* system that their code was run on was also certainly an option here.

It's also interesting how indirect the 2024 attack was. Instead of targeting OpenSSH itself, they noticed that modern Linux distributions inserted an unexpected dependency on liblzma into OpenSSH. This was the path to victory – instead of targeting a mature and well-funded project that's maintained by world renowned security experts, go for the underfunded and understaffed utility library that no one even realized was in the critical path. Defenders think in lists, attackers think in graphs.

Aside from this, another small innovation stood out to me. Rather than inserting obfuscated shell scripts, hiding in a C file (like the 2002 attack did), or fetching a payload over the network, the xz backdoor's payload was pre-staged in a binary-only test file. I think this was demonstrated to be an effective approach given that nobody noticed the payload in the xz-utils source code repository until after the backdoor had been detected at runtime using performance analysis. If there hadn't been a performance regression, and if the attacker's had been slightly less aggressive in their social maneuvering, I suspect both the "hook" and the payload may have gone undetected for a long time.

The final big difference is in the attacker's methodology. In the 2002 attack, we saw the attackers go straight for the infrastructure that was hosting OpenSSH. In contrast, the xz backdoor was the culmination of an extended social engineering campaign that led to the attacker becoming a trusted part of the core development team. Any way you look at it, that's an impressive effort.

Analysis

There's a lot to unpack here. Supply chain attacks have certainly evolved, but... not by as much as expected? For a moment, let's put aside the "malicious insider" approach used by the xz-utils attackers and focus on the "attack the infrastructure" approach instead. If you look at the details of the 2002 attack, at a very fundamental level there's nothing that would stop this attack from succeeding today. With a little bit more finesse and patience, an infrastructure-focused attack that aims to subvert a source code distribution is still certainly plausible.

My favorite example of this is zlib. Just like xz-utils, it's a compression library. Arguably it's *the* compression library, because zlib is absolutely everywhere – including in OpenSSH. New versions of their source code are distributed from zlib.net, and the server running zlib.net is hosted by a small company in Michigan called a2hosting.com where a managed VPS starts at $26.95/month. This hosting company is particularly fond of using CPanel and exim, both of which are enabled for zlib.net.

That means the supply chain integrity for *practically everything* relies on the integrity of a2hosting.com and the absence of any remote exploits in CPanel or exim. The track record here isn't exactly encouraging, and I haven't even got to Pure-FTPD, Apache httpd, or Dovecot (and this is just the stuff that's directly on zlib.net, we're not even considering how a2hosting.com itself might be attacked). Find a good vulnerability in any one of these projects, or a way to backdoor them for that matter, and you have a good shot at backdooring the zlib source code distribution.

Things have improved for zlib in recent years, because at least they have a dedicated host or VPS now. For a long time zlib.net was backed by PHP shared hosting (e.g. the site was shared with many other websites and anyone could pay to be resident on the same machine). This was a bit of a running joke among vulnerability researchers: since finding a real bug in zlib is extraordinarily difficult, inserting one in the code base when the maintainer announced a new release was probably the path of least resistance.

The point of this isn't to pick on zlib. Their maintainer is world class (I had the privilege of reading their Huffman table code during this analysis), and they're doing a great job overall. The problem here isn't unique to zlib, or xz-utils, or OpenSSH. Everyone is exposed in fairly similar ways. The point is that when you look at the cumulative risk, we're in really bad shape. I'm not usually prone to exaggeration, but our current exposure to supply chain attacks is somewhat alarmingly high.

Let's think about this. If you compile OpenSSH from source, you end up with code (libraries and executables) from about 5 different distro packages running in your address space. Not too bad. In practice though, everyone runs a systemd-based Linux distribution of some sort – in which case you end up running code from around 30 different packages in your OpenSSH address space (including our friends xz and zlib of course). That's already starting to get uncomfortable.

But it doesn't stop there. While both the historical attack and the xz-utils attackers tried to go straight for OpenSSH, there's really no need. If you can get your backdoor running as root *somewhere* on the system, then you can inject yourself into the sshd process. It's one extra step, but not a very difficult one. On a stock Ubuntu Server 22.04 system after boot, there is code from 97 packages running as root (that's 16% of all the packages installed by default). On my daily driver Linux desktop (where I use remote access via OpenSSH almost every day), there's code from a remarkable 384 packages running as root.

Is this really a defensible security boundary? Hundreds of projects with distinct cultures, motivations, funding, expertise, and resources? Whether it's an infrastructure-focused attack like 2002, or a social engineering attack like 2024, it doesn't really matter. Throw in the angle of targeting and compromising individual developers for that matter. Regardless, I'm not convinced we can defend against this with the way we're currently thinking about operating system design.

Final Thoughts

The supply chain attacks from 20 years ago still look like they're viable today, and we're further behind in our defensive posture than we'd all like. Truthfully we've mostly gotten away with it up until now because there's been a steady supply of exploitable vulnerabilities that have enabled the attacker's to achieve their goals in other ways. However in a world where exploitable vulnerabilities become sparse (and there are some initial signs that this is happening), it's not unreasonable to think that attackers will pursue supply chain attacks at a much higher level. If that's the case, we're not prepared for it yet.

The answer will inevitably involve attack surface reduction and compartmentalization. That means making a conscious effort to reduce the amount of code we have running in remotely exposed processes or at high privilege levels like root. This means accelerating our deployment of sandboxing. We used to think of sandboxing as only applicable to the parts of the codebase that handled untrusted data – image parsers, video decoders, JavaScript engines, and so on. In a world where it's the code rather than the data that's untrusted, the goal should be to reorient toward system designs where all code is constrained to least privilege, and where there are technical controls in place that enforce that.

Fortunately there's some positive steps in this direction, at least for Linux. On Ubuntu 24.04 you can no longer find liblzma in the OpenSSH address space, on Android almost every process is constrained by a mix of SELinux and seccomp-bpf, and on recent Linux kernels we now have support for a promising technology called landlock that will allow even unprivileged apps to run in a sandbox. It takes about 250 lines of code to write a landlock sandbox for "make" that would prevent the 2002 attack.

With the xz-utils backdoor we learnt that there is an extraordinary willingness to invest time, money, and other resources into supply chain attacks. This feels *different* now. The stakes have changed. There's lots of work to do, and it's going to be a long road to get to where we really need to be. I suspect that there may need to be some fairly radical changes around how we think about operating system design and application development along the way.

The good news is that there seems to be a lot of interest in matching the attacker's enthusiasm on the defensive side of supply-chain security. It may not seem like much, but interest and enthusiasm is a great start – and it's more than we had 20 years ago.

← Previous