NEWS · AUGUST 11, 2026

My Server Was Compromised — and Recovery Took One Command

An attacker got code execution on my server and ran a cryptominer for hours — and recovery came down to throwing away one container and rebuilding it, because the architecture decided the blast radius long before the breach and the backups had already been tested.

By Dr. Holger Flick

The site was down, so I did what anyone does: I opened a terminal and started looking. What I found was a process called `kmod-cache-update`, which sounds like something the kernel would run and is in fact XMRig, an open-source Monero miner, wearing a false name. It was pinned to three threads, it was talking to a mining pool, and five separate watchdog scripts stood ready to restart it every sixty seconds if anything killed it.

Someone had code execution on my server. Let me be precise about what that did and did not mean, because the gap between those two things is the whole point of this post.

It meant a miner burned CPU I had not authorized, drove the machine into memory exhaustion, and took my website down for three to four hours. It did **not** mean they reached the host. It did not mean they reached the database, or the mail server, or any of the other services on that machine. Everything the attacker could touch was inside a single container, and when I threw that container away, the problem left with it.

That boundary was not luck and it was not quick thinking on the day. It was drawn months earlier, on a quiet afternoon when nothing was on fire. And the reason I could work the problem calmly instead of frantically comes down to something even less glamorous: I had backups, and I had actually restored from them before.

The door I left open

The way in was my Gitea instance — Gitea being a lightweight, self-hosted Git server, the kind of thing you run when you want your own private GitHub. It was published properly, at a real hostname behind Caddy with a valid certificate, which is exactly how a Git server should be exposed.

The mistake was smaller and more specific than that. **Registration was open, and a fresh account could create Git hooks.**

Gitea's `DISABLE_GIT_HOOKS` setting defaults to `false`, which means users may attach custom scripts that the server runs when a repository event fires. That is a genuine feature — it is how you wire up deployments — but combined with a registration form that anyone can fill in, the sequence writes itself: sign up, create a repository, add a hook, push, and the hook executes on my machine. Four steps, no exploit, no stolen password. Every one of them a feature behaving exactly as designed.

The two settings that close it:

```
; app.ini
[service]
DISABLE_REGISTRATION = true

[security]
DISABLE_GIT_HOOKS = true
```

There is a broader lesson here that I would give any customer, and it is not really about Gitea. **An account should not come into existence, and certainly should not become active, without something verifying that a human is behind it.** Gitea will hand out a working account from a form submission with nothing in between — no email confirmation, no approval, no second factor at any point. On a system whose entire purpose is executing code, an unverified account is a code execution grant, and it should be treated with the same seriousness you would treat handing someone a shell.

So my rule, and I will defend this one: if an application can run code on your infrastructure, either close self-service registration entirely and create accounts yourself, or gate activation behind real verification. And every account that does exist gets a second factor — TOTP or a WebAuthn passkey, not SMS, which NIST has restricted for years because phone numbers can be redirected through SIM swaps.

> **Two controls, two different jobs**
> To be honest about what would have helped here: two-factor authentication on its own would *not* have stopped this attack, because the attacker never touched an existing account — they made their own. Verification at registration is what closes that door; 2FA protects the accounts that already exist from being stolen. They solve different problems, and you want both. Confusing them is how people end up with a well-defended login page in front of a wide-open signup form.

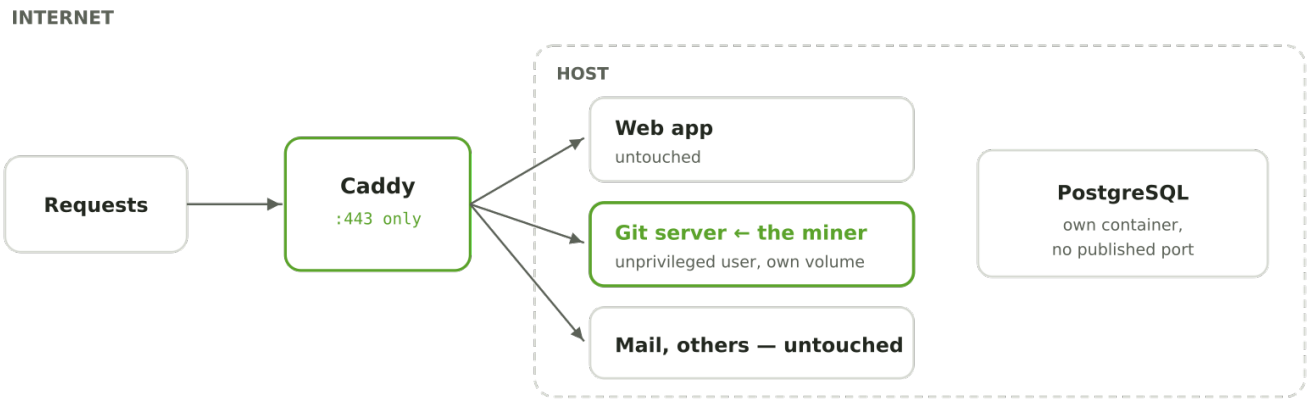
> **How I know this was not personal**
> The miner's configuration tagged its worker with my container's ID — attackers use that field to tell workers apart across thousands of victims, which means I was not selected, I was enumerated. Security firm Wiz documents this exact campaign as JINX-0132: automated scanning for internet-exposed DevOps

infrastructure, followed by XMRig pulled straight from its public GitHub releases. A scanner found a registration form and used it, the same way it does to everyone else it finds.

The sandbox is the story

Now the part that actually matters, because it is the difference between a bad afternoon and a bad year.

Every service on that machine runs in its own Docker container, and nothing is reachable from outside except through Caddy, a reverse proxy that terminates TLS and forwards requests inward. The applications never need a public face of their own.



One public entrance, every service in its own box — the miner landed in one of them and stayed there

Read that diagram as a set of walls, and then read the incident against it. The attacker's code ran in the highlighted box. It ran as that container's unprivileged user, not as root and not as me. It wrote to that container's own data volume. It appeared in that container's own control group. The web application in the box above it kept serving pages until memory pressure took the whole machine down. The database, in its own container with no published port at all, was never reachable from where the attacker stood.

This is what people mean by *blast radius*, and the important property is that **you decide it in advance.** Not during the incident, when you are tired and guessing — months earlier, when you are choosing whether the database gets a published port and whether each service gets its own container. Those decisions cost nothing at the time. They pay out entirely on one afternoon you did not schedule.

The payoff is also what made cleanup trivial. I did not have to disinfect anything or hunt for what else the attacker had touched across the filesystem. The blast radius was a container, so the remedy was a container: stop it, throw the state away, rebuild from a definition I trust. That last part matters — my compose files, Caddy configuration and deployment scripts all live in Git, which means "rebuild from something the attacker never had access to" is a command, not a project.

> **None of this requires my exact stack**
> Podman gives you the same isolation with rootless containers by default, which is stronger still. Traefik and nginx do the reverse-proxy job just as well, and Forgejo — a community-governed fork of Gitea — is an excellent Git server with the same settings worth checking. Managed hosting handles much of the isolation for you, and the residual question becomes who is allowed to register on the things you expose. The principle travels: one public entrance, least privilege inside, and a boundary you chose on purpose.

The reason I never got nervous

I want to describe something that is not technical, because it is the part I would most want a client to understand about how I work.

When I found that miner, I felt annoyance. I did not feel fear. That difference was not composure and it was not experience — it was purchased months earlier by a backup regime I had already tested. Because a restore was available and known to work, every option stayed open to me:

- I could stop the service immediately, without weighing what taking it offline might cost.
- I could preserve the evidence before cleaning, instead of destroying the forensics in a rush to make the problem disappear.
- I could refuse to move a production database across a link from a host that had an attacker on it, and simply wait until that was safe.
- I could choose to rebuild rather than repair, because I knew what I would be rebuilding from.

That is what backups actually buy, and it is not files. It is **decisions**. An administrator without a restore path ends up negotiating with the attacker inside their own head — *maybe if I clean this up quickly, maybe if I do not look too closely, maybe it is fine* — and every one of those pressures points toward the worst available choice.

An administrator with a tested restore feels none of that, and makes better decisions as a direct result.

The discipline behind it is the boring, well-known one:

1. **Three copies of the data, on two kinds of media, one off-site.** The 3-2-1 rule has survived this long because it keeps surviving.
2. **At least one copy the running server cannot reach.** This is the part that specifically defeats ransomware. If your server holds valid write credentials to the backup target, then whoever holds your server holds your backups too. Append-only repositories or pull-based backups solve it.
3. **A restore you have actually performed.** restic and BorgBackup will verify repository integrity on demand and you should run those checks — but integrity is not recoverability. A backup you have never restored is a hypothesis.
4. **Configuration in version control, not just data in backups.** Data restores get you your content back. Configuration in Git gets you the whole machine back.

Point three is the one people skip, and it is the one that converts a backup from a feeling into an asset. The reason I was calm was not that backups existed. It was that I had watched one work.

What this actually proves

A compromise is not a binary event where you are either fine or ruined. It has a radius, and the radius is something you design.

- **The sandbox did its job.** Someone had code execution on my machine for hours and could not get out of one container. That containment was decided long before the incident, and it turned a potential catastrophe into a few hours of downtime.
- **Do not let accounts exist without verification.** On any system that runs code, an account created from an unverified form submission is a code execution grant. Close self-service registration or gate activation behind real verification — and put a second factor on the accounts that remain, since the two controls guard different doors.
-

Backups buy decisions, not files. The tested restore is the reason I could act deliberately instead of desperately. That calm is the real deliverable.

- **My alert was an outage, not a security tool.** Worth admitting: containment limited the damage but told me nothing was wrong. A performance symptom did. Resource alerting on sustained CPU is the cheap fix, because a quieter miner would still be running.

> Design the blast radius while nothing is wrong, and test the restore before you need it. Do those two things and a breach becomes an inconvenience instead of an emergency.

If you are running your own infrastructure and want a second pair of eyes on where your blast radius currently ends, let's talk.

INDEPENDENT · AD-FREE · READER-SUPPORTED

Free to read. Not free to make.

Every tutorial, article, and line of example code on Holger's Code is published without a paywall and without ads. This page is about what keeps it that way.

Professional work, published free

The content here is held to the same standard as professional client work: every tutorial is built as a real project, tested against current versions, documented step by step, and revisited when the frameworks move on. A single in-depth tutorial routinely takes days — and that's before the bills behind it: servers, storage, build infrastructure, licenses, and the hardware everything is verified on.

If an article or tutorial here has saved you an afternoon of debugging, taught you a technique, or helped you ship, it created real value for you. Supporting it isn't charity — it's paying for professional work you found useful, at whatever price you choose. And if you can't, keep reading anyway. That's what the free part means.

Ways to support

KO-FI

Buy me a coffee

A one-time tip — no account, no subscription, any amount.

BOOKS & COURSES

Buy the paid work

The paid tier of everything published free here — and you get something lasting in return.

SPREAD THE WORD

Share an article

Send a tutorial to a colleague, cite an article, subscribe to the RSS feed.

Nothing here goes behind a paywall, whether you chip in or not. But if something on this site earned its keep today, this is where to say so.

Support on Ko-fi —