COSIGN · VERIFY MON 07:12

image novamed:v4.22.1
signed √ VERIFIED
source @a83f04e
shipped @4f2a…e1b9

hashes do not match.

Signed & *Sealed*

A CLOUD SECURITY THRILLER
SUPPLY CHAIN · HEALTHCARE · 6 CHAPTERS

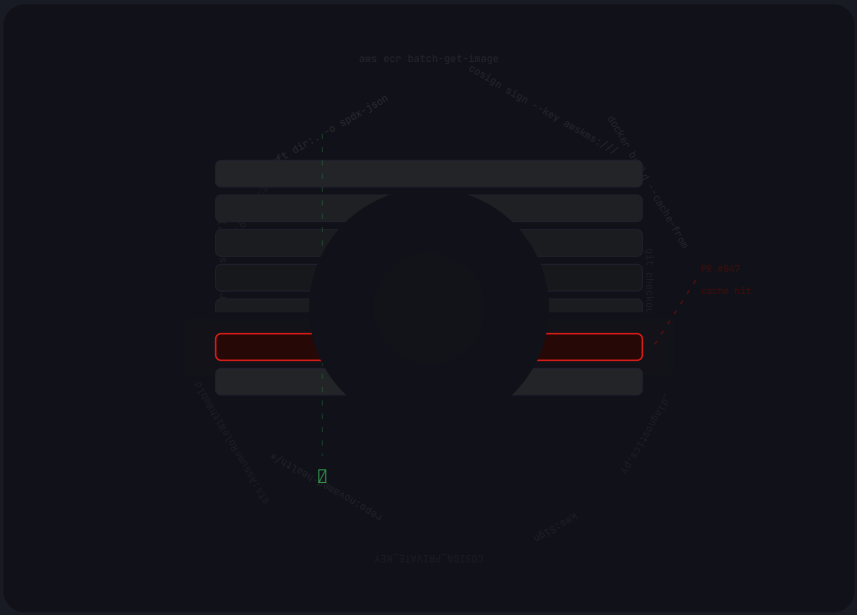
sha256:a83f04e… √ · sha256:4f2a…e1b9

CSCU · BOOK 02 RAAJHESH KANNAA

PATIENT	NOVAMED, INC.
AGE	7 years in production
VITALS	compromised
ADMITTED	23 FEB 2026 · 07:12 EST
DISCOVERED BY	J. PARK · Application Security
REFERENCE	§164.402 Breach Notification Rule
CLASSIFICATION	INTERNAL · PRIVILEGED · DO NOT DISTRIBUTE
FILED	2026-02-27

The following narrative is a reconstruction. It is fiction. The techniques, events, and artifacts it describes are real — present in supply-chain security literature, public post-incident reports, and the author's professional experience. Nothing here enables an attack a motivated adversary could not already execute. Everything here enables a defense.

INTERNAL



This is fiction. NovaMed is not a real company. No real hospitals, patients, or patient data were involved. The techniques described are real. GitHub Actions cache poisoning, OIDC trust policy misconfiguration, container signing bypass, and DNS data exfiltration are documented in public security research, CVE databases, and post-incident reports. The ICD-10 codes are real diagnostic classifications. The FDA regulatory framework for Software as a Medical Device is real. This story exists to teach defense, not to enable offense.

For everyone who approved
a `repo` *-scoped token*
two roadmaps ago

and never came back.

▯ *the book turns here.*

Chapter 1

The Hash That Didn't Match

CHANGESET / 01 4f2ae1b

@@ what Jess believed @@

```
- the pipeline signs what we build.
+ the pipeline signs what it is told to build.
```

> AUTHOR'S NOTE
>
> *I wrote the verify step. I wrote the TODO. I never came back.*
>
> — J.P., Mon 09:02

• • •

Monday morning. 7:12 AM. The coffee is already cold.

I made it forty minutes ago, set it on the desk next to my keyboard, and forgot about it the way you forget about a coffee

when you're three hundred lines deep into an SBOM audit and the numbers aren't making sense yet. They will. They always do. This is the boring part of the job, and I love the boring part.

I'm Jess. I run application security at NovaMed. Healthcare SaaS — we build the platform that 400 hospitals use to manage patient intake, scheduling, and clinical workflows. Six hundred employees, eighty AWS accounts, and a deployment pipeline I designed from scratch three years ago when I joined as employee number ninety-something and the "build process" was a shell script on someone's laptop that pushed Docker images to ECR with `latest` as the only tag.

That pipeline is my cathedral. GitHub Actions for CI, ECR for container storage, cosign for image signing via KMS, and `syft` generating SBOMs — software bills of materials, basically an ingredient list for every container — at every stage. Every image that makes it to production is signed, scanned, and catalogued. I know what's in our containers the way a pharmacist knows what's in the bottles. That's not arrogance. That's the job. When your software auto-updates into hospital networks and a bad build could knock out an emergency department's intake system, you don't get to be casual about provenance. I sleep well because the pipeline is airtight. If I built it right, nothing gets through. That's the contract.

My team — four people, all sharp, all overworked — handles everything from dependency scanning to pen testing the new features before release. I used to do the pen testing myself. Spent five years at a boutique offensive shop before NovaMed, breaking into things for a living. Turns out I'm better at building walls than climbing them. Or maybe I just got tired of writing reports that nobody read.

Every quarter, I run the full audit. Pull the production container images, regenerate the SBOMs from the source tree, compare hashes. It's a formality. The pipeline generates matching artifacts

every time because the pipeline is deterministic, the builds are reproducible, and I designed it that way. Three years, twelve quarterly audits, zero mismatches.

Today is audit number thirteen.

• • •

The script is simple. I wrote it myself, which means it's ugly and correct — my two favorite qualities in code.

Pull the SBOM artifact from S3. Pull the production image from ECR. Generate a fresh SBOM from the source tree at the tagged commit. Compare the hashes.

```
# Pull production SBOM from S3
aws s3 cp s3://novamed-sbom-artifacts/platform-api/v4.14.2/sbom.spdx.json \
    /tmp/prod-sbom.spdx.json --profile prod-artifacts

# Generate source-tree SBOM at the release tag
git checkout v4.14.2
syft dir:. -o spdx-json > /tmp/source-sbom.spdx.json

# Compare
sha256sum /tmp/prod-sbom.spdx.json /tmp/source-sbom.spdx.json
```

```
a]4f8e2c1b...d9  /tmp/prod-sbom.spdx.json
7c3a91d0e...f2   /tmp/source-sbom.spdx.json
```

I stare at the output.

They don't match.

The production SBOM — the one generated during the actual build and stored in S3 — has a different hash than the SBOM I just generated from the same source code at the same commit tag. This has never happened. Twelve quarters. Thirty-six months. Never.

My first thought is tooling drift. `syft` updated between when the image was built and now. Different version, slightly different output, hash changes. It's the obvious answer. It's the answer I want.

I check the `syft` version pinned in the build workflow. `0.97.1`. I check my local version. `0.97.1`. Same version. Same source. Different hash.

My second thought is that someone manually tagged the release from a dirty working tree. It happens. Developers commit, forget to push, tag locally. The CI pulls the tag but the tree doesn't match. Except our pipeline doesn't work that way — the build triggers on the tag push to remote, checks out from the remote ref, and the SBOM generates inside the build container. There's no local tree involved.

I take a sip of the cold coffee. It's terrible. I drink it anyway.

Third thought: the image itself is different from what the source tree would produce.

I pull the image.

• • •

Container images are layer cakes. Each instruction in the Dockerfile produces a layer, and the layers stack. When you pull an image, you're pulling the manifest — which lists the layer digests — and then each layer individually. If you want to know what's in an image, you read the layers.

```
aws ecr batch-get-image \
    --repository-name novamed/platform-api \
    --image-ids imageTag=v4.14.2 \
    --profile prod-ecr \
    --output json | jq '.images[0].imageManifest' -r | jq '.layers'
```

Seven layers. Our standard base image accounts for five. The application code is layer six. Layer seven is the runtime configuration — environment variable defaults and the entrypoint script. I know these layers by heart.

I diff the layer digests against the expected output from a clean build.

Layers one through five, and seven: match.

Layer six: doesn't match.

I export it.

```
docker save novamed/platform-api:v4.14.2 -o /tmp/platform-api.tar
mkdir /tmp/platform-api-layers && tar -xf /tmp/platform-api.tar -C /tmp/platform-
  api-layers
```

I unpack layer six and start reading. The application modules are there. The route handlers, the models, the service files. And there's something else.

A Python file. `_diagnostics.py`. Forty-one lines. It doesn't exist in the source repository. I've never seen it before.

• • •

The file registers a Flask route. Our API gateway runs Flask. This file hooks into the application startup and adds an endpoint that doesn't appear in any spec, any PR, any design document, any Jira ticket.

`/internal/diag`.

Forty-one lines. No comments. No docstring. Clean code, actually — whoever wrote this knew Flask well. The endpoint accepts a POST request with a JSON body. The body contains a `cmd` field. The handler passes the value of `cmd` to `subprocess.run` with `shell=True`, captures the output, and returns it in the response.

Remote code execution. Hidden behind an innocent-sounding path. Sitting in the container image that's running in production across every hospital network in our fleet.

I read the forty-one lines three more times. Each time I'm hoping I'll see something that explains it away. A debug tool

someone forgot to remove. A feature flag experiment. A health check that got weird. Each time, I see the same thing: `subprocess.run(cmd, shell=True, capture_output=True)`.

That's not a health check. That's a backdoor.

I check the ECR image history. This version was pushed last week, but the anomalous layer has been present since v4.12.0 — March 1st. Three weeks ago. Our deployment pipeline pushes new images through a canary rollout — 10% of clusters, then 50%, then 100% over seventy-two hours. Every hospital running NovaMed has had this code since early March.

Three weeks. Whatever this endpoint does, whoever put it there, they've had three weeks of access to every NovaMed deployment. `shell=True` means they can run anything. Inside the container. Which has IAM credentials via the ECS task role. Which has access to DynamoDB tables containing patient scheduling data. Which connects to RDS instances containing clinical records.

HIPAA doesn't come to mind as an acronym. It comes to mind as every hospital in our network trusting us with their patients' data.

• • •

I set down the cold coffee. My hands aren't shaking, which surprises me. Everything else is. My mind is running the blast radius calculation on a loop, and somewhere underneath it is the thought that I was supposed to call my sister back today. That wasn't happening now.

The pipeline built this image. The pipeline signed this image. The SBOM was generated from this image. Everything downstream of the build is consistent — the SBOM matches the image because the SBOM was generated from the image, not from the source tree. My quarterly audit is the only check that compares the image back to the source. And I only run it once every three months.

Someone knew that.

I open a terminal and verify the image signature, because I need to know.

```
cosign verify \
    --key awskms:///arn:aws:kms:us-east-1:319471200841:key/ab3c7f91-d204-4b78-9e6
  2-1a5c83f6d271 \
    319471200841.dkr.ecr.us-east-1.amazonaws.com/novamed/platform-api:v4.14.2
```

```
Verification for 319471200841.dkr.ecr.us-east-1.amazonaws.com/novamed/platform-ap
  i:v4.14.2 --
The following checks were performed on each of these signatures:
  - The cosign claims were validated
  - The signatures were verified against the specified public key

[{"critical":{"identity":{"docker-reference":"319471200841.dkr.ecr.us-east-1.amaz
  onaws.com/novamed/platform-api"},"image":{"docker-manifest-digest":"sha256:e4d9
  a1c..."},"type":"cosign container image signature"},"optional":null}]
```

Valid. The signature is valid. The KMS key signed this exact image digest. The pipeline built the image, signed it, and shipped it. Everything the pipeline did was correct. The pipeline did exactly what it was supposed to do.

The pipeline built the wrong thing.

I close my laptop halfway, then open it again. Dawn light is starting to leak through the blinds. I've been at this for forty minutes. Forty minutes ago this was a routine quarterly audit and I was thinking about what to have for lunch.

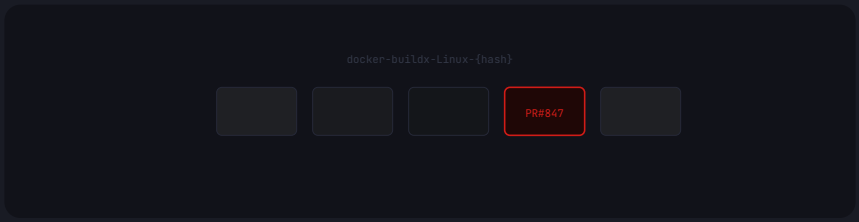
The code is still there on my screen. Forty-one lines. `subprocess.run`. `shell=True`. `/internal/diag`.

This isn't a tooling glitch. This isn't version drift. This isn't a forgotten debug endpoint.

Someone put this here. Someone who understands our build pipeline well enough to inject code into the final image layer without touching the source repository. Someone who knew the signature would still verify. Someone who knew the SBOM would still generate

clean. Someone who knew I wouldn't check for ninety days.

I stare at the forty-one lines and realize I'm not looking at a bug. I'm looking at a door. And it's been open for three weeks.



Chapter 2

Trusted Source

CHANGESET / 02 b91c7d2

@@ what Jess believed @@

- the cache is a speed optimization.

+ the cache is a shared-memory vulnerability.

> AUTHOR'S NOTE
>
> *The PAT was scoped repo because that's what the docs said to do.*
>
> — J.P., Mon 11:30

• • •

The first thing I do is not panic. The second thing I do is close my office door, which I never do, which means Priya on my team will notice, which means I have about twenty minutes before she Slacks me asking if everything's okay.

I need to trace the image backward. Not forward — I already know where it went. Four hundred hospitals. What I need to know is

where it came from. Not the repository. I know the repository. I need to know how forty-one lines of Python ended up in a signed production container without existing in the source tree.

The image was built by our GitHub Actions workflow. The workflow triggers on a tag push to `main`, builds the Docker image on a GitHub-hosted runner, pushes to ECR, signs with cosign, generates the SBOM, stores everything in S3. I designed this. Every step has a log.

I trace backward through the builds. The first compromised version is v4.12.0 — March 1st. 14:22 UTC. The workflow YAML is `release-platform-api.yml`. I read the logs line by line. Checkout: clean. Dependency install: normal. Docker build: normal. Push to ECR: normal. Cosign sign: normal. SBOM generation: normal.

Normal. Normal. Normal. The logs say nothing happened because from the pipeline's perspective, nothing did. It built what it was told to build.

So what was it told to build?

• • •

Docker builds are deterministic given the same inputs. Same Dockerfile, same context, same base image, same build cache — same output. Change any input, the output changes. The forty-one lines aren't in the source tree, which means they aren't in the Dockerfile, which means they came from one of the other inputs.

The base image. Or the cache.

I check the base image first. We use a hardened Python 3.11 image that we build internally and push to a private ECR repository. I pull the base image digest referenced in the `v4.12.0` build and compare it against the known-good digest in our `base-images` repository.

They match. The base image is clean.

That leaves the cache.

GitHub Actions has a build cache. It's one of those features that sounds purely benign — cache dependencies between workflow runs so builds are faster. Docker layer caching, pip dependencies, node modules. Every CI system has some version of it. The implementation detail that matters is this: the cache is shared across all branches in the repository.

A workflow running on a pull request branch can write to the cache. A workflow running on `main` can read from that cache. GitHub's own documentation says this. It's not a bug. It's a design decision. Think of it as a shared refrigerator — anyone in the office can put food in, and the chef grabs whatever's freshest without checking who left it. The rationale is that PR branches need to be able to seed the cache so that subsequent runs on `main` don't start cold. The side effect is that any branch with workflow permissions can poison the cache that `main` consumes.

I know this. I've known this. I read the blog post from that CI/CD security researcher two years ago. I bookmarked it. I never did anything about it.

I start searching the GitHub Actions cache entries for our `platform-api` repository.

• • •

GitHub Actions caches are keyed by a combination of the runner OS, a user-defined key string, and optionally a hash of a lockfile. Our Docker build cache uses the key format `docker-buildx-${{ runner.os }}-${{ hashFiles('requirements.txt') }}`, with a restore-keys fallback of `docker-buildx-${{ runner.os }}-`. Standard pattern. If the exact key doesn't hit, the fallback restores the most recent entry with that prefix — regardless of which branch created it. If the

requirements don't change, the cache hits. Fast builds. Everyone's happy.

I pull the cache entries via the API. The current cache for the `docker-buildx-Linux-*` key was created on February 28th. Three days before the `v4.12.0` release. The cache was populated by a workflow run on branch `docs/update-readme-links`.

I find the pull request. PR #847. Opened February 26th. Merged February 28th. Title: "fix: update broken links in README and contributing guide." Author: `dependabot[bot]`.

No. Not Dependabot. The username is `dependab0t-novamed`. Zero instead of an O. A GitHub account created February 14th. Two weeks before the PR. The account has no other activity — no other repositories, no other contributions, no profile information. Just this one PR to this one repository.

The PR diff is twelve lines. Four Markdown link corrections in `README.md`. Two in `CONTRIBUTING.md`. The kind of PR that gets a thumbs-up emoji and a merge within the hour. Sam approved it.

Sam. Junior DevOps engineer. Twenty-six, earnest, joined NovaMed because he liked the mission — healthcare software that helps people. Owns the CI/CD workflows because nobody else wants to. Uses a personal access token with `repo` scope for the GitHub Actions workflows because the fine-grained tokens "kept breaking" and he couldn't figure out the minimum permissions. I told him to fix it. Twice. He said he would. He didn't. I let it slide because we had a SOC 2 audit to prepare for and I was triaging thirty other things.

The PR's workflow run is where it happened. When the workflow ran on the `docs/update-readme-links` branch, it executed the Docker build step with Docker buildx and the GitHub Actions cache backend. The build step that installs dependencies has a `RUN` instruction that pulls packages and copies application files — the

cached output of that layer is what gets reused across builds. On this run, the attacker's workflow had modified the cached layer output to include `_diagnostics.py` alongside the legitimate application files. The cache stored the poisoned layer, keyed to the unchanged `requirements.txt` hash.

The README PR didn't change the Dockerfile. It didn't change the requirements. It didn't change any Python code. But the workflow ran the Docker build step on every PR — because the workflow was configured that way, because running the build on PRs is how you catch build failures before they hit `main`. Normal. Expected. My design.

And when the `v4.12.0` release workflow ran three days later, it pulled the cache entry that the docs PR had populated. The cache key matched because `requirements.txt` hadn't changed. The poisoned layer slotted in. The Docker build completed. The image was pushed, signed, shipped.

• • •

I lean back and grip the armrests until the vinyl creaks. The HVAC is humming that low frequency that lives just below conscious hearing — you only notice it when everything else goes quiet. Everything else has gone quiet.

The sequence is elegant. Repulsive, but elegant. Create a convincing GitHub account. Open an innocuous PR. The PR triggers the build workflow. The workflow populates the build cache. The cache entry persists until the next matching build on `main`. When the release happens, the cache delivers the payload. No direct modification to the Dockerfile. No suspicious commits to `main`. No access to the signing key. The pipeline does all the work for you.

I need to see the signing chain. I need to understand exactly why cosign verified clean.

Our signing setup uses AWS KMS. The workflow's IAM role can call `kms:Sign` on our signing key. After the Docker image is pushed to ECR, the workflow calls `cosign sign` with the KMS key ARN. (There's also a `COSIGN_PRIVATE_KEY` in our GitHub Actions secrets — a local keypair backup from an incident nine months ago. I keep meaning to audit that. Later.) Cosign computes the image digest, sends it to KMS for signing, and attaches the signature to the image in the registry.

The signature proves that the KMS key signed this specific image digest. It proves that the signing happened via an OIDC-authenticated role, which is the only principal authorized to use the key. It proves the image wasn't tampered with after signing.

It does not prove the image is correct. It proves the image is the one we built. The one we built happened to include a poisoned cache layer.

The signature proves who signed it. It doesn't prove it's the right thing.

• • •

I check the OIDC trust policy. When our GitHub Actions workflows need AWS credentials, they use OpenID Connect — the workflow assumes an IAM role by presenting a JWT token from GitHub's OIDC provider. No static credentials. No access keys stored in secrets. Clean. Modern. Mine.

I pull the trust policy on the role.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Federated": "arn:aws:iam::319471200841:oidc-provider/token.actions.githu
  busercontent.com"
      },
      "Action": "sts:AssumeRoleWithWebIdentity",
      "Condition": {
        "StringLike": {
          "token.actions.githubusercontent.com:sub": "repo:novamed-health/*"
        }
      }
    }
  ]
}
```

`repo:novamed-health/*`.

Not `repo:novamed-health/platform-api:ref:refs/heads/main`. Not even `repo:novamed-health/platform-api:*`. Any repository in the entire `novamed-health` GitHub organization. Any branch. Any workflow. Any PR.

I wrote this policy. Eighteen months ago, when we were rolling out OIDC across all our repositories. I used the wildcard because we had thirty-two repositories and I didn't want to maintain thirty-two separate trust policies. I told myself I'd tighten it later. I didn't.

The docs PR workflow ran on a branch. The OIDC token it presented had a subject claim of `repo:novamed-health/platform-api:pull_request`. The trust policy accepted it. The workflow assumed the role. The role had permissions to push to ECR, to sign with KMS, to write to the SBOM bucket. All of it.

The attacker didn't need to compromise our credentials. They didn't need to steal a key. They didn't need to hack into anything. They opened a pull request, and the system I built gave them everything.

• • •

VEGA found NovaMed the way you find a loose thread on a sweater — by pulling gently and watching what unravels.

The purchase on Cerberus Market wasn't credentials this time. It was reconnaissance. A seller called `gh0stpipe` offered GitHub organization mapping: CI/CD workflow analysis, token scope enumeration, cache behavior profiling. The listing read like a consulting engagement. Price: 0.4 BTC. VEGA paid without negotiating.

The dossier arrived three days later. VEGA read it twice. The second time was for pleasure. Forty-seven pages. NovaMed's GitHub organization structure, every public workflow file parsed and annotated, the OIDC trust pattern identified, the cache key format reverse-engineered from workflow YAML committed to public repositories that NovaMed's developers had forked for reference. The dossier noted that NovaMed's `platform-api` repository used Docker layer caching with a key tied to `requirements.txt`, which changed approximately once per quarter. Forty-five days of observation. Patient work. VEGA appreciated patient work.

The `dependab0t-novamed` account was created on a Tuesday. The README PR was drafted on a Thursday. The rest was arithmetic.

VEGA closed the dossier and thought about the signing key. A system that signs its own poison. There was a word for that. Trust.

• • •

I check the timeline one more time. February 14th: GitHub account created. February 26th: PR opened. February 28th: PR merged, cache poisoned. March 1st: release built from poisoned cache, signed, shipped. March 4th: full rollout to all hospital deployments.

Three weeks. Twenty-one days of a remote code execution backdoor sitting in production, signed by our own key, verified by our own tooling, distributed by our own pipeline. And the only reason I found it is because I run a quarterly SBOM audit that compares image hashes to source tree hashes — a check that exists because I'm paranoid, not because anyone asked for it.

If the audit had been next quarter, it would have been six months. If I'd never built the audit at all, it could have been forever.

Priya messages me on Slack. "Everything ok? Door's closed."

I type "fine" and delete it. I type "busy" and delete it. I type "can you come to my office" and delete that too.

I pick up my phone and call Carmen. Carmen Reyes, our CISO. She was the second security hire after me, brought in when NovaMed hit three hundred employees and the board decided "security team of one" wasn't a great look for a healthcare company courting enterprise contracts. Carmen came from a hospital system — she's done breach response on the provider side, knows what happens when health tech goes dark in an ED. She doesn't panic. She asks the right questions.

The phone rings twice.

"Jess. It's seven fifty in the morning."

"I know."

"This is going to ruin my day, isn't it."

"We have a problem," I say. "And it's been in production for three weeks."

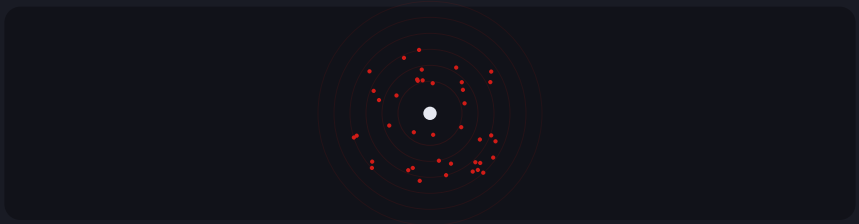
Silence on the line. I can hear Carmen processing — not the information, she doesn't have enough yet — but the tone. The sound of someone who builds systems for a living telling you the system is compromised.

"My office," she says. "Ten minutes. Bring your laptop."

I close the lid. The cold coffee sits untouched. Dawn light cuts hard lines across the desk now, the morning fully arrived, the Monday fully ruined. I grab my laptop bag — it's heavier than it should be, the way it always is when you're carrying the weight of a thing you haven't said out loud yet to anyone who can do something about it.

I designed this pipeline. I trusted this pipeline. I told four hundred hospitals they could trust it too.

I head for the elevator.



Chapter 3

The Blast Radius

CHANGESET / 03 a83f04e

@@ what Jess believed @@

```
- the blast radius is our tenants.
+ the blast radius is their patients.
```

> AUTHOR'S NOTE
>
> *Three hundred names is not a number. Three hundred names is three hundred.*
>
> — J.P., Mon 14:22

• • •

Monday. 1:47 PM.

Carmen's office smells like cold coffee and dry-erase markers. She's already at the whiteboard when I walk in, drawing two columns. One says KNOW. The other says ASSUME. The ASSUME column is longer.

I've worked with CISOs who panic. CISOs who delegate the panic. CISOs who make the panic about themselves. Carmen does none of these things. She uncaps a red marker, writes TIMELINE across the top, and says, "Start from the beginning. Don't skip anything."

So I don't.

I walk her through the SBOM mismatch. The container layer diff. The hidden endpoint in the trojanized image. The cosign signature that verified clean because the pipeline signed exactly what it built — from poisoned cache. A compromised developer credential — we don't know whose yet. The OIDC trust policy that trusts every repo in the org.

Carmen listens without interrupting. When I'm done, she puts the cap back on the marker and says, "How many hospitals?"

"All of them."

She doesn't blink. She writes 400 on the whiteboard, circles it twice, and says, "Show me."

• • •

I pull up my laptop and connect to the conference room display. The war room is just Carmen's office with the door closed, at least for now. She'll spin up the full incident response team by end of day — legal, compliance, engineering leads, external counsel. But right now it's the two of us, and I need to map the blast radius before anyone else walks in.

ECR first. Every container image we've ever pushed to production lives in our registry, tagged by version and git commit SHA. I need to know which versions contain the trojanized layer.

```
aws ecr describe-images \
    --repository-name novamed/platform-api \
    --filter tagStatus=TAGGED \
    --query 'imageDetails[?imagePushedAt>=`2026-02-24`] | sort_by(@, &imagePushed
  At)' \
    --output table \
    --profile prod-ecr
```

```
-------------------------------------------------------------------
|                        DescribeImages                           |
+------------------+--------------------------+-------------------+
|    imageDigest   |      imagePushedAt       |    imageTags      |
+------------------+--------------------------+-------------------+
|  sha256:a3f8c1...| 2026-02-24T03:14:22Z     | v4.12.0, latest   |
|  sha256:b7d2e9...| 2026-02-27T18:41:07Z     | v4.12.1           |
|  sha256:c4a1f0...| 2026-03-03T09:22:15Z     | v4.13.0           |
|  sha256:d8e3b2...| 2026-03-07T14:55:33Z     | v4.13.1           |
|  sha256:e2f4a7...| 2026-03-10T11:08:44Z     | v4.14.0           |
|  sha256:f1c9d3...| 2026-03-14T08:30:19Z     | v4.14.1           |
|  sha256:a9b0e6...| 2026-03-17T02:17:51Z     | v4.14.2, latest   |
+------------------+--------------------------+-------------------+
```

Seven versions in three weeks. Every single one built from the poisoned cache. Every single one signed with our KMS key. Every single one passed vulnerability scanning, passed our Inspector checks, passed policy validation. Because none of those tools check whether the source code matches the artifact. They check the artifact against known-bad signatures. This artifact has no known-bad signature. It was custom.

"v4.12.0 through v4.14.2," I say. "Everything since February 24th."

Carmen writes the version range on the whiteboard. "Which hospitals are running which versions?"

This is the part where I get to feel competent for about thirty seconds before the scale of it crushes me.

Our deployment pipeline pushes to ECS. Each hospital customer gets a dedicated ECS cluster — network isolation, data residency, because healthcare compliance doesn't believe in multi-tenancy and neither do I. I wrote the deployment automation. Querying it is the one thing today I'm not worried about.

Every hospital is configured for auto-update. When we push a new image to ECR and update the task definition, ECS rolls the new tasks across every cluster. Rolling deployment. Health checks pass, old tasks drain, new tasks take over. Zero downtime. The whole thing takes about forty minutes per cluster.

I designed it that way. Fast deploys, minimal human intervention, consistent state across the fleet. It was a good design. It is a good design. It just doesn't account for the possibility that the thing being deployed is hostile.

"All of them," I say. "Every cluster auto-updated. Some are on v4.14.2, some might be on v4.13.x if they had health check issues that paused the rollout. But the trojanized layer is present in every version since v4.12.0. There's no clean version running anywhere."

Carmen puts the marker down. She pulls a chair to the table and sits across from me. Her left hand is doing the thing it always does under stress — she's pressing her thumb against each fingertip in sequence, pinky to index, index to pinky, a metronome of controlled anxiety.

"Walk me through the OIDC policy," she says. "The one that let this happen."

• • •

Six months ago, Ray called a meeting. Ray is VP of Engineering — good at his job in the way that matters to boards: shipping features, hitting roadmap targets. The meeting was about deploy friction. Our pipeline had a manual approval gate. A human had to click "Approve" in GitHub Actions before the deployment rolled out to production. That human was usually me or Sam.

Ray's argument was simple: the gate added two hours to every deploy, had caught zero issues in eighteen months, and was a rubber stamp. "If we trust the signing," Ray said, "we should trust the

signing."

He was right. The gate was security theater — my own security theater. I didn't fight it. Carmen wasn't CISO yet. By the time she inherited the pipeline, the approval gate was already gone and deploy times had dropped from three hours to forty minutes.

The OIDC trust policy was part of that same push. Our GitHub Actions workflows assume an IAM role via OIDC federation to push images to ECR and update ECS task definitions. The trust policy controls which workflows can assume the role. I'd originally scoped it to the specific repo and branch:

```
repo:novamed-health/platform-api:ref:refs/heads/main
```

Ray wanted it broader. Other repos needed to trigger deploys — the infrastructure repo, the config repo, the monitoring repo. Rather than create separate roles for each, the team widened the trust policy to the org level:

```
repo:novamed-health/*
```

Any workflow in any repo in the NovaMed GitHub org could now assume the production deployment role. It was a convenience decision. Made in a 30-minute meeting. Documented in a Jira ticket that nobody has looked at since.

That trust policy is why a cache-poisoned build from a docs-update PR could sign and deploy a trojanized image to every hospital in our network. The pipeline trusted the org. The org included every repo. Every repo included every branch. Every branch included every PR.

I explain this to Carmen. She writes the OIDC policy on the whiteboard — the current one and what it should have been.

"Who approved this change?" she asks.

"Ray. In a meeting. I was there."

"Did you push back?"

"Not hard enough."

She nods. Not judgment. Inventory. She's cataloging who knew what, when, and what decisions led here. She'll need that for the incident report. She'll need that for the lawyers.

• • •

Carmen calls the war room to order at 3 PM. By then she's pulled in David from legal, Priya from compliance (our Priya, not Dr. Anand — that call comes tomorrow), Sam, two senior engineers from the platform team, and Ray.

Ray walks in last. He's reading something on his phone. He sits down, looks at the whiteboard with its timeline and version numbers and the number 400 circled in red, and says, "What am I looking at?"

Carmen runs it. Clean, structured, no editorializing. Compromised build artifact. Supply chain attack. Poisoned GitHub Actions cache. Valid cosign signature on a trojanized image. All customers affected.

Ray's face changes the way faces change when someone who builds things realizes the thing they built is the thing that broke. Not panic. Recognition.

"The OIDC policy," he says quietly.

Carmen nods.

"That was my call."

"We know."

Ray doesn't make excuses. I'll give him that. He doesn't say "but the deploy times" or "but the team velocity." He sits down and says, "What do you need from engineering?"

Then, quietly, to no one in particular: "I have the same meeting on my calendar next Tuesday. Different repo, same pattern. I was going to approve it again."

The room is silent for three seconds. Ray looks at the whiteboard. Carmen looks at Ray. I look at the table.

Carmen assigns workstreams. David from legal starts drafting the notification framework — HIPAA requires breach notification within 60 days of discovery, but "discovery" is a legal term with specific meaning and David needs to nail that down before the clock starts. Priya from compliance starts the HIPAA Security Rule assessment — which administrative, physical, and technical safeguards failed, and how do we document the gap analysis. The platform engineers start building the list of which ECS clusters are running which task definition versions.

Sam is in the room. He doesn't know his PAT was the initial access vector yet. Carmen and I made that call together — we'll tell him, but not in front of the group, and not until we've confirmed it with forensic evidence. Right now he's an asset, not a liability. He knows the workflows better than anyone.

I should be telling them about the hidden endpoint. I should be telling them that the trojanized image isn't just compromised — it's actively serving patient data to anyone who knows how to ask.

I don't.

Here's my reasoning, and it's bad reasoning, and I know it's bad reasoning even as I'm doing it: I need to understand the scope. If I tell fifteen people that there's an active data exfiltration endpoint in production right now, someone is going to do something impulsive. Pull the service. Kill the containers. Trigger a cascading failure across every hospital system in our fleet. A hospital EHR going down isn't like an e-commerce site going down. People are making medication decisions based on data from our platform. Dosage

calculations. Allergy checks. Lab result interpretations.

If we kill the service, hospitals fall back to manual processes. Paper charts. Phone calls to labs. The kind of thing that works until it doesn't, until someone misreads a handwritten dosage and a patient gets ten times the prescribed amount of heparin.

So I sit on the information. Three hours. I tell myself I'm being responsible. I'm prioritizing patient safety. I'm preventing a panic response.

What I'm actually doing is making a unilateral decision about risk that isn't mine to make. Carmen is the CISO. It's her call. But I'm the one who found the endpoint, and I'm the one who understands what pulling the plug means, and somewhere in the gap between those two facts I convince myself that knowing more before escalating is the right move.

It's not the right move. It's the comfortable move. There's a difference.

• • •

6:14 PM.

The war room has been running for three hours. The blast radius is mapped. Every cluster confirmed compromised. David from legal has a preliminary notification timeline. Carmen has a call with outside counsel at 7 PM.

I refresh the WAF dashboard. Three new blocked requests to `/internal/diag` in the last hour. Source: a hospital cluster in Pennsylvania. Someone is querying the endpoint while we sit here. I can see the timestamps. 5:47 PM. 5:52 PM. 6:01 PM. Each one a request that, if it gets through, returns patient records. Each one proof that the clock is running and I'm watching it run.

I stand up from the table. My neck has been making sounds it shouldn't. I've been staring at the same deployment dashboard for so long that when I close my eyes I can still see the cluster names scrolling.

"Carmen. I need to show you something."

She looks at me. Reads something in my face. Gets up without a word and walks to the hallway. I follow.

"I found something else in the trojanized image," I say. "This morning. Before the briefing."

Her expression doesn't change. She's waiting.

"There's a hidden API endpoint. `/internal/diag`. It's not in our source code. It was injected via the cache-poisoned build layer. It responds to a specific User-Agent and header combination. When you hit it with the right request, it queries the application database and returns patient records."

"When did you find this?"

"11 AM."

"It's 6 PM."

"I know."

Carmen is quiet for four seconds. I count them. In those four seconds I watch her calculate the implications — the data that may have been exfiltrated in the seven hours I sat on this, the notification timeline that just shifted, the legal exposure that just doubled.

"That's not a backdoor," she says. "That's a data pipeline."

She's right. A backdoor gives you access to the system. A data pipeline gives you access to the data. This endpoint isn't a way in. It's a way out. Someone built an exfiltration channel directly into our application, signed it with our own key, and deployed it to every

hospital in our network using the pipeline I designed.

Carmen turns and walks back into the war room. I hear her voice, steady and clear: "Everyone stop what you're doing. We have a new priority."

I stand in the hallway for a moment, alone with the shame of knowing and choosing to wait. Every hour I sat on that information was another hour the endpoint was live. Another hour of patient records flowing to whoever built it.

From inside the war room, I hear Carmen's voice: "We need to assume active exfiltration."

She's already ahead of me. She's been ahead of me since 1:47 PM. I walk back in and start thinking about what else I haven't looked at yet.

One commit. Four hundred hospitals.

… × 394 more hospitals …

~ 2.3M patients in the blast radius.

~ 300 records exfiltrated. Each one a person.

What 7h 14m cost.

11:00 — DISCOVERY *Jess knows.*

12:00 —

13:00 —

14:30 —

16:00 —

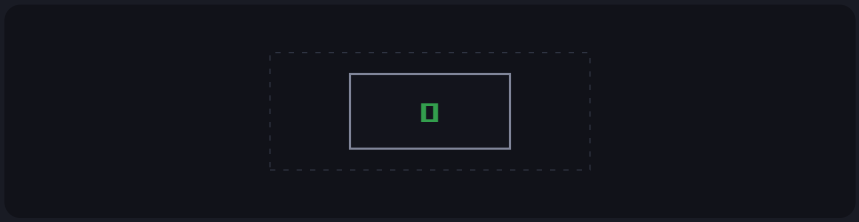
17:40 —

18:14 — DISCLOSURE *Jess tells Carmen.*

HIPAA clock starts at 11:00. Not 18:14.

— David (legal counsel), Thu morning

Discovery starts when Jess knew. Not when you told me.



Chapter 4

Clean Room

CHANGESET / 04 2d7e091

@@ what Jess believed @@

```
- we can trust the tools to diagnose the tools.
+ the pipeline is the crime scene.
```

> AUTHOR'S NOTE
>
> *The pipeline is the crime scene. You can't use the crime scene to process the crime scene.*
>
> — J.P., Tue 06:30

• • •

Tuesday. 5:48 AM.

I haven't slept. Not a badge of honor — just a fact, like the temperature or the day of the week. My eyes feel like someone sanded them. The war room smells like pizza boxes and desperation. Sam is asleep on the couch in the break room. Carmen went home at midnight, back at 4 AM. She changed her shirt. I didn't.

Here's the problem with incident response in a supply chain attack: you can't trust the tools you used to build the thing, because the thing was built with compromised tools. The pipeline is the crime scene. You can't use the crime scene to process the crime scene.

So you build a clean room.

A clean room, in this context, is a build environment constructed from first principles. No cache. No shared state. No inherited trust. Every dependency pinned to a verified hash. Every tool downloaded from a known-good source and checksum-verified before execution. The build happens in isolation, on infrastructure that never touched the compromised pipeline.

It's the software equivalent of performing surgery in a sterile field. You don't just wash your hands. You assume everything is contaminated until proven otherwise.

• • •

First: the signing key.

Our cosign key pair is backed by AWS KMS. The private key never leaves the HSM — signing happens via API call. This is the right architecture. It means the attacker never had the raw private key. What they had was the ability to invoke `kms:Sign` through the OIDC-federated role, which means they could sign any artifact they wanted, but they couldn't extract the key itself.

Small comfort. The key is still compromised in the meaningful sense: artifacts signed with it can't be trusted. Every image signed by that key since the OIDC policy was widened is suspect. We need to rotate.

Carmen approved the key rotation at 11 PM last night. I execute it at 6 AM.

```
# Generate new cosign key pair backed by a new KMS key
cosign generate-key-pair \
    --kms awskms:///alias/cosign-signing-v2

# Output:
# Public key written to cosign.pub
# KMS Key ID: a1b2c3d4-5678-90ab-cdef-EXAMPLE11111
# KMS Key ARN: arn:aws:kms:us-east-1:ACCOUNT_ID:key/a1b2c3d4-...
```

New key pair. New trust root. The old key stays active for now — we can't disable it until every hospital has migrated to images signed by the new key, and that migration hasn't started yet. We'll disable it when the last hospital confirms they're running clean.

I schedule the old key for deletion with a 30-day waiting period. Long enough to handle stragglers. Short enough to limit the window.

Next: the clean room build itself.

I write the workflow from scratch. Not modify — write. The existing workflow is evidence now. I don't touch it. I create a new file in a new repo, on a new branch, with a new OIDC trust policy scoped to exactly one repo, one branch, one workflow.

The trust policy this time:

```
{
  "Condition": {
    "StringEquals": {
      "token.actions.githubusercontent.com:sub": "repo:novamed-health/platform-ap
  i-clean:ref:refs/heads/main",
      "token.actions.githubusercontent.com:aud": "sts.amazonaws.com"
    }
  }
}
```

`StringEquals`. Not `StringLike`. No wildcards. One repo. One branch. The way it should have been from the start.

The workflow is stripped to the bone. No cache. No matrix builds. No reusable workflows from shared repos. Every action pinned to a full commit SHA, not a tag — tags can be moved, and we just learned what happens when you trust mutable references.

I verify the source against our pre-poisoning builds. The last clean build was v4.11.3, from the week before the cache poisoning PR merged. I diff the source tree at that commit against `main`, confirm the application code is identical.

The clean image builds in fourteen minutes. Every layer from scratch. Base image pulled by digest, not tag. Dependencies from a lockfile with integrity hashes.

I sign it with the new key. Then, because I'm paranoid and paranoia is the only thing that's working for me right now, I verify both signatures — the old compromised key should NOT have signed this image, and the new key should have.

```
# Verify: new key DID sign the clean image
cosign verify \
    --key awskms:///arn:aws:kms:us-east-1:ACCOUNT_ID:key/a1b2c3d4-... \
    ACCOUNT_ID.dkr.ecr.us-east-1.amazonaws.com/novamed/platform-api:v4.15.0-clean

# Verification for ACCOUNT_ID.dkr.ecr.../novamed/platform-api:v4.15.0-clean --
# The following checks were performed on each of these signatures:
#   - The cosign claims were validated
#   - The signatures were verified against the specified public key
# [{"critical":{"identity":{"docker-reference":"ACCOUNT_ID.dkr.ecr...

# Verify: old key did NOT sign the clean image
cosign verify \
    --key awskms:///arn:aws:kms:us-east-1:ACCOUNT_ID:key/OLD-KEY-ID... \
    ACCOUNT_ID.dkr.ecr.us-east-1.amazonaws.com/novamed/platform-api:v4.15.0-clean

# Error: no matching signatures
# main.go:62: error during command execution: no matching signatures
```

Clean. Verified. Signed by the right key, not signed by the wrong one.

I push the image to ECR. Tag it `v4.15.0-clean`. The `-clean` suffix is vanity — there's no technical reason for it. But when this shows up in deployment logs six months from now, I want whoever reads it to know this was the one we built by hand, in a sterile field, at 6 AM on a Tuesday while the rest of the pipeline was on fire.

• • •

8:30 AM. The clean image deploys to our staging environment. Automated tests run. Integration tests. Contract tests. Performance baseline. Everything passes.

Carmen walks into the war room at 9 AM and I show her the results. Green across the board. Clean build, clean sign, clean deploy. We're ready for production.

For about forty-five minutes, I feel something I haven't felt since Monday morning. Not relief exactly. More like the absence of active dread. A pause in the signal.

• • •

10:07 AM. Carmen's phone rings. She looks at the caller ID, and I watch her posture change — shoulders back, jaw set, the specific body language of someone about to have a conversation they've rehearsed but hoped to avoid.

"This is Carmen Reyes, NovaMed CISO."

Pause.

"Yes. We submitted the notification this morning. Thank you for the prompt response."

Longer pause. Carmen is nodding at things I can't hear.

"I understand. Yes. We have the emergency update ready to deploy. We can begin rolling it out to production clusters within the hour."

The longest pause yet. Carmen sits down.

"Forty-eight to seventy-two hours."

She hangs up. She sets the phone on the table very carefully, the way you set down something fragile.

"FDA," she says. "CDRH Division of Digital Health Technology."

I know what's coming. I know it the way I knew the SBOM mismatch wasn't a glitch. Pattern recognition, except the pattern is bureaucratic instead of technical.

NovaMed's platform is a Class II medical device. SaMD — Software as a Medical Device. The 510(k) clearance took fourteen months and aged me five years. The classification means every change that touches clinical functionality — including backend services that process lab results and prescription data — needs FDA sign-off. Even the emergency ones. Especially the emergency ones.

Emergency updates to a Class II medical device require notification to the FDA's CDRH. The expedited pathway exists. But "expedited" in regulatory terms means 48 to 72 hours, not 48 to 72 minutes.

"They want to review the update before we deploy," Carmen says. "Standard protocol for a Class II SaMD security-related modification."

"We don't have 48 hours," I say. "The endpoint is live. Right now. Every hospital running the compromised image is serving patient data to whoever built that endpoint."

"I know."

"We have the clean image. It's tested. It's signed with the new key. We can start pushing it to production clusters right now."

"No."

The word sits between us like a wall.

"Jess. Listen to me. Our software is a regulated medical device. If we deploy an unauthorized update to four hundred hospital systems and something goes wrong — a service disruption during a critical care workflow, a data processing error that affects a lab result, anything — we're not just in breach of FDA regulations. We're in

criminal territory. Federal criminal territory. The kind where individual executives get named in indictments."

"People are being exfiltrated RIGHT NOW."

"And if we push an emergency update that wasn't reviewed, and a hospital's EHR integration breaks during a code blue, and a patient dies because the attending couldn't pull their medication history — what then? We deployed unauthorized software to a medical device. We knew the regulatory requirement. We chose to skip it."

I open my mouth. Close it. She's right. I know she's right. The part of me that knows she's right is fighting the part of me that knows what `/internal/diag` does.

This is the thing they don't teach you in security certifications. The moment when compliance with one regulation — FDA device oversight — means non-compliance with another — HIPAA's requirement to prevent unauthorized disclosure. When doing nothing is a violation and doing something is also a violation.

Carmen sees it on my face. She softens, slightly.

"I've already asked David to push for expedited review. We're submitting the technical documentation to CDRH within the hour. We're flagging the active security threat. The FDA has a process for this. It won't be 72 hours. Probably closer to 36."

She pulls out her phone. Shows me a photo. A hospital hallway, fluorescent-lit, blurred at the edges like it was taken in a hurry. "Sinai Health, 2019. Ransomware took down their EHR for eleven hours. I was the IT director. A nurse hand-calculated a heparin drip and got the decimal wrong. The patient coded." She pauses. "She survived. I left hospital IT the next week."

She puts the phone away. "That's why we wait for the FDA. I've seen what happens when we don't."

Thirty-six hours. At the rate I estimated — 10 to 20 records per day — that's another 15 to 30 patient records exfiltrated before we can deploy the fix. Mental health diagnoses. HIV status. Substance abuse treatment records. The kind of data that doesn't just violate privacy. It destroys lives.

"What about containment?" I ask. "We can't deploy the clean image. Can we block the endpoint at the network level?"

Carmen considers this. "Would that require a change to the deployed software?"

"No. WAF rule. Block requests matching the specific User-Agent and header pattern that the endpoint responds to."

"Do it."

I turn to my laptop. Even as I write the WAF rule, I know it's insufficient. I reverse-engineered the endpoint's authentication mechanism from the decompiled container layer, but the attacker can rotate their parameters. A speed bump, not a wall.

I deploy it anyway.

• • •

The attacker anticipated this. I keep coming back to that thought, turning it over like a coin.

Not the WAF rule. The FDA. The regulatory framework. The 48-to-72-hour review window.

Think about it. If you're going to compromise a healthcare SaaS platform, you pick a clinical-grade platform — SaMD, FDA-regulated — not a consumer health app that can push hotfixes whenever it wants. You pick the target where the regulatory framework itself creates a window between discovery and remediation. Where the very thing that's supposed to protect patients becomes a shield for

the attack.

VEGA. The name from the Cerberus Market listing, the one Carmen's external threat intel team pulled from the dark web forensics. Patient, methodical, sophisticated. And clearly familiar with how healthcare compliance works.

• • •

2:30 PM. The war room has expanded to the large conference room on four. Screens on every wall. Carmen is on the phone with outside counsel. David is on a parallel call with our cyber insurance carrier. The platform team is preparing deployment manifests, ready to push the moment FDA gives clearance.

Sam comes up to me. He looks like I feel, which is to say terrible.

"Jess, the cache poisoning PR — the workflow shouldn't have had permission to write to the production cache. But GitHub Actions cache is scoped to the repo, not the branch. Any workflow run in any branch can write to the cache, and the default branch gets read access to all entries."

"I know," I say. "I traced it yesterday."

Sam is quiet for a moment. Then: "I feel like I should have known this."

He means the PAT. We told him an hour ago. Carmen and I, in a small room, privately. His `repo`-scoped token, phished three weeks ago. The initial access vector.

I told him what I would have wanted someone to tell me: it wasn't his fault, phishing works because it's designed to work, the real failure was systemic. A `repo`-scoped PAT shouldn't have been able to trigger a production deployment even if compromised. The OIDC policy was the root cause, not his token.

I believed about sixty percent of what I said. The other forty percent was thinking about the manual approval gate. The one Ray removed. The one I didn't fight hard enough to keep.

"Go home," I say. "Sleep. Come back tomorrow. I need you sharp for the deployment marathon."

He leaves. I watch him go and think about junior engineers carrying guilt for systemic failures. The system let them fail, then blamed them for failing.

• • •

6:00 PM. Tuesday.

I'm sitting in the war room, alone for the first time all day. Everyone's on dinner break. My apartment is fourteen minutes away and it might as well be in another country. The screens glow with dashboards I've been staring at for ten hours.

The deployment dashboard shows v4.15.0-clean staged and ready. Green checkmarks down the line. Built from verified source. Signed with the new key. Tested in staging. Deployment manifests prepared for the entire fleet.

The FDA submission status shows "Under Review." Submitted at 11 AM. No response yet.

The WAF metrics show 14 blocked requests matching the `/internal/diag` pattern since I deployed the rule at 2 PM. Someone is actively querying the endpoint. The WAF caught them. But if the attacker rotates their User-Agent, the rule is useless.

I pull up the ECS console and look at the production clusters. Green dots across the fleet. All healthy. All running the trojanized image. All serving the hidden endpoint to anyone who knows the right handshake.

The clean image is right there. One command, forty minutes, rolling deployment across the fleet. I could do it right now. Carmen isn't in the room. Nobody is in the room. By the time anyone noticed, half the clusters would already be running clean.

I think about this for longer than I should. My fingers don't hover. They type. `aws ecs update-service —force-new-deployment —cluster novamed-prod-` and the cursor blinks after the hyphen, waiting for a cluster name. I stare at it. The cursor blinks. I delete the command character by character. Not Ctrl+C. One letter at a time. Each deletion deliberate, the way I write audit scripts — careful, precise, irreversible in the other direction.

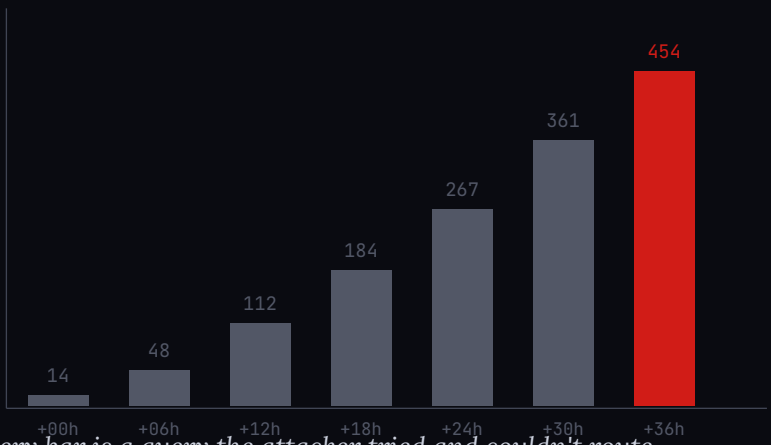
I close the ECS console and open the FDA submission tracker instead.

Carmen is right. The regulatory framework exists because software that processes clinical data is different from software that processes shopping carts. The consequences of a bad deploy aren't a checkout failure. They're a misread lab result, a wrong dosage, a patient harmed by the fix for the thing that was harming patients.

So I sit. And I wait.

Somewhere, right now, someone is querying `/internal/diag` and getting patient records back. The clean image is ready. The FDA hasn't responded. And the clock is running.

WAF blocks on `/internal/diag`, ticking.



> *Every bar is a query the attacker tried and couldn't route. They're not giving up. They're waiting us out.*

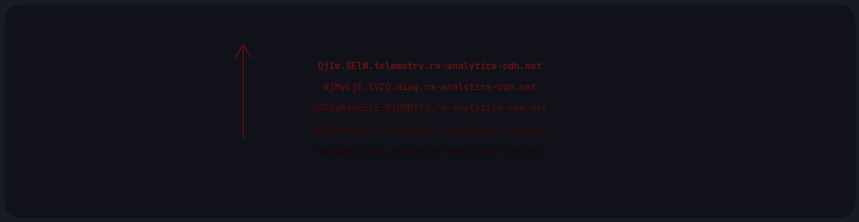
+32h 14m · FDA: ACK

Bucharest · RO

20:00

TUE · EET · DURING THE FDA WAIT

The query-log dashboard refreshes.
Volume is down — not much.
Enough.



Chapter 5

Patient Zero

```
CHANGESET / 05                                  9fb4c17

@@  what Jess believed  @@

-  DNS never leaves the VPC.

+  DNS never lies.
```

> AUTHOR'S NOTE
>
> *We didn't have Route 53 Query Logging either.*
>
> — J.P., Wed 09:14

• • •

Wednesday. 8:07 AM. Forty-eight hours since I found the hash mismatch, and I'm staring at a log that shouldn't exist because I should have been staring at it three weeks ago.

Route 53 Resolver Query Logging. A few API calls per VPC. Create the config, point it at CloudWatch Logs, associate it with the VPC. The cost is a rounding error compared to what we spend on

compute. It catches almost everything that network-level monitoring misses because most exfiltration doesn't look like exfiltration on port 443 — but DNS never lies.

We didn't have it enabled. Nobody does, until they need it. Then they enable it retroactively and spend the next twelve hours wishing they'd done it sooner.

I enabled it at 6 AM. Two hours of queries. Already enough.

• • •

The hidden endpoint wasn't just a backdoor. A backdoor sits and waits. This was a machine.

I'd been focused on the `/internal/diag` route — the `subprocess.run` shell execution — because that's the loud part. The part that screams. A real APT wouldn't leave that sitting next to a covert exfil channel. Unless they wanted it found first, to distract from the quieter mechanism. But there was a second function in `_diagnostics.py` that I glossed over on Monday because it looked like a health check. A background thread, spun up at container start, running on a thirty-minute interval. No inbound trigger. No API call. Just a timer and a function called `_sync_telemetry`.

It queried the application database. Not broadly — it used a parameterized query with specific ICD-10 diagnosis codes hardcoded into the function. Then it encoded the results, chunked them into 253-byte segments, and sent them as DNS TXT record lookups to an external domain.

```
;; QUESTION SECTION
;dGVsZW1ldHJ5.c3luYw.status.rx-analytics-cdn.net. IN TXT
;UGF0aWVudElE.OTg3NjU0.Rjky.rx-analytics-cdn.net. IN TXT
;UGF0aWVudElE.MjM0NTY3.QjIw.rx-analytics-cdn.net. IN TXT
```

`rx-analytics-cdn.net`. Registered three months ago. Nameservers in Romania. The domain looks like a pharmaceutical analytics CDN if

you're scanning logs at 2 AM and your brain is filling in plausible explanations for things that don't deserve plausibility.

Each DNS query carries a few dozen bytes of base64-encoded data as subdomain labels. The TXT response comes back empty — the data is in the query itself, captured by whoever controls the authoritative nameserver. No TCP connection. No HTTP request. No payload in the response. The query is the exfiltration. Every DNS resolver between the container and the internet dutifully forwards it, caches nothing because TXT records for random subdomains don't get cached, and logs nothing unless you're specifically watching.

We weren't watching.

• • •

I decoded the first batch from the two hours of logs I had. Patient IDs and diagnosis codes. I cross-referenced against NovaMed's clinical data dictionary.

The diagnosis codes weren't random.

F32.1 — Major depressive disorder, single episode, moderate.

B20 — Human immunodeficiency virus disease.

F10.20 — Alcohol dependence, uncomplicated.

F11.20 — Opioid dependence, uncomplicated.

F14.20 — Cocaine dependence, uncomplicated.

Every code in the hardcoded list was a mental health diagnosis, a substance abuse record, or an HIV status indicator. Nothing else. Not cardiac. Not oncology. Not diabetes. Not the things that make up 80% of hospital records. The query was a scalpel, not a net.

I pulled the CloudWatch metrics for DNS query volume from the VPCs where we'd just enabled logging and extrapolated backward using the thirty-minute interval and the average payload size per query. Three weeks. Thirty-minute cycles. Ten to twenty records per cycle, varying by which hospital's database the container instance connected to.

Roughly three hundred patient records. Maybe fewer. Maybe slightly more. It didn't matter whether it was 280 or 320. Each one was a person.

• • •

Carmen called an emergency bridge at noon. Legal was on. Compliance was on. Our outside counsel from Baker McKenzie was on. I presented the technical findings. DNS exfiltration. Targeted patient records. Specific diagnosis categories. Three-week window.

The compliance lead asked the question everyone was thinking: "How many affected individuals?"

"Approximately three hundred," I said. "All with mental health, substance abuse, or HIV diagnosis codes."

Silence on the line. Not the silence of people absorbing information. The silence of people calculating consequences. OCR breach notification: for breaches affecting five hundred or more, you notify HHS within sixty days and alert media in each affected state. Under five hundred, you can wait until year-end. We were under five hundred total. Carmen started the sixty-day clock anyway — our outside counsel advised conservative timing for anything involving Part 2 records. May 14th.

• • •

My phone rang at 6:18 PM. Unknown number, 617 area code. Boston.

"This is Dr. Priya Anand. I'm the Chief Medical Information Officer at Brigham Health Network. We received your notification at 2 PM. I've been reviewing it since."

I'd seen her name on the customer list. Brigham was our largest deployment. Twelve hospitals, 4,200 beds, integrated with NovaMed for clinical workflows since our Series B. She'd been the executive sponsor for the integration.

"Dr. Anand, thank you for —"

"I have a question about the diagnosis codes in your notification. You listed ICD-10 categories F10 through F19, F32, and B20. Can you confirm that the exfiltrated records are limited to those categories?"

"Yes. The query in the malicious code specifically targeted those codes. No other clinical data was accessed through this vector."

"So the records that were taken are exclusively patients with mental health diagnoses, substance use disorders, and HIV."

"That's correct."

She paused. When she spoke again, her voice hadn't changed. Same cadence. Same clinical precision. That was the thing about Dr. Anand — she didn't need to raise her voice to make you understand the weight of what she was saying. The facts were heavy enough.

"I want to make sure you understand what those categories mean in practice. Not as data classifications. As patient outcomes."

I didn't say anything.

"There's a reason substance abuse records have their own federal protections — 42 U.S.C. Part 2, separate from HIPAA, stronger than HIPAA. Patients won't walk through our doors if they think the records follow them out. That's not theoretical. We have thirty years of data showing that breaches of substance abuse records correlate directly with reduced treatment enrollment in

affected populations."

"I understand."

"I don't think you do. Not yet." She wasn't being cruel. She was being precise. "I have a patient — I won't name her, obviously — who is seventeen years old. She's in an opioid recovery program at one of our outpatient facilities. Her parents are divorced. There is a custody dispute. If her treatment records appear in any database accessible to either parent's legal counsel, that information will be used in court. Not to help her. To win an argument about who is the more fit parent. She will become evidence in her own custody battle. She will learn that seeking help for her addiction made her a liability in her parents' divorce."

I swallowed against nothing.

"I have another patient. Male, forty-one. HIV-positive. Diagnosed eight months ago. He lives in a town of six thousand people in western Massachusetts. He has not disclosed his status to his employer. He works at a school. He has committed no crime. He presents no risk. If his diagnosis becomes known in that community, he will lose his job. Not formally, not on paper, but he will lose it. He knows this. I know this. This is why he drives ninety minutes to our clinic instead of going to the facility ten minutes from his house."

The fluorescent light above my desk was buzzing. I'd never noticed it before. A high, thin whine at the edge of hearing, the kind of sound your brain filters out until the room gets quiet enough for it to be the only thing left.

"Three hundred records," she said. "You're thinking of it as a number. I need you to think of it as three hundred individual disclosures, each one capable of altering the course of a person's life. Not through identity theft. Not through financial fraud. Through shame."

"Dr. Anand —"

"I'm not assigning blame. I've read your technical notification. I understand that this was a sophisticated supply chain attack. I understand that your team discovered it and is remediating. What I need you to understand is that remediation doesn't undo disclosure. You can patch the software. You cannot un-know something. Those records are in someone's possession, and they will remain in someone's possession, and every one of those three hundred patients will live with that fact whether they know it yet or not."

"Yes," I said. It was the only honest thing I could say.

"We'll cooperate fully with your incident response. I've assigned our privacy officer to coordinate with your legal team. I'll need daily updates on attribution and any recovery of exfiltrated data."

"You'll have them."

"Thank you, Jess."

She hung up. No pleasantries. No softening. Just the click and then the silence and the fluorescent buzz.

• • •

Three hundred people. I kept coming back to the number. Not three million. Not three hundred thousand. A number small enough to imagine individually. Small enough that someone chose each one.

The rate — ten to twenty per day — was deliberate. Any bulk extraction would have triggered the database monitoring alerts we had in place. The WAF would have flagged anomalous query patterns. CloudWatch alarms would have fired on DynamoDB read capacity spikes. VEGA knew our thresholds, or guessed them well enough. Stay under the noise floor. Take a little at a time. The most dangerous exfiltration isn't the kind that trips alarms. It's the kind that looks like Tuesday.

I thought about the ICD-10 codes again. Someone had selected them. Sat at a keyboard and typed `F32.1`, `B20`, `F10.20` into a Python list. Made a conscious decision to target the records that would cause the most damage per record. Not the most valuable on the black market — those would be financial records, insurance data, SSNs. These were chosen for leverage. For the specific, surgical destruction that comes from knowing someone's diagnosis when they don't know you know.

This wasn't data theft. It was a weapon assembled one record at a time.

• • •

I went back to `_diagnostics.py`. I'd read it five times now. The sixth time, I stopped scrolling at line 23. Above the function that built the ICD-10 query, two lines I'd blown past every previous read because I was focused on the `subprocess.run` call:

```
# These are the records no one will admit exist.
# That's what makes them valuable.
```

Fourteen words. Not addressed to me. Not addressed to anyone. A note to self, left in the code the way a carpenter pencils a measurement on a stud that'll be covered by drywall. The person who wrote this understood something about healthcare data that most security professionals never articulate: the protection comes from the shame, and the shame is the vulnerability. The records are valuable precisely because the system would rather pretend they don't exist.

I stared at those two comment lines longer than I stared at the `shell=True`. The backdoor told me what VEGA could do. The comment told me who VEGA was.

• • •

At 10 PM, the forensics team finished the PAT analysis. Sam's personal access token. `repo` scope. Generated eleven months ago. Last rotated: never.

The phishing email was in Sam's trash folder. He'd deleted it but Gmail doesn't really delete anything for thirty days, and his corporate account synced through Google Workspace.

The email was a perfect replica of GitHub's security advisory notification. Subject line: "Action Required: Potential credential exposure in novamed-health repositories." It warned of a leaked secret detected in a public commit and directed Sam to a URL that was one character off from github.com — a homoglyph attack, the Cyrillic 'a' instead of the Latin 'a'. The landing page was a pixel-perfect clone of GitHub's OAuth authorization screen.

Sam had entered his credentials and approved what he thought was a security audit scope. What he actually approved was a token grant to an application controlled by VEGA. The token was generated, transmitted, and operational within four seconds of Sam clicking "Authorize."

I pulled the email up on my screen and read it three times. The grammar was flawless. The formatting matched GitHub's template exactly. The urgency was calibrated — serious enough to act on, not alarming enough to verify through a second channel. The sender domain had valid SPF, DKIM, and DMARC because VEGA had registered a lookalike domain and configured all three. This wasn't a Nigerian prince. This was a professional operation that invested more effort in one phishing email than most companies invest in their entire security awareness program.

Sam was at his desk. It was late. He'd been there since Tuesday. His hoodie was the same one he'd worn on Monday — gray, Kubernetes logo, a coffee stain on the sleeve that had dried into a brown half-moon. He looked like he hadn't slept, which he probably

hadn't.

"I found the email," I said.

He didn't look up. "I know which one it is."

"Sam."

He looked up. His eyes were red.

"I need to show you something." I pulled up my own inbox, scrolled back eight months. Found the email I was looking for. A fake Slack security notification I'd received in June. Same playbook — lookalike domain, urgency, credential harvesting page. "I almost clicked this one. Got as far as the login page before I noticed the URL. The difference between us is timing."

"That doesn't make me feel better."

"It's not supposed to. It's supposed to make you accurate. You didn't cause this. You were the access vector. There's a difference. The access vector for SolarWinds was a compromised build server. Nobody blamed the server."

He almost smiled. Almost.

"Your PAT was `repo`-scoped. It should have been scoped to specific repositories with IP restrictions and a ninety-day expiration. That's a policy failure, not a Sam failure. I approved the PAT policy. I should have required fine-grained tokens when GitHub released them. I didn't because migration was going to take a sprint and we were behind on three other things."

He nodded. The half-moon coffee stain caught the light as he shifted in his chair.

"Go home, Sam. Shower. Sleep. We need you tomorrow for the remediation work, and I need you functional, not martyred."

He left. I watched him walk to the elevator, hoodie pulled up, backpack hanging off one shoulder. Twenty-six years old. His second real job out of college. He'd joined NovaMed because he liked the mission — healthcare software that helps people. And now those patients' most private medical records were in a database in Romania because he clicked a link in an email that looked exactly like it was supposed to look.

• • •

I sat in the empty office. My laptop screen showed the Route 53 query log, the decoded patient IDs, the ICD-10 codes arranged in neat columns.

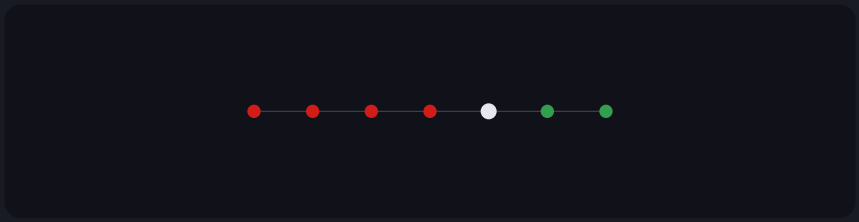
I built this pipeline. Three years ago, in this building, at a desk twenty feet from where I'm sitting now. GitHub Actions for CI because it integrated cleanly. Cosign for signing because it was simpler than Notation and I understood the trust model — or thought I did. The Actions cache because build times were too long. The OIDC trust policy scoped to the organization because scoping it to individual repos meant updating the policy every time we created a new repository.

Every decision made sense at the time. Every decision was a brick in the path that led from Sam's inbox to a database in Romania.

I didn't build a pipeline. I built a weapon. I just didn't know who I was arming.

I closed my laptop. Three hundred people. Somewhere out there, a seventeen-year-old in a custody battle. A man who drives ninety minutes for privacy. And I designed the system that failed them.

I sat in the silence and let it be heavy.



Chapter 6

Chain of Custody

```
CHANGESET / 06                                  c15a83b

@@  what Jess believed  @@

-  I built the pipeline.

+  I built the weapon. I just didn't know who I was arming.
```

> AUTHOR'S NOTE
>
> *I didn't build a pipeline. I built a weapon.*
>
> — J.P., Wed 23:55

• • •

Wednesday night. 11:40 PM. The office is empty except for me and the cleaning crew, and the cleaning crew has learned not to vacuum near my desk when the laptop is open.

I'm writing two incident reports. The first is for the board, for legal, for the regulators. Carmen will review it. Outside counsel will sanitize it. It will use phrases like "unauthorized access to protected

health information" and "sophisticated threat actor" and "immediate remediation steps." It will be accurate without being honest.

The second is for me. It will never leave my laptop. It's the one I'm writing now.

• • •

The FDA authorization came through at 7:14 PM. Emergency cybersecurity patch pathway — CDRH has a process for critical security updates to Software as a Medical Device that bypasses the standard 510(k) review timeline. Carmen had been on the phone with the FDA's Digital Health Center of Excellence since Tuesday afternoon. The reviewer understood that a Class II SaMD running a trojanized container across 400 hospital networks was not a situation that could wait for a 90-day review cycle.

The authorization was narrow: permission to deploy the clean build we'd prepared on Tuesday, to the specific image tag and configuration we'd submitted for review. No scope creep. No additional features. Just the security patch, the clean signing key, and a commitment to submit a full post-market cybersecurity report within 30 days.

Carmen printed the authorization letter on the good printer — the color one in the conference room that nobody uses because the toner costs more than the paper. She taped it to the wall above her desk. I think she wanted something physical. Something she could point to if anyone asked whether we had permission.

By midnight, 340 of the 400 hospitals had applied the patch. The remaining sixty were in maintenance windows that wouldn't open until Thursday morning. For those sixty, the compromised image was still running, but we'd deployed a network-level block on the exfiltration domain via the hospitals' DNS resolvers. Not elegant. Effective.

• • •

Thursday morning. Carmen ran the coordinated disclosure.

The notification list: every hospital's IT security team. Every privacy officer. Every general counsel. The Office for Civil Rights at HHS. State attorneys general in every state where affected patients resided — fourteen states, because the three hundred records were distributed across the hospital network geographically. Our cyber insurance carrier. The FBI's Internet Crime Complaint Center, because outside counsel said we should and I didn't have a reason to argue.

Carmen handled it the way she handles everything — methodically, without visible emotion, each notification drafted and reviewed and sent on schedule. She'd been CISO for six months. She'd inherited the pipeline, the OIDC policy, the PAT standards, the missing Route 53 logging. None of it was hers. All of it was now her problem. She never once said that. Not to me, not to anyone on the bridge calls.

I asked her, at 3 AM on Wednesday, whether she blamed me. For the pipeline design. For the OIDC scope. For the cache configuration. She looked at me with the particular exhaustion of someone who has been awake for forty hours and no longer has the energy for anything but the truth.

"I blame the person who did this," she said. "When we're done, I'll blame the decisions. Right now I blame the attacker."

Fair enough.

• • •

My private incident report. The real one. I wrote it in a markdown file on my local machine because I didn't trust anything else and because the irony of not trusting the systems I built was not lost on

me.

```
## Incident Timeline: Key Decisions

2025-04-15 (T-11 months)
  Sam generates PAT with `repo` scope. Policy allows it.
  I approved the PAT policy. No expiration requirement.
  No fine-grained token mandate. Migration deferred.

2025-06-01 (T-9 months)
  Emergency incident: production deploy blocked by KMS
  throttling. On-call engineer copies cosign private key
  to GitHub Actions secret as temporary workaround.
  Temporary. The word that ends careers.
  The key was never removed. I never audited it.

2025-09-12 (T-6 months)
  Ray approves OIDC trust policy scoped to org wildcard.
  I wrote the policy. Ray approved it. Neither of us
  scoped it to specific repos and refs.
  Reason: "too many repos, too much maintenance."
  Translation: convenience over security.

2025-10-18 (T-5 months)
  Ray removes manual approval gate for production deploys.
  Reason: "slows down releases, team is frustrated."
  I didn't fight it. I should have fought it.

2025-12-20 (T-3 months)
  VEGA purchases CI/CD reconnaissance from Cerberus Market.
  Org mapping, cache behavior, token scopes. Forty-seven pages.
  We don't know this yet. We won't know for months.

2026-01-15 (T-60 days)
  Sam's PAT phished. Access to entire GitHub org.

2026-02-01 (T-45 days)
  Attacker maps repos, identifies cache key patterns.

2026-02-28 (T-23 days)
  Cache poisoning. Trojanized image built, signed, deployed.
  Pipeline does exactly what it's supposed to do.
  The system works. The system is compromised.

2026-03-01 to 2026-03-23
  300 patient records exfiltrated via DNS.
  10-20 per day. Under every threshold we set.

2026-03-23 (T+0, Monday)
  I find the SBOM mismatch. 22 days late.
```

Every line in that timeline is a reasonable decision. A PAT policy that matches industry standard. A temporary workaround during an incident. An OIDC scope that reduces operational overhead. A deployment gate removed to improve velocity. No single decision is

indefensible. Together, they form a kill chain.

• • •

The signing key. This is the part that kept me up Wednesday night.

The cosign signing key was supposed to live in KMS. That was the design. The private key material never leaves the HSM. Cosign calls `kms:Sign` via OIDC-authenticated role assumption. The key is never exposed, never exportable, never accessible outside of the signing ceremony.

But nine months ago, during an incident where KMS was throttling our signing requests and production deploys were blocked, an on-call engineer — I checked the PagerDuty logs, it was Marcus, who left the company four months later — generated a cosign keypair locally and stored the private key in a GitHub Actions secret. `COSIGN_PRIVATE_KEY`. Encrypted at rest, but injected as an environment variable into every workflow run in the repository.

Marcus filed a ticket to remove it after the incident. The ticket was triaged, deprioritized, moved to the backlog, and forgotten. I reviewed the backlog in October. I saw the ticket. I moved it to "next sprint." It never made it to the sprint.

VEGA didn't need the plaintext key. They didn't even need direct access to KMS. The cache poisoning was the whole play — poison the cache, wait for the legitimate release workflow on `main` to pick it up, and let the pipeline sign the trojanized image through its normal KMS flow. Our own workflow, our own OIDC role assumption, our own `cosign sign` command. That's why my `cosign verify` on Monday returned clean. The pipeline signed exactly what it built. It just built the wrong thing.

The OIDC wildcard would have let VEGA sign images directly from a PR workflow — a second path to the same outcome. And if both of those had been locked down? The backup key was right

there. Anyone with a `repo`-scoped token could read it. Nine months of exposure, and the only reason it wasn't the attack vector is that there was an easier one.

The sophisticated cryptographic signing infrastructure I'd designed had a skeleton key duct-taped to the back of the door the entire time. I saw the ticket to remove it. I deprioritized it.

I keep thinking about what Dr. Anand said. You can patch the software. You cannot un-know something. I cannot un-know that the signing key was accessible to any workflow for nine months. I cannot un-know that every design decision I made assumed the build environment was trustworthy, and I never once tested that assumption.

• • •

VEGA was already gone.

We pulled every CloudTrail log from every account for the past ninety days. No anomalous `AssumeRole` events. No direct API calls from unrecognized IP addresses. No console logins. No nothing.

The entire attack was conducted through the pipeline. GitHub Actions workflows running on GitHub-hosted runners. OIDC role assumption from a legitimate GitHub org. Cache operations through GitHub's built-in caching. Image builds on GitHub-hosted runners. Signing through OIDC-authenticated KMS access. Push to ECR through the pipeline's IAM role. Deployment through ECS's rolling update.

Every API call in CloudTrail looked like a normal CI/CD operation. Because it was a normal CI/CD operation. The attacker didn't break in through a window. They walked in through the front door, used our tools, followed our processes, and left through the same door. The only anomaly was a cache entry populated from a PR branch instead of main, and we had no monitoring for that because I

never considered the cache a trust boundary.

I never considered the cache a trust boundary. I'm going to be thinking about that sentence for a long time.

• • •

Thursday afternoon. The remediation list.

```
# Pipeline Security Controls - Post-Incident

cache_isolation:
  key: "${{ runner.os }}-build-${{ github.ref }}-${{ hashFiles('**/lockfiles') }}
  "
  restore-keys: "${{ runner.os }}-build-${{ github.ref }}-"

oidc_trust_policy:
  condition:
    "token.actions.githubusercontent.com:sub":
      "repo:novamed-health/platform-api:ref:refs/heads/main"

signing:
  method: "cosign sign --oidc-issuer=https://token.actions.githubusercontent.com"
```

Six controls in total. Cache isolation. Scoped OIDC. KMS-only signing with keyless mode. Fine-grained PATs. Manual approval gate. Route 53 query logging. I wrote each one knowing that any single one would have broken the kill chain.

Any one. We had none.

• • •

Thursday evening. The office was quiet. Most of the team had gone home. Carmen was still in her office, on the phone with our insurance carrier. Ray was somewhere — I hadn't seen him since Tuesday. He'd been in the board meeting all afternoon. I imagined that meeting was not pleasant for him.

I saved the private incident report. Closed the file. Opened a new terminal.

The pipeline was rebuilt. The signing key was rotated. The cache was isolated. The OIDC policy was scoped. The approval gate was back. The Route 53 logs were flowing. The FDA patch was deployed to 398 of 400 hospitals, with the last two scheduled for their maintenance window overnight.

The system was fixed. I could say that with confidence now. The system that replaced the system I built three years ago was better. Harder. Scoped tighter. Monitored properly.

And the people whose records were taken were still living with the consequences of the system I built first.

I thought about Dr. Anand's patients. The seventeen-year-old in the custody battle. The man who drives ninety minutes for privacy. I didn't know their names. I would never know their names. That was the point of the protections we'd failed to provide — the data was supposed to be anonymous to everyone except the people who needed it to deliver care. Now it was anonymous to me but not to whoever controlled `rx-analytics-cdn.net`.

Sam came back Thursday morning, showered, wearing a different hoodie. He worked the remediation all day without complaint. He implemented the fine-grained PAT migration himself, wrote the runbook, filed the tickets for every team. He didn't need to be told. He's twenty-six years old and he clicked a link in a phishing email and he's going to carry that for a while. Maybe always. I know, because I'm going to carry the OIDC policy and the cache configuration and the backlog ticket I deprioritized for exactly as long.

The difference between a security incident and a Tuesday is one decision, made months ago, that nobody thought to revisit.

• • •

I closed my laptop at 9 PM. The fluorescent light was still buzzing. I reached up and twisted the tube until it went dark. The office settled into the glow of exit signs and the blue light of a forgotten monitor across the room.

Fourteen states. The incident report was saved. The pipeline was rebuilt. The patch was deployed. VEGA was gone — scrubbing infrastructure, moving to the next engagement, already inside something else.

We fixed the system. The damage is procedural now. Lawyers and notifications and regulatory timelines. The part where you fill out forms and make phone calls and pretend that process can contain consequence.

I picked up my bag. The office was dark. The exit sign threw a red wash across the carpet. Somewhere out there, a seventeen-year-old and a man who drives ninety minutes and three hundred other people whose records were in a database they'd never consented to. I used to think the contract was simple: build it right, and nothing gets through. The contract was always a lie I told myself so I could sleep. You don't secure a pipeline by being good enough. You secure it by assuming you aren't.

The elevator arrived. I stepped in. The doors closed.

• • •

• • •

• • •

Six weeks later. A Thursday.

A network engineer named David Okafor at a transit company in Atlanta was reviewing route tables for a hub-and-spoke Transit Gateway architecture. Routine audit. The kind of work nobody

notices until it matters.

The Transit Gateway connected fourteen VPCs across three AWS accounts: production, staging, and a shared services account that handled DNS resolution and centralized logging. Standard pattern. Well-documented. David had built it eighteen months ago and it hadn't changed since.

Except one of the route table entries was new.

A static route, pointing a CIDR block he didn't recognize to an attachment he hadn't created. The attachment connected to a VPC in the shared services account — a VPC that wasn't in any architecture diagram, any Terraform state file, any change management ticket.

David pulled the CloudTrail logs for the Transit Gateway. The `CreateTransitGatewayRoute` event was there, timestamped eleven days ago. The principal was a service-linked role. The source IP was an internal AWS address. Everything about it looked automated. Legitimate. Routine.

He almost closed the tab.

Instead, he clicked into the VPC. A single EC2 instance. `t3.micro`. Running Amazon Linux. No name tag. No purpose tag. No owner tag. The instance had been launched from a shared AMI catalog that the company maintained for quick-start development environments. The security group allowed inbound on port 443 from the Transit Gateway CIDR. Outbound to anywhere.

The instance was listening. Patient. Precise. Already inside.

David stared at the route table entry. He didn't know it yet, but he was looking at a door. And it had been open for eleven days.

Author's Note

Every supply chain attack in this book happened somewhere. The CI/CD signing key that was copied to a secret store "temporarily." The OIDC trust policy that trusted the entire org instead of one repo. The build cache shared across branches with no integrity verification. The personal access token with `repo` scope because it was easier. Different companies, different years, same patterns.

Jess is fictional. Her situation isn't. Most companies sign their builds, scan their containers, and call it supply chain security. But signing proves who signed the artifact. It doesn't prove the artifact is what you think it is. If the build itself is compromised before any of those controls run, every downstream check passes. Your signature, your scanner, your deployment pipeline all confirm: this is exactly what we built. They just can't tell you that what you built isn't what you wrote.

If you recognized the attack in this book, check your cache isolation. Check your OIDC scope. Check whether that "temporary" secret is still there.

The tools Jess rebuilt exist. Search for them.

Signing binds the signer to a digest. Not to the truth.

COSIGN SIGNATURE · binds signer digest

digest	sha256:e4d9a1c…7f02
signed by	awskms:///alias/novam
claim	image · docker-manife
result	VERIFIED ☑

Proves: this key signed this digest.
Silent on: what the digest contains.

SLSA PROVENANCE · what built the digest

digest	sha256:e4d9a1c…7f02
source	git a83f04e ☑
cache hit	Feb '26 (PR #847)
includes	_diagnostics.py

What the builder actually did, attested separately. Often absent.

Signing proves who signed. Provenance proves what was built.

Most teams only enforce the first.

Appendix

Techniques, Detections & Real-World References

MITRE ATT&CK Mapping

Technique ID Story Element

Phishing: Spearphishing Link T1566.002 Sam's PAT phished via fake GitHub security advisory

Valid Accounts: Cloud Accounts T1078.004 Phished PAT grants org-wide GitHub access

Supply Chain Compromise: Compromise Software Supply Chain T1195.002 GitHub Actions cache poisoning injects trojanized layer

Trusted Relationship T1199 OIDC trust policy trusts entire GitHub org

Build Image on Host T1612 Poisoned cache produces trojanized container image

Subvert Trust Controls: Code Signing T1553.006 Legitimate cosign + KMS signs the trojanized image

Automated Exfiltration T1020 Background thread exfiltrates on 30-minute interval

Exfiltration Over Alternative Protocol: DNS T1048.003 Patient records encoded in DNS TXT query subdomains

Data from Information Repositories T1213 Targeted ICD-10 queries against clinical database

Indicator Removal T1070 VEGA cleans infrastructure, leaves zero anomalous CloudTrail events

Real-World Parallels

SolarWinds (2020)

Build system compromise producing signed, legitimate-looking updates distributed to ~18,000 customers. The SUNBURST backdoor was inserted during the build process, not in source control. Parallels: build artifact substitution, legitimate signing of compromised software, auto-update distribution.

Codecov (2021)

Bash uploader script modified to exfiltrate CI environment variables (including secrets and tokens). Attackers modified a script in the CI/CD pipeline that customers executed during builds. Parallels: CI/CD as attack surface, credential harvesting through build tooling.

XZ Utils / liblzma (2024)

Multi-year social engineering campaign to gain maintainer trust and insert a backdoor into a widely-used compression library. The attacker ("Jia Tan") contributed legitimate patches for years before inserting the backdoor. Parallels: patience, long-game trust exploitation, targeting infrastructure that people depend on without auditing.

tj-actions/changed-files (2025)

GitHub Action compromised to exfiltrate secrets from CI/CD workflows. The action's code was modified to dump environment variables (including secrets) to workflow logs. Affected ~23,000 repositories. Parallels: GitHub Actions supply chain, mutable tags (tags can be moved to point to compromised code).

GitHub Actions Cache Poisoning (Documented 2022-2023)

Researched and documented by multiple CI/CD security researchers. The cache is shared across branches within a repository. PR branches can write cache entries that default branch builds consume. Not a vulnerability — a design decision with security implications. This is the primary attack vector in Signed & Sealed.

Key Technical Concepts

GitHub Actions Cache Scoping

- Cache entries are scoped to the repository, not the branch
- The default branch can read cache entries created by any branch
- Non-default branches can read caches from the same branch or the default branch
- Cache keys based on `hashFiles()` match regardless of which branch populated the entry
- Mitigation: include `github.ref` in cache keys to scope entries per branch

OIDC Trust Policy Scoping

- GitHub Actions OIDC tokens include a `sub` (subject) claim with format: `repo:OWNER/REPO:REF_TYPE`
- Examples: `repo:org/repo:ref:refs/heads/main`, `repo:org/repo:pull_request`, `repo:org/repo:environment:prod`
- `StringLike` with `repo:org/*` matches ALL repos, branches, PRs in the org
- `StringEquals` with a specific repo + ref is the correct scoping
- Always include the `aud` claim check: `sts.amazonaws.com`

Container Image Signing (cosign + KMS)

- `cosign sign –key awskms:///ARN` signs the image digest via KMS API call
- The signature proves: (1) the KMS key signed this digest, (2) the signing principal was authorized
- The signature does NOT prove: the image contains the correct code
- Verification: `cosign verify –key awskms:///ARN IMAGE` confirms signature validity
- Keyless signing (Fulcio + Rekor) provides transparency logging and eliminates key management

DNS Data Exfiltration

- Data encoded as subdomain labels in DNS queries (e.g., `base64data.attacker-domain.net`)
- DNS TXT record queries carry data in the query itself, not the response
- Authoritative nameserver for attacker domain captures all queries

- No TCP connection, no HTTP request, invisible to most network monitoring
- Detection: Route 53 Resolver Query Logging + alerts on TXT queries to unrecognized domains

FDA Software as a Medical Device (SaMD)

- Software that performs clinical functions (diagnosis, treatment decisions) is classified as a medical device
- Class II SaMD requires 510(k) premarket clearance
- Security patches to SaMD may require FDA notification under the 2023 cybersecurity guidance
- Emergency cybersecurity patches have an expedited review pathway through CDRH
- Tension: HIPAA requires rapid breach remediation; FDA oversight may delay software deployment

Tools Referenced

Tool	Purpose	URL
cosign	Container image signing and verification	https://github.com/sigstore/cosign
syft	SBOM generation from container images	https://github.com/anchore/syft
AWS KMS	Key management for cryptographic signing	AWS service
AWS ECR	Container image registry	AWS service
AWS ECS	Container orchestration	AWS service
Route 53 Resolver Query Logging	VPC DNS query logging	AWS service

GitHub Actions CI/CD workflow automation https://github.com/features/actions

AWS Console · Transit Gateway · tgw-0f4e…

ROUTE TABLE · tgw-rtb-087a…

CIDR	TARGET	TYPE	NOTE
10.20.0.0/16	tgw-attach-a1…	static	hr
10.21.0.0/16	tgw-attach-b2…	static	erp
10.22.0.0/16	tgw-attach-c3…	static	warehouse
0.0.0.0/0	tgw-attach-9f…	static	*17:04 — who?*

> *“I thought the hub-spoke was the isolation.”*
>
> — D. Okafor, Principal Network Engineer

• • •

Six weeks later. A Thursday.

A network engineer named David Okafor at a transit company in Atlanta was reviewing route tables for a hub-and-spoke Transit Gateway architecture. Routine audit. The kind of work nobody notices until it matters.

The Transit Gateway connected fourteen VPCs across three AWS accounts: production, staging, and a shared services account that handled DNS resolution and centralized logging. Standard pattern. Well-documented. David had built it eighteen months ago and it hadn't changed since.

Except one of the route table entries was new.

A static route, pointing a CIDR block he didn't recognize to an attachment he hadn't created. The attachment connected to a VPC in the shared services account — a VPC that wasn't in any architecture diagram, any Terraform state file, any change management ticket.

David pulled the CloudTrail logs for the Transit Gateway. The CreateTransitGatewayRoute event was there, timestamped eleven days ago. The principal was a service-linked role. The source IP was an internal AWS address. Everything about it looked automated. Legitimate. Routine.

He almost closed the tab.

Instead, he clicked into the VPC. A single EC2 instance. t3.micro. Running Amazon Linux. No name tag. No purpose tag. No owner tag. The instance had been launched from a shared AMI catalog that the company maintained for quick-start development environments. The security group allowed inbound on port 443 from the Transit Gateway CIDR. Outbound to anywhere.

The instance was listening. Patient. Precise. Already inside.

David stared at the route table entry. He didn't know it yet, but he was looking at a door. And it had been open for eleven days.

“The signature was valid.
It signed the wrong thing.”

NovaMed ships imaging software to hospitals. On Monday morning, an engineer notices a hash that doesn’t match what she reviewed. By Wednesday night, she’s tracing a CI/CD compromise through three companies.

The attacker didn’t break the signature. They changed what “the code” meant.

INSIDE

- CI/CD key theft & signer impersonation
- Build cache poisoning
- SBOM divergence & in-toto attestations
- Incident response under regulatory pause

> FDA ACK · 2026-04-22T18:14:00Z
> AFFECTED: 147 HOSPITALS · RESTORED: ALL