Table of Contents

Introduction

Security is paramount when building admin panels and back-office applications. While username and password authentication provides basic security, it's no longer sufficient in today's threat landscape. Multi-Factor Authentication (MFA), also known as Two-Factor Authentication (2FA), adds a critical extra layer of protection to your FilamentPHP applications.

In this comprehensive guide, you'll learn how to implement MFA in FilamentPHP v4 using both the native features introduced in version 4 and the popular community plugin by Stephen Jude. Whether you're building a new application or securing an existing one, this tutorial will walk you through everything you need to know.

By the end of this article, you'll be able to:

- Implement app-based TOTP authentication using Google Authenticator or similar apps
- Set up email-based one-time codes
- Configure recovery codes for account recovery
- Enable passkey authentication for passwordless login
- Force users to set up MFA before accessing your application
- Handle MFA events and customize the user experience

Why Multi-Factor Authentication Matters

Before diving into implementation details, let's understand why MFA is essential for your FilamentPHP applications:

Protection Against Credential Theft: Even if an attacker obtains a user's password through phishing, data breaches, or brute force attacks, they cannot access the account without the second factor.

Compliance Requirements: Many industries and regulations (GDPR, HIPAA, PCI DSS) require or strongly recommend MFA for applications handling sensitive data.

User Trust: Implementing MFA demonstrates your commitment to security, building trust with your users and clients.

Reduced Support Burden: While it may seem counterintuitive, MFA actually reduces unauthorized access incidents, meaning fewer account recovery requests and security-related support tickets.

According to Microsoft's security reports, MFA blocks over 99.9% of account compromise attacks. That's a compelling reason to implement it in your FilamentPHP applications.

Understanding MFA in FilamentPHP v4

FilamentPHP v4 introduced native support for multi-factor authentication, making it easier than ever to secure your admin panels. The framework provides two built-in MFA methods:

App-Based Authentication (TOTP): Users scan a QR code with an authenticator app like Google Authenticator, Authy, or Microsoft Authenticator. The app generates time-based one-time passwords (TOTP) that users enter during login.

Email-Based Authentication: Filament sends a one-time code to the user's registered email address, which they must enter to complete authentication.

Both methods integrate seamlessly with Filament's profile page, giving users a familiar interface to manage their security settings. Additionally, you can use the community plugin for more advanced features like passkey authentication and enhanced event handling.

Method 1: Native Filament MFA Implementation

Let's start with the native implementation, which gives you fine-grained control over the authentication process without external dependencies.

Setting Up App-Based Authentication (TOTP)

App-based authentication is the most secure MFA method, as it doesn't rely on email delivery and works offline. Here's how to implement it step by step.

Step 1: Create the Database Migration

First, add a column to store the encrypted authentication secret in your users table:

```
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

Schema::table('users', function (Blueprint $table) {
    $table->text('app_authentication_secret')->nullable();
});
```

Run the migration:

```
php artisan migrate
```

Step 2: Update Your User Model

Modify your `User` model to implement the necessary interfaces and handle the authentication secret securely:

```
<?php

namespace App\Models;

use Filament\Auth\MultiFactor\App\Contracts\HasAppAuthentication;
use Filament\Models\Contracts\FilamentUser;
use Illuminate\Contracts\Auth\MustVerifyEmail;
use Illuminate\Foundation\Auth\User as Authenticatable;
use Illuminate\Notifications\Notifiable;

class User extends Authenticatable implements
    FilamentUser,
    HasAppAuthentication,
    MustVerifyEmail
{
    use Notifiable;

    /**
     * The attributes that should be hidden for serialization.
     *
     * @var array<int, string>
     */
    protected $hidden = [
        'password',
        'remember_token',
        'app_authentication_secret',
    ];

    /**
     * Get the attributes that should be cast.
     *
     * @return array<string, string>
     */
    protected function casts(): array
    {
```

```
        return [
            'email_verified_at' => 'datetime',
            'password' => 'hashed',
            'app_authentication_secret' => 'encrypted',
        ];
    }

    /**
     * Get the user's app authentication secret.
     */
    public function getAppAuthenticationSecret(): ?string
    {
        return $this->app_authentication_secret;
    }

    /**
     * Save the user's app authentication secret.
     */
    public function saveAppAuthenticationSecret(?string $secret): void
    {
        $this->app_authentication_secret = $secret;
        $this->save();
    }

    /**
     * Get the holder name for the authenticator app.
     * This is displayed in the user's authenticator app.
     */
    public function getAppAuthenticationHolderName(): string
    {
        // Using email ensures unique identification
        // even for users with multiple accounts
        return $this->email;
    }

    // ... other methods
}
```

Key Points About the Implementation:

- The `app_authentication_secret` is marked as `hidden` to prevent it from being exposed in JSON responses
- The secret is encrypted using Laravel's encrypted cast for maximum security
- The `getAppAuthenticationHolderName()` method returns the email address, which appears in the authenticator app

Step 3: Enable App Authentication in Your Panel

Configure your Filament panel to enable app-based MFA:

```
<?php
```

```
namespace App\Providers\Filament;

use Filament\Auth\MultiFactor\App\AppAuthentication;
use Filament\Panel;
use Filament\PanelProvider;

class AdminPanelProvider extends PanelProvider
{
    public function panel(Panel $panel): Panel
    {
        return $panel
            ->default()
            ->id('admin')
            ->path('admin')
            ->profile()
            ->multiFactorAuthentication([
                AppAuthentication::make(),
            ])
            // ... other configuration
    }
}
```

That's it! Users can now set up TOTP authentication from their profile page. When they enable it, Filament will:

1. Generate a unique secret key
2. Display a QR code they can scan with their authenticator app
3. Require them to verify by entering a code from the app
4. Save the encrypted secret to the database

Implementing Recovery Codes

Recovery codes are essential for account recovery when users lose access to their authenticator app. Without them, locked-out users would need administrator intervention to regain access.

Step 1: Add Recovery Codes Column

Create another migration to store recovery codes:

```
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

Schema::table('users', function (Blueprint $table) {
    $table->text('app_authentication_recovery_codes')->nullable();
});
```

Step 2: Update User Model for Recovery

Implement the `HasAppAuthenticationRecovery` interface:

```php
<?php

namespace App\Models;

use Filament\Auth\MultiFactor\App\Contracts\HasAppAuthentication;
use Filament\Auth\MultiFactor\App\Contracts\HasAppAuthenticationRecovery;
use Filament\Models\Contracts\FilamentUser;
use Illuminate\Contracts\Auth\MustVerifyEmail;
use Illuminate\Foundation\Auth\User as Authenticatable;

class User extends Authenticatable implements
    FilamentUser,
    HasAppAuthentication,
    HasAppAuthenticationRecovery,
    MustVerifyEmail
{
    protected $hidden = [
        'password',
        'remember_token',
        'app_authentication_secret',
        'app_authentication_recovery_codes',
    ];

    protected function casts(): array
    {
        return [
            'email_verified_at' => 'datetime',
            'password' => 'hashed',
            'app_authentication_secret' => 'encrypted',
            'app_authentication_recovery_codes' => 'encrypted:array',
        ];
    }

    /**
     * Get the user's recovery codes.
     *
     * @return array<string>|null
     */
    public function getAppAuthenticationRecoveryCodes(): ?array
    {
        return $this->app_authentication_recovery_codes;
    }

    /**
     * Save the user's recovery codes.
     *
     * @param array<string>|null $codes
     */
    public function saveAppAuthenticationRecoveryCodes(?array $codes): void
    {
        $this->app_authentication_recovery_codes = $codes;
        $this->save();
    }
```

```
    // ... other methods
}
```

Step 3: Enable Recovery Codes in Panel Configuration

Update your panel configuration to enable recovery code generation:

```
use Filament\Auth\MultiFactor\App\AppAuthentication;
use Filament\Panel;

public function panel(Panel $panel): Panel
{
    return $panel
        ->profile()
        ->multiFactorAuthentication([
            AppAuthentication::make()
                ->recoverable()
                ->recoveryCodeCount(10) // Generate 10 recovery codes
                ->regenerableRecoveryCodes(true), // Allow users to regenerate
        ])
        // ... other configuration
}
```

Customization Options:

- `recoveryCodeCount(int $count)` : Set how many recovery codes to generate (default is 8)
- `regenerableRecoveryCodes(bool $condition)` : Allow or prevent users from regenerating codes
- Each recovery code can only be used once, enhancing security

When users enable MFA with recovery codes, Filament generates random alphanumeric codes that users should download and store securely. If they lose access to their authenticator app, they can use one of these codes to log in.

Configuring Email-Based Authentication

Email-based MFA is simpler to set up and more user-friendly, though slightly less secure than app-based authentication since it depends on email delivery.

Step 1: Add Email Authentication Column

Create a migration for the email authentication flag:

```
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

Schema::table('users', function (Blueprint $table) {
    $table->boolean('has_email_authentication')->default(false);
});
```

Step 2: Implement the Interface

Update your User model to support email authentication:

```
<?php

namespace App\Models;

use Filament\Auth\MultiFactor\Email\Contracts\HasEmailAuthentication;
use Filament\Models\Contracts\FilamentUser;
use Illuminate\Contracts\Auth\MustVerifyEmail;
use Illuminate\Foundation\Auth\User as Authenticatable;

class User extends Authenticatable implements
    FilamentUser,
    HasEmailAuthentication,
    MustVerifyEmail
{
    protected function casts(): array
    {
        return [
            'email_verified_at' => 'datetime',
            'password' => 'hashed',
            'has_email_authentication' => 'boolean',
        ];
    }

    /**
     * Check if email authentication is enabled.
     */
    public function hasEmailAuthentication(): bool
    {
        return $this->has_email_authentication;
    }

    /**
     * Toggle email authentication on or off.
     */
    public function toggleEmailAuthentication(bool $condition): void
    {
        $this->has_email_authentication = $condition;
        $this->save();
    }

    // ... other methods
}
```

Step 3: Enable Email Authentication in Panel

Add email authentication to your panel configuration:

```
use Filament\Auth\MultiFactor\Email\EmailAuthentication;
use Filament\Panel;

public function panel(Panel $panel): Panel
{
    return $panel
        ->profile()
        ->multiFactorAuthentication([
            EmailAuthentication::make()
                ->codeExpiryMinutes(5), // Code valid for 5 minutes
        ])
        // ... other configuration
}
```

How Email Authentication Works:

1. User enables email MFA in their profile
2. During login, after entering correct credentials, Filament generates a random 6-digit code
3. The code is sent to the user's email address
4. User enters the code to complete authentication
5. Codes expire after the configured time (default 4 minutes)

Combining Both Methods:

You can enable both app and email authentication simultaneously, giving users the choice:

```
use Filament\Auth\MultiFactor\App\AppAuthentication;
use Filament\Auth\MultiFactor\Email\EmailAuthentication;
use Filament\Panel;

public function panel(Panel $panel): Panel
{
    return $panel
        ->profile()
        ->multiFactorAuthentication([
            AppAuthentication::make()
                ->recoverable()
                ->recoveryCodeCount(10),
            EmailAuthentication::make()
                ->codeExpiryMinutes(5),
        ])
        // ... other configuration
}
```

Method 2: Using the Community Plugin

If you want additional features like passkey authentication, comprehensive event handling, and easier setup, Stephen Jude's Two-Factor Authentication plugin is an excellent choice.

Installing Stephen Jude's 2FA Plugin

The plugin provides a more feature-rich MFA experience with less boilerplate code.

Step 1: Install via Composer

```
composer require stephenjude/filament-two-factor-authentication
```

Step 2: Update Your User Model

Add the required trait and interface:

```
<?php

namespace App\Models;

use Filament\Models\Contracts\FilamentUser;
use Illuminate\Foundation\Auth\User as Authenticatable;
use Spatie\LaravelPasskeys\Models\Concerns\HasPasskeys;
use Stephenjude\FilamentTwoFactorAuthentication\TwoFactorAuthenticatable;

class User extends Authenticatable implements FilamentUser, HasPasskeys
{
    use TwoFactorAuthenticatable;

    // ... rest of your model
}
```

Step 3: Run the Plugin Installation

Install the necessary migrations and assets:

```
php artisan filament-two-factor-authentication:install
php artisan migrate
```

This command creates the required database tables for storing 2FA settings, recovery codes, and passkey data.

Configuring the Plugin

The plugin offers a fluent configuration API that makes setup straightforward:

```
<?php

namespace App\Providers\Filament;

use Filament\Panel;
use Filament\PanelProvider;
```

```
use Stephenjude\FilamentTwoFactorAuthentication\TwoFactorAuthenticationPlugin;

class AdminPanelProvider extends PanelProvider
{
    public function panel(Panel $panel): Panel
    {
        return $panel
            ->default()
            ->id('admin')
            ->path('admin')
            ->plugins([
                TwoFactorAuthenticationPlugin::make()
                    ->enableTwoFactorAuthentication()
                    ->enablePasskeyAuthentication()
                    ->addTwoFactorMenuItem()
                    ->forceTwoFactorSetup(),
            ])
            // ... other configuration
    }
}
```

Advanced Plugin Configuration:

For more control, you can customize each aspect:

```
use Stephenjude\FilamentTwoFactorAuthentication\TwoFactorAuthenticationPlugin;
use Stephenjude\FilamentTwoFactorAuthentication\Middleware\ForceTwoFactorSetup;
use Stephenjude\FilamentTwoFactorAuthentication\Middleware\TwoFactorChallenge;

TwoFactorAuthenticationPlugin::make()
    ->enableTwoFactorAuthentication(
        condition: true,
        challengeMiddleware: TwoFactorChallenge::class,
    )
    ->enablePasskeyAuthentication(
        condition: true,
    )
    ->forceTwoFactorSetup(
        condition: true,
        requiresPassword: true, // Require password confirmation during setup
        forceMiddleware: ForceTwoFactorSetup::class,
    )
    ->addTwoFactorMenuItem(
        condition: true,
        label: 'Security Settings',
        icon: 'heroicon-o-shield-check',
    )
```

Adding Passkey Authentication

One of the plugin's standout features is passkey support, enabling passwordless authentication using biometrics or hardware security keys.

Passkeys offer several advantages:

- **Phishing-Resistant**: Cryptographically bound to your domain
- **User-Friendly**: Face ID, Touch ID, or Windows Hello
- **Privacy-Preserving**: No shared secrets stored on servers
- **Fast**: One-tap authentication on supported devices

The plugin handles all the complexity of WebAuthn implementation behind the scenes.

Integrating into Custom Profile Pages:

If you have a custom profile page, you can add the MFA components directly:

```
<x-filament-panels::page>
    <div class="space-y-6">
        {{-- Two-Factor Authentication Settings --}}
        @livewire(\Stephenjude\FilamentTwoFactorAuthentication\Livewire\TwoFactorAuthentication::class)

        {{-- Passkey Authentication Settings --}}
        @livewire(\Stephenjude\FilamentTwoFactorAuthentication\Livewire\PasskeyAuthentication::class)
    </div>
</x-filament-panels::page>
```

Listening to Plugin Events:

The plugin dispatches comprehensive events that you can use for logging, notifications, or custom business logic:

```
<?php

namespace App\Providers;

use Illuminate\Foundation\Support\Providers\EventServiceProvider as ServiceProvider;
use Stephenjude\FilamentTwoFactorAuthentication\Events\{
    RecoveryCodeReplaced,
    RecoveryCodesGenerated,
    TwoFactorAuthenticationChallenged,
    TwoFactorAuthenticationConfirmed,
    TwoFactorAuthenticationDisabled,
    TwoFactorAuthenticationEnabled,
    TwoFactorAuthenticationFailed,
    ValidTwoFactorAuthenticationCodeProvided
};

class EventServiceProvider extends ServiceProvider
{
    protected $listen = [
```

```
        TwoFactorAuthenticationEnabled::class => [
            // Send notification to user about 2FA activation
            \App\Listeners\NotifyUserAbout2FAEnabled::class,
        ],

        TwoFactorAuthenticationFailed::class => [
            // Log failed 2FA attempts for security monitoring
            \App\Listeners\LogFailedTwoFactorAttempt::class,
        ],

        TwoFactorAuthenticationChallenged::class => [
            // Track when users are challenged for 2FA
            \App\Listeners\Track2FAChallenge::class,
        ],

        RecoveryCodesGenerated::class => [
            // Notify user to save their recovery codes
            \App\Listeners\RemindUserToSaveRecoveryCodes::class,
        ],
    ];
}
```

Example Event Listener:

```
<?php

namespace App\Listeners;

use Illuminate\Support\Facades\Log;
use Stephenjude\FilamentTwoFactorAuthentication\Events\TwoFactorAuthenticationFailed;

class LogFailedTwoFactorAttempt
{
    public function handle(TwoFactorAuthenticationFailed $event): void
    {
        Log::warning('Failed 2FA attempt', [
            'user_id' => $event->user->id,
            'email' => $event->user->email,
            'ip_address' => request()->ip(),
            'timestamp' => now(),
        ]);

        // Could also trigger alerts for multiple failed attempts
        $this->checkForBruteForce($event->user);
    }

    private function checkForBruteForce($user): void
    {
        // Implement brute force detection logic
    }
}
```

Advanced Configuration Options

Both native implementation and the plugin offer advanced customization options.

Requiring MFA for All Users

Force all users to set up MFA before accessing the application:

Native Implementation:

```
use Filament\Auth\MultiFactor\App\AppAuthentication;
use Filament\Panel;

public function panel(Panel $panel): Panel
{
    return $panel
        ->multiFactorAuthentication([
            AppAuthentication::make(),
        ], isRequired: true)
        // ... other configuration
}
```

Plugin Implementation:

```
TwoFactorAuthenticationPlugin::make()
    ->enableTwoFactorAuthentication()
    ->forceTwoFactorSetup(
        condition: true,
        requiresPassword: true,
    )
```

When MFA is required, users will be redirected to the setup page immediately after their first login and cannot access the application until they complete the setup.

Customizing Code Expiration Times

Adjust how long authentication codes remain valid:

App Authentication Code Window:

```
AppAuthentication::make()
    ->codeWindow(4) // 2-minute validity period
```

The code window defines how many 30-second intervals the TOTP code is valid. A window of 4 means the code is valid for 2 minutes (4 × 30 seconds).

Email Authentication Expiration:

```
EmailAuthentication::make()
    ->codeExpiryMinutes(3) // 3-minute expiration
```

Customizing Brand Name for Authenticator Apps

Change how your application appears in users' authenticator apps:

```
AppAuthentication::make()
    ->brandName('Cherradix Admin Panel')
```

This is particularly useful if you manage multiple Filament panels or want to clearly identify the authentication in the user's authenticator app.

Preventing Recovery Code Regeneration

For high-security environments, prevent users from regenerating recovery codes:

```
AppAuthentication::make()
    ->recoverable()
    ->recoveryCodeCount(8)
    ->regenerableRecoveryCodes(false)
```

This ensures users can't casually regenerate codes, adding an extra security layer.

Security Best Practices

Implementing MFA correctly requires attention to security details:

Store Secrets Securely

Always encrypt sensitive MFA data:

```
protected function casts(): array
{
    return [
        'app_authentication_secret' => 'encrypted',
        'app_authentication_recovery_codes' => 'encrypted:array',
    ];
}
```

Laravel's encrypted cast uses your application's encryption key, ensuring data is protected at rest.

Hide Sensitive Attributes

Prevent accidental exposure of secrets in JSON responses:

```
protected $hidden = [
    'password',
    'remember_token',
    'app_authentication_secret',
    'app_authentication_recovery_codes',
];
```

Rate Limit Authentication Attempts

Protect against brute force attacks by implementing rate limiting:

```
use Illuminate\Support\Facades\RateLimiter;

// In your event listener or middleware
RateLimiter::attempt(
    'two-factor:'.$user->id,
    $maxAttempts = 5,
    function() {
        // Allow authentication attempt
    },
    $decaySeconds = 60
);
```

Log Security Events

Monitor MFA usage for security auditing:

```
use Illuminate\Support\Facades\Log;

Log::channel('security')->info('2FA enabled', [
    'user_id' => $user->id,
    'method' => 'app_authentication',
    'ip_address' => request()->ip(),
]);
```

Educate Users

Provide clear instructions and warnings:

- Explain how to safely store recovery codes
- Warn about the consequences of losing authenticator access
- Recommend using password managers with 2FA support
- Suggest multiple backup methods (recovery codes + backup phone)

Consider Backup Authentication Methods

Implement multiple MFA options so users aren't locked out if one method fails:

```
->multiFactorAuthentication([
    AppAuthentication::make()->recoverable(),
    EmailAuthentication::make(),
])
```

Secure Your Email Delivery

If using email-based MFA, ensure your email infrastructure is secure:

- Use SPF, DKIM, and DMARC records
- Monitor email delivery rates
- Consider using a dedicated transactional email service
- Implement email security headers

Troubleshooting Common Issues

Time Synchronization Problems

TOTP codes are time-sensitive. If users report codes not working:

Problem: Server time is out of sync with NTP servers.

Solution: Ensure your server's time is synchronized:

```
# Check current time
timedatectl status

# Enable NTP synchronization
sudo timedatectl set-ntp true

# Verify synchronization
sudo systemctl status systemd-timesyncd
```

You can also increase the code window to accommodate minor time drift:

```
AppAuthentication::make()
    ->codeWindow(8) // More lenient timing
```

Recovery Codes Not Working

Problem: Recovery codes return "invalid code" error.

Solution: Verify the codes are stored correctly:

```
// In User model
public function saveAppAuthenticationRecoveryCodes(?array $codes): void
{
    // Ensure codes are stored as plain array, not JSON string
    $this->app_authentication_recovery_codes = $codes;
    $this->save();

    // Debug: Check what's being saved
    \Log::info('Saved recovery codes', [
        'user_id' => $this->id,
        'count' => count($codes ?? []),
    ]);
}
```

Email Codes Not Received

Problem: Users don't receive email authentication codes.

Solution: Check your mail configuration:

```
// Test email sending
Mail::raw('Test email from MFA', function ($message) {
    $message->to('test@example.com')
            ->subject('MFA Test');
});

// Check mail logs
tail -f storage/logs/laravel.log | grep -i mail
```

Also verify your queue is processing if using queued emails:

```
php artisan queue:work --verbose
```

Migration Conflicts

Problem: Migration fails with "column already exists" error.

Solution: Check if migrations were partially run:

```
# Check migration status
php artisan migrate:status

# Roll back specific migration
php artisan migrate:rollback --step=1

# Or drop and recreate
php artisan migrate:fresh
```

Users Locked Out Without Recovery Codes

Problem: User enabled MFA but didn't save recovery codes and lost their authenticator.

Solution: As an administrator, you can reset MFA for a user:

```
use App\Models\User;

$user = User::find($userId);
$user->app_authentication_secret = null;
$user->app_authentication_recovery_codes = null;
$user->has_email_authentication = false;
$user->save();
```

Consider creating an Artisan command for this:

```
php artisan make:command ResetUserMFA
```

```
<?php

namespace App\Console\Commands;

use App\Models\User;
use Illuminate\Console\Command;

class ResetUserMFA extends Command
{
    protected $signature = 'mfa:reset {email : The user email address}';
    protected $description = 'Reset MFA for a locked-out user';

    public function handle()
    {
        $email = $this->argument('email');
        $user = User::where('email', $email)->first();

        if (!$user) {
            $this->error("User not found: {$email}");
            return 1;
        }

        if (!$this->confirm("Reset MFA for {$user->name} ({$email})?")) {
            return 0;
        }

        $user->update([
            'app_authentication_secret' => null,
            'app_authentication_recovery_codes' => null,
            'has_email_authentication' => false,
        ]);

        $this->info("MFA successfully reset for {$email}");
```

```
        return 0;
    }
}
```

Conclusion

Implementing Multi-Factor Authentication in FilamentPHP v4 is straightforward, whether you choose the native implementation or leverage the community plugin. Both approaches provide robust security enhancements that protect your applications from unauthorized access.

Key Takeaways:

- **Native implementation** gives you fine-grained control and zero external dependencies
- **Community plugin** offers additional features like passkey authentication and comprehensive event handling
- Always implement recovery codes to prevent permanent lockouts
- Both app-based and email-based MFA significantly improve security
- Force MFA setup for high-security applications
- Monitor MFA events for security auditing and user support
- Keep your server time synchronized for TOTP reliability

Next Steps:

1. Choose between native implementation and the plugin based on your needs
2. Start with app-based authentication for maximum security
3. Add recovery codes to prevent lockouts
4. Consider enabling email authentication as a backup method
5. Test the entire flow thoroughly before deploying to production
6. Document the MFA setup process for your users
7. Monitor authentication logs for suspicious activity

Security is an ongoing process, not a one-time implementation. Regularly review your MFA configuration, keep your dependencies updated, and stay informed about emerging authentication standards.

For more Laravel and FilamentPHP tutorials, visit cherradix.dev where we share practical guides and best practices for modern PHP development.

Additional Resources:

- FilamentPHP Official Documentation
- Stephen Jude's 2FA Plugin
- OWASP Multi-Factor Authentication Guidelines
- WebAuthn Specification

Have you implemented MFA in your FilamentPHP applications? Share your experiences and tips in the comments below, and don't forget to explore more advanced FilamentPHP features on cherradix.dev!