1v0 Ship v1.0 Faster

Best Practices for Storing API Keys Securely in Laravel

Hardcoding secrets in code or committing them to Git is a fast way to leak credentials. In this guide, you'll learn **how to store API keys securely in Laravel 12** using environment variables, configuration, optional encrypted database storage with an admin UI, and (optionally) cloud secret managers. Every step includes code and explanations so beginners can implement it safely.

By the end, you'll be able to: keep secrets out of source control, load them via `config()` (not `env()`) in app code, optionally encrypt them at rest in your database with a simple settings UI, and understand when to adopt a managed secret store in production.

1 – Core Principles (Read This First)

- **Never commit secrets to Git** (including private repos). Use `.env` or secret managers.
- **Read secrets via `config()`** in application code. Avoid calling `env()` outside config files.
- **Limit exposure**: only inject the secrets you actually need in each environment.
- **Rotate regularly**, log access, and **never print secrets in logs**.

2 – Put secrets in `.env`, read them from `config/services.php`

```
# .env
MAILGUN_API_KEY=your-mailgun-key
STRIPE_SECRET=sk_live_xxx
THIRD_PARTY_MAPS_KEY=maps-xxxCode language: PHP (php)
```

Expose them to your app via configuration so you can safely call `config()` everywhere else:

```
// config/services.php
return [
    'mailgun' => [
        'key' => env('MAILGUN_API_KEY'),
    ],
    'stripe' => [
        'secret' => env('STRIPE_SECRET'),
    ],
    'maps' => [
        'key' => env('THIRD_PARTY_MAPS_KEY'),
    ],
];
Code language: PHP (php)
```

In your controllers/services, reference `config('services.mailgun.key')` (not `env()`). This keeps production fast and predictable when you run `php artisan config:cache`.

```
php artisan config:cacheCode language: Bash (bash)
```

3 – Use secrets in code (safely)

Fetch secrets via `config()` and never log or echo them:

```
// app/Services/EmailService.php
namespace App\Services;

use Illuminate\Support\Facades\Http;

class EmailService
{
    public function sendTransactional($to, $subject, $html)
    {
        $key = config('services.mailgun.key'); // do not use env()
here

        return Http::withToken($key)
            ->post('https://api.mailgun.net/v3/your-domain/messages',
[
                'to' => $to,
                'subject' => $subject,
                'html' => $html,
            ]);
    }
}Code language: PHP (php)
```

Keep secrets out of debug pages and logs: don't dump `$key`, don't include it in exceptions.

4 – Optional: Encrypt secrets at rest in your database

1v0 Ship v1.0 Faster

(with an Admin UI)

> **Why do this?** If non-technical admins need to update API keys without SSH access, store them in a small `app_settings` table *encrypted at rest*, exposed via a protected settings page. This approach complements (not replaces) `.env` for certain keys.

Create the table:

```
php artisan make:migration create_app_settings_table
```
Code language: Bash (bash)

```
// database/migrations/xxxx_xx_xx_create_app_settings_table.php
use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

return new class extends Migration {
  public function up(): void {
    Schema::create('app_settings', function (Blueprint $table) {
      $table->id();
      $table->string('key')->unique();
      $table->text('value');   // encrypted blob
      $table->timestamps();
    });
  }
  public function down(): void {
    Schema::dropIfExists('app_settings');
  }
};
```
Code language: PHP (php)

Model with transparent encryption/decryption:

```
// app/Models/AppSetting.php
namespace App\Models;
```

```
use Illuminate\Database\Eloquent\Model;

class AppSetting extends Model
{
    protected $fillable = ['key','value'];

    public function setValueAttribute($val)
    {
        $this->attributes['value'] = encrypt($val);
    }
    public function getValueAttribute($val)
    {
        return decrypt($val);
    }
}
```

Code language: PHP (php)

Routes + controller (lock behind Admin role):

```
// routes/web.php
use App\Http\Controllers\Admin\SettingsController;

Route::middleware(['auth','role:Admin'])->group(function () {
  Route::get('/admin/settings',
[SettingsController::class,'edit'])->name('admin.settings.edit');
  Route::post('/admin/settings',
[SettingsController::class,'update'])->name('admin.settings.update');
});
```

Code language: PHP (php)

```
// app/Http/Controllers/Admin/SettingsController.php
namespace App\Http\Controllers\Admin;

use App\Http\Controllers\Controller;
use App\Models\AppSetting;
use Illuminate\Http\Request;

class SettingsController extends Controller
{
```

```
    public function edit()
    {
        $mailgun =
AppSetting::firstWhere('key','mailgun_key')?->value;
        $stripe  =
AppSetting::firstWhere('key','stripe_secret')?->value;

        return view('admin.settings', compact('mailgun','stripe'));
    }

    public function update(Request $r)
    {
        $data = $r->validate([
            'mailgun_key' => 'nullable|string',
            'stripe_secret' => 'nullable|string',
        ]);

        if(isset($data['mailgun_key'])){
            AppSetting::updateOrCreate(['key' => 'mailgun_key'],
['value' => $data['mailgun_key']]);
        }
        if(isset($data['stripe_secret'])){
            AppSetting::updateOrCreate(['key' => 'stripe_secret'],
['value' => $data['stripe_secret']]);
        }

        return back()->with('status','Settings updated.');
    }
}
```

Code language: PHP (php)

Minimal UI:

```
<!-- resources/views/admin/settings.blade.php -->
@extends('layouts.app')

@section('content')
<div class="container py-4" style="max-width:680px">
  <h1 class="h5 mb-3">Secure Settings</h1>
```

```
  @if (session('status')) <div class="alert alert-success">{{
session('status') }}</div> @endif

  <form method="POST" action="{{ route('admin.settings.update') }}"
class="card card-body">
    @csrf
    <label class="form-label">Mailgun API Key</label>
    <input class="form-control mb-3" name="mailgun_key" value="{{
old('mailgun_key', $mailgun) }}" autocomplete="off">

    <label class="form-label">Stripe Secret</label>
    <input class="form-control mb-3" name="stripe_secret" value="{{
old('stripe_secret', $stripe) }}" autocomplete="off">

    <button class="btn btn-primary">Save</button>
    <p class="text-muted small mt-2 mb-0">Values are encrypted at
rest.</p>
  </form>
</div>
@endsection
```
Code language: PHP (php)

How to use these settings in code? Read them (decrypted) via the model or a small cache:

```
$mailgun =
optional(\App\Models\AppSetting::firstWhere('key','mailgun_key'))?->va
lue;
$stripeSecret =
optional(\App\Models\AppSetting::firstWhere('key','stripe_secret'))?->
value;
```
Code language: PHP (php)

Security notes: protect the route with Admin-only access, consider activity logs, and never display secrets in plain text to non-privileged users.

5 – Optional: Use a managed Secrets Manager in production (how-to)

- **AWS Secrets Manager**: store secrets by name; in your deploy pipeline/instance, fetch them and write to `.env` (or set as environment variables). Rotate via AWS console/policies.
- **GCP Secret Manager**: grant your service account access; at boot, fetch and inject into env.
- **Azure Key Vault**: grant your app's managed identity access; resolve secrets at startup.

Implementation sketch (AWS CLI during deploy):

```
# Retrieve and write to .env during CI/CD or instance boot
MAILGUN_API_KEY=$(aws secretsmanager get-secret-value --secret-id
prod/mailgun --query 'SecretString' --output text)
echo "MAILGUN_API_KEY=$MAILGUN_API_KEY" >> .env

# Then cache config for performance
php artisan config:cacheCode language: Bash (bash)
```

This keeps secrets outside the repo and centralizes rotation. Your app still reads from `config()` like normal.

6 – Common mistakes to avoid

- Calling `env()` in controllers/services. Use `config()` so cached config works.
- Leaving secrets in exception messages or debug logs.
- Committing `.env`, screenshots of keys, or curl commands with secrets to Git.
- Not rotating keys after breaches or role changes.

Wrapping Up

You learned how to store API keys securely in Laravel 12: keep them in `.env`, read them through `config()`, cache configuration in production, and optionally provide an encrypted **Admin Settings** UI for select keys. For larger teams or stricter environments, integrate a **managed secrets manager** and inject values at deploy time. Most importantly: never log secrets, never commit them, and rotate regularly.

What's Next

- How to Prevent CSRF, XSS, and SQL Injection in Laravel Apps — defense-in-depth for common web attacks.
- Securing Laravel APIs with Sanctum: Complete Guide — protect SPAs and mobile clients.
- How to Expire User Sessions Automatically in Laravel — reduce session risk.


1v0
Ship v1.0
Faster