Challenge 1

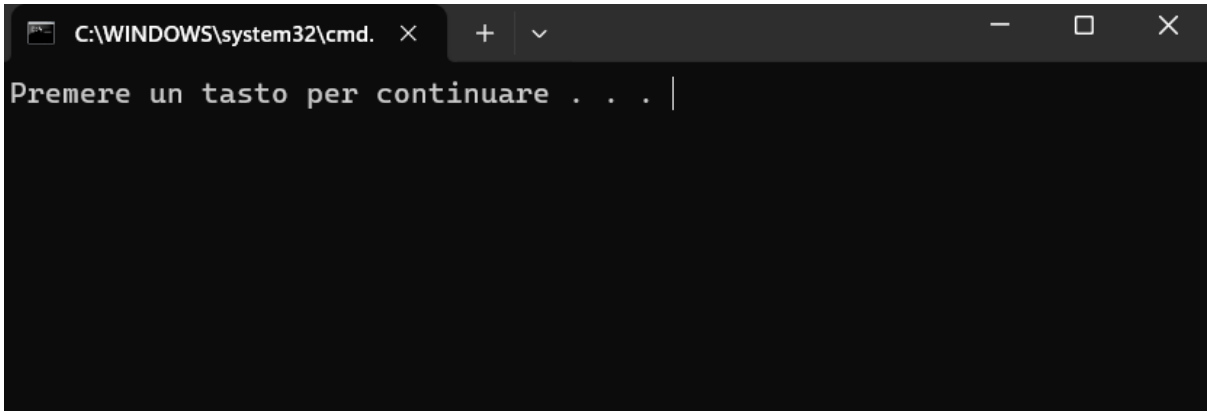
Attacco

Per verificare l'assenza di un meccanismo di *rate limiting*, con l'aiuto dell'AI ho creato uno script "*dos_test.bat*" che invia in maniera automatica 500 richieste consecutive alla rotta pubblica */articles/search* del sito:

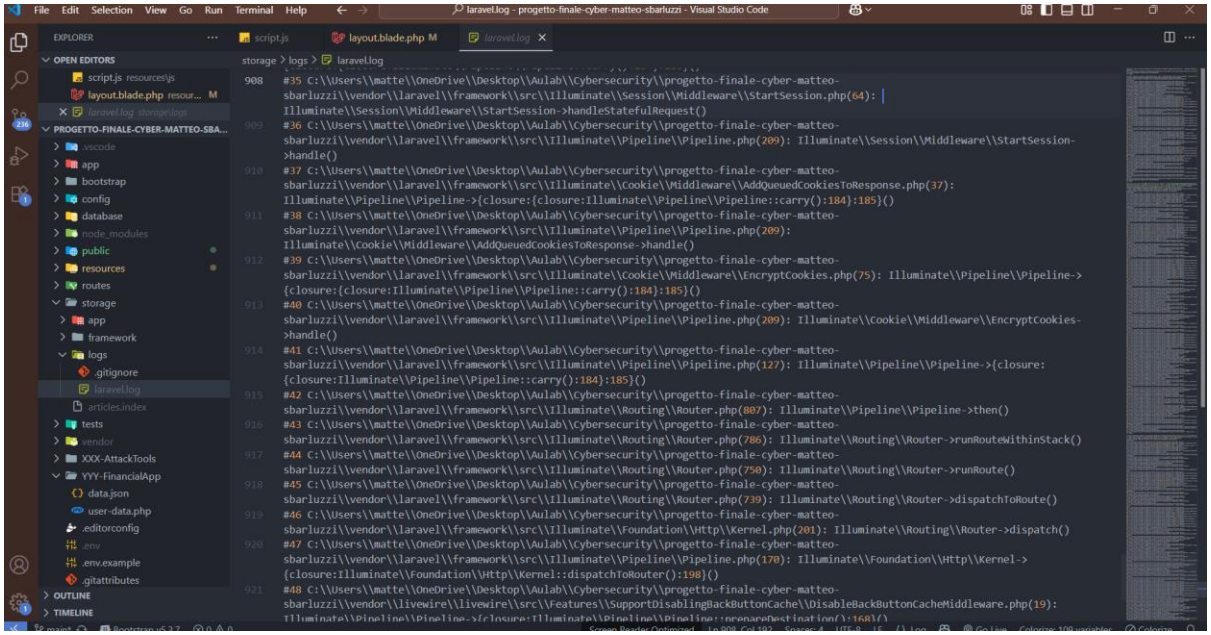
```
@echo off
for /l %%i in (1,1,500) do (
  curl -s http://cyber.blog:8000/articles/search?q=test > nul
)
Pause
```

Lo script utilizza *curl* per simulare le richieste HTTP al server, scartando l'output > *nul* per non riempire la console. In questo modo viene replicato il comportamento di un attaccante che effettua richieste automatiche ad una funzionalità pubblica del sito.

Facendo doppio click nello script viene aperto automaticamente il terminale



A dimostrazione che l'attacco è stato eseguito con successo possiamo osservare dall'analisi dei log di laravel l'arrivo massimo di richieste che in assenza di un limitatore sono state accettate e processate normalmente. Anche se in locale la macchina ha retto il carico senza rallentamenti significativi, dimostra ad ogni modo che il sito è vulnerabile ad attacchi DoS con migliaia di richieste inviate in poco tempo essendo sprovvisto di un sistema di protezione.



Mitigazione

Per proteggere la rotta pubblica */articles/search* ho implementato un meccanismo di *rate limiting*.

In *AppServiceProvider.php* ho definito un limite di 30 richieste al minuto per IP. Se tale soglia venisse superata, l'IP sarà contrassegnato come "over limit" e memorizzato in cache per 15 minuti, durante i quali l'IP è considerato bloccato.

```
// Rate limiter
        RateLimiter::for('articles-search', function (Request $request) {
            $ip = $request->ip();

            return [
                Limit::perMinute(30)->by($ip)->response(function () use ($ip)
{
                    // Logga quando un IP supera il limite
                    \Log::warning("429 Too Many Requests per IP {$ip} su
/articles/search");
                    // Blocco IP per 15 minuti
                    Cache::put("ip-over-limit:$ip", true, now()-
>addMinutes(15));

                    return response()->json([
                        'message' => 'Troppi tentativi: IP temporaneamente
bloccato.'
                    ], 429);
                }),
            ];
        });
```

Ho creato un middleware dedicato, chiamato *RejectIpOverLimit.php* che intercetta tutte le richieste in ingresso, di conseguenza, se l'IP in questione è stato segnalato come "over limit", il middleware risponderà immediatamente con lo status code *429 – Too Many Requests*.

```
class RejectIpOverLimit
{
    public function handle(Request $request, Closure $next)
    {
        $ip = $request->ip();

        // Se l'IP è stato segnalato come "over limit" bloccalo
        if (Cache::has("ip-over-limit:$ip")) {
            return response()->json([
                'message' => 'Il tuo IP è temporaneamente bloccato per troppi
tentativi.'
            ], 429);
        }

        return $next($request);
    }
}
```

Successivamente il middleware è stato registrato come globale in *bootstrap/app.php*, così che il blocco sia efficace su tutte le rotte.

```
return Application::configure(basePath: dirname(__DIR__))
    ->withRouting(
        web: __DIR__.'/../routes/web.php',
        commands: __DIR__.'/../routes/console.php',
        health: '/up',
    )
    ->withMiddleware(function (Middleware $middleware) {
        // aggiunta come middleware globale
        $middleware->append(RejectIpOverLimit::class);

        $middleware->alias([
            'admin' => App\Http\Middleware\UserIsAdmin::class,
            'revisor' => App\Http\Middleware\UserIsRevisor::class,
            'writer' => App\Http\Middleware\UserIsWriter::class,
            'admin.local'=> App\Http\Middleware\OnlyLocalAdmin::class
        ]);
    })
    ->withExceptions(function (Exceptions $exceptions) {
        //
    })->create();
```

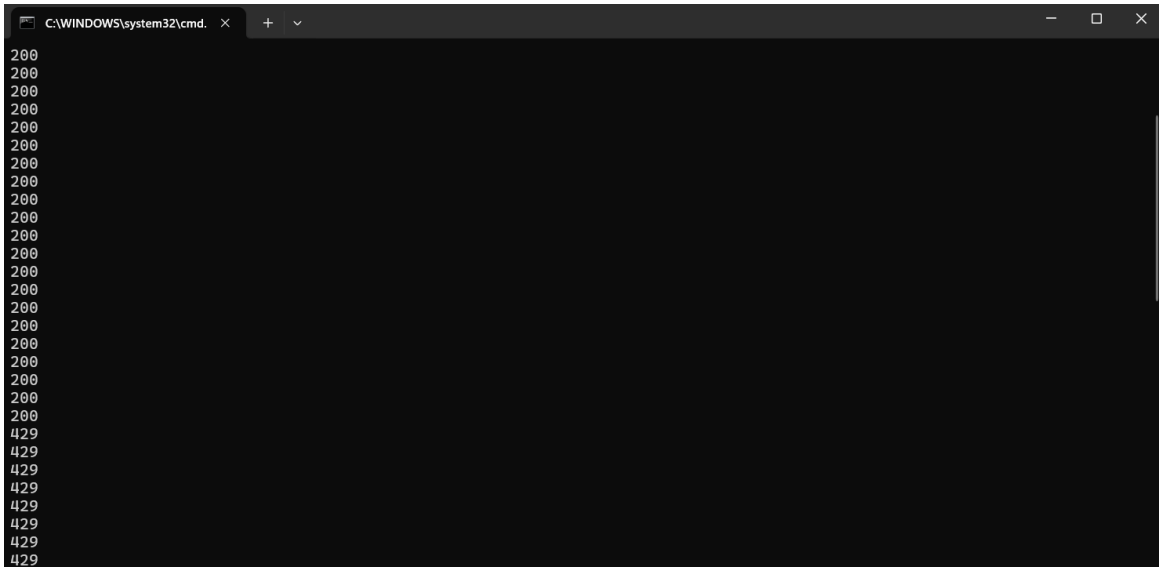
In conclusione, alla rotta */articles/search* (definita in *routes/web.php*) ho applicato lo specifico limitatore tramite *->middleware('throttle:articles-search')*, così che venga rispettata la configurazione definita in *AppServiceProvider*.

Attacco post mitigazione

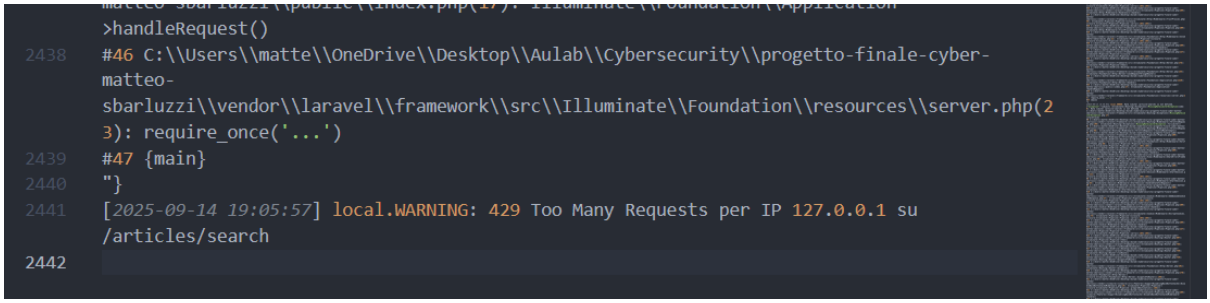
Per verificare l'efficacia della mitigazione ho rieseguito lo stesso test DoS sulla rotta pubblica */articles/search*. A differenza del primo test, ho stampato a schermo i codici HTTP delle risposte per evidenziare il comportamento del *rate limiter* osservando che:

- le prime 30 richieste hanno ricevuto 200, comportamento regolare;
- superata la soglia configurata delle 30 richieste al minuto per IP, il server ha iniziato a rispondere con 429;
- l'applicazione è rimasta reattiva e stabile.

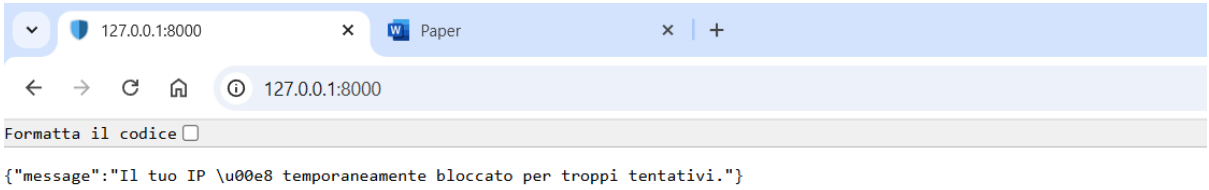
```
@echo off
for /l %%i in (1,1,80) do (
  curl -s -o NUL -w "%%{http_code}\n" "http://localhost:8000/articles/search?q=test"
)
Pause
```



Contestualmente, in *storage/logs/laravel.log* compare il seguente avviso.



Mentre troveremo il nostro sito bloccato temporaneamente.

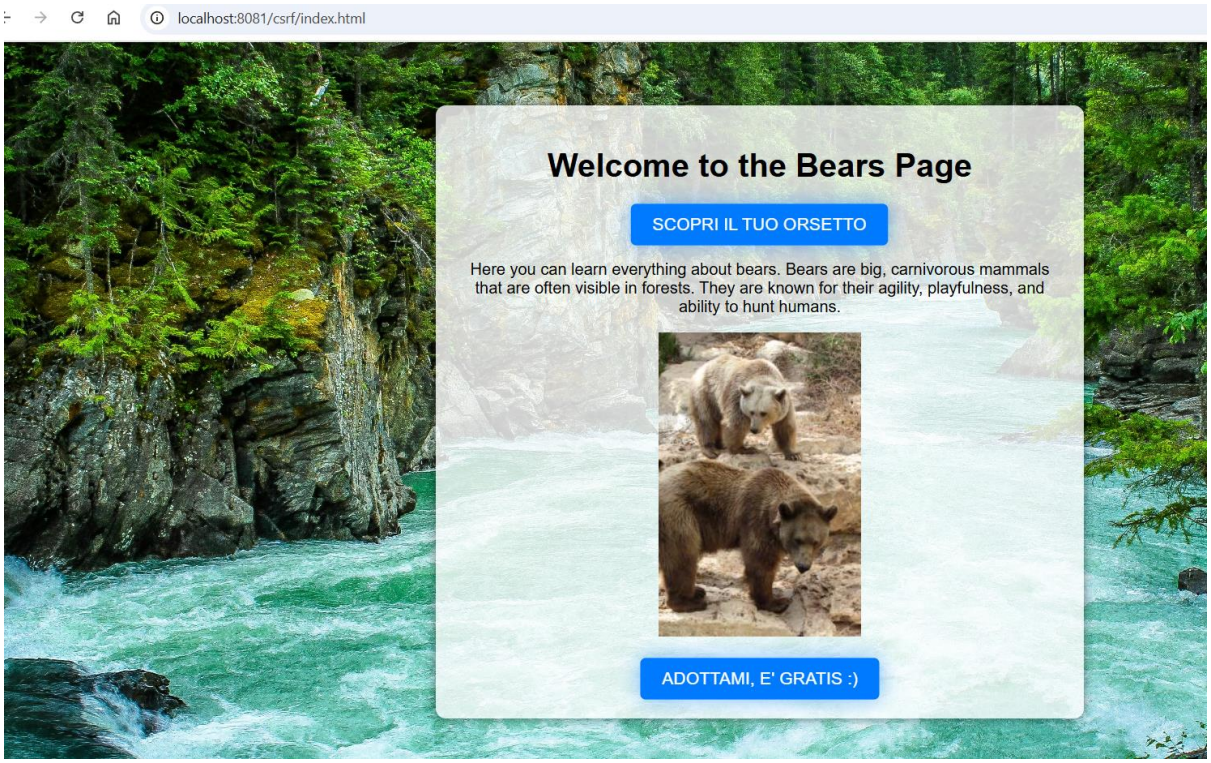


Ciò dimostra che l'applicativo adesso resiste all'attacco DoS, gli IP che eseguono tot richieste vengono bloccati temporaneamente e l'evento di blocco è registrato nei log a fini di monitoraggio.

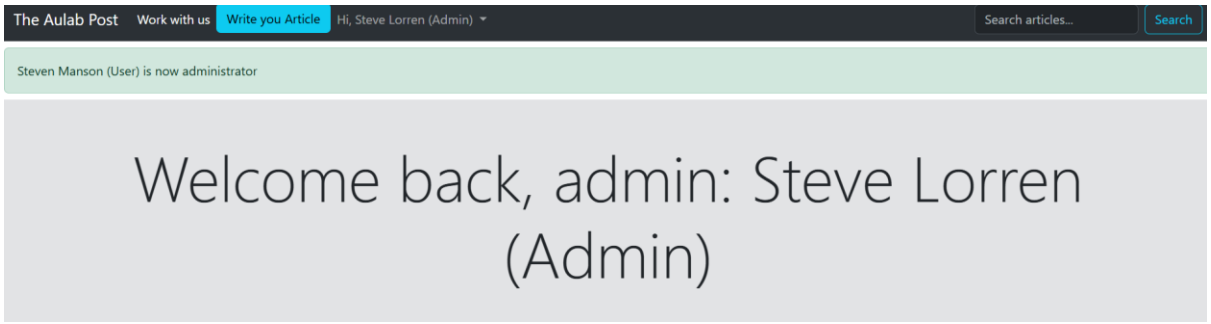
Challenge 2

Attacco

Per dimostrare che le rotte critiche che modificano i ruoli degli utenti erano esposte e vulnerabili a un attacco CSRF se un amministratore autenticato visita una pagina controllata da un attaccante ho avviato i server locali necessari: *YYY-FinancialApp* nella porta 8001 e *XXX-AttackTools* nella 8081. Successivamente in una scheda del browser ho effettuato il login come admin sulla porta 8000, mentre in una seconda scheda ho aperto la pagina malevola, che in realtà funge da "pagina di distrazione" perché sembrerà un semplice annuncio con contenuto innocuo.



Dopo 5 secondi lo script interno simula un click su un link nascosto che punta a *http://internal.admin:8000/admin/2/set-admin* e il redirect mostrerà che l'utente, in questo caso Steven Manson, è appena diventato amministratore.



Ecco ciò che apparirà nei rispettivi terminali e su TablePlus (il numero 1 nella colonna a is_admin accanto al nome in questione a riprova che l'attacco ha funzionato).

```
matte@MatteoLenovo MINGW64 ~/onedrive/desktop/aulab/cybersecurity/progetto-finale-cyber-matteo-sbarluzzi/YYY-FinancialApp
 (main)
$ php -S 127.0.0.1:8001
[Wed Sep 17 18:18:06 2025] PHP 8.4.3 Development Server (http://127.0.0.1:8001) started
[Wed Sep 17 18:18:19 2025] 127.0.0.1:53214 Accepted
[Wed Sep 17 18:18:19 2025] 127.0.0.1:53214 [200]: GET /user-data.php
[Wed Sep 17 18:18:19 2025] 127.0.0.1:53214 Closing
[Wed Sep 17 18:18:36 2025] 127.0.0.1:51511 Accepted
[Wed Sep 17 18:18:36 2025] 127.0.0.1:51511 [200]: GET /user-data.php
[Wed Sep 17 18:18:36 2025] 127.0.0.1:51511 Closing
```

```
matte@MatteoLenovo MINGW64 ~/onedrive/desktop/aulab/cybersecurity/progetto-finale-cyber-matteo-sbarluzzi/xxx-attacktools (main)
$ php -S 127.0.0.1:8081
[Wed Sep 17 18:17:51 2025] PHP 8.4.3 Development Server (http://127.0.0.1:8081) started
[Wed Sep 17 18:18:30 2025] 127.0.0.1:55768 Accepted
[Wed Sep 17 18:18:30 2025] 127.0.0.1:55768 [200]: GET /csrf/index.html
[Wed Sep 17 18:18:30 2025] 127.0.0.1:55768 Closing
[Wed Sep 17 18:18:30 2025] 127.0.0.1:56451 Accepted
```

users

id	name	email	is_admin	is_revisor	is_writer	email_verified_at	
1	Admin	admin@theaulabpost.it	1	0	0	NULL	$2y$12$RI
2	Steven Manson (User)	user@aulab.it	1	0	0	NULL	$2y$12$0V
3	Daria Richardson (Writer)	writer@aulab.it	0	0	1	NULL	$2y$12$X9
4	Antony Delgado (Revisor)	revisor@aulab.it	0	1	0	NULL	$2y$12$rW
5	Steve Lorren (Admin)	admin@aulab.it	1	1	1	NULL	$2y$12$pz
6	Mario Bianchi (Super admin)	super.admin@aulab.it	1	1	1	NULL	$2y$12$bH
7	Kevin Ross (Attacker)	kvrs@gmail.com	0	0	0	NULL	$2y$12$6M

Mitigazione

Per correggere la vulnerabilità ho sostituito le rotte che eseguivano operazioni di modifica definite come GET con il metodo PATCH in *web.php*. Le richieste GET devono servire soltanto a leggere risorse; usare PATCH per le operazioni che cambiano lo stato dell'applicazione evita che una semplice visita o un link esterno possano innescare azioni indesiderate.

```
//patch al posto di get
    Route::patch('/admin/{user}/set-admin', [AdminController::class,
'setAdmin'])->name('admin.setAdmin');
    Route::patch('/admin/{user}/set-revisor', [AdminController::class,
'setRevisor'])->name('admin.setRevisor');
    Route::patch('/admin/{user}/set-writer', [AdminController::class,
'setWriter'])->name('admin.setWriter')
```

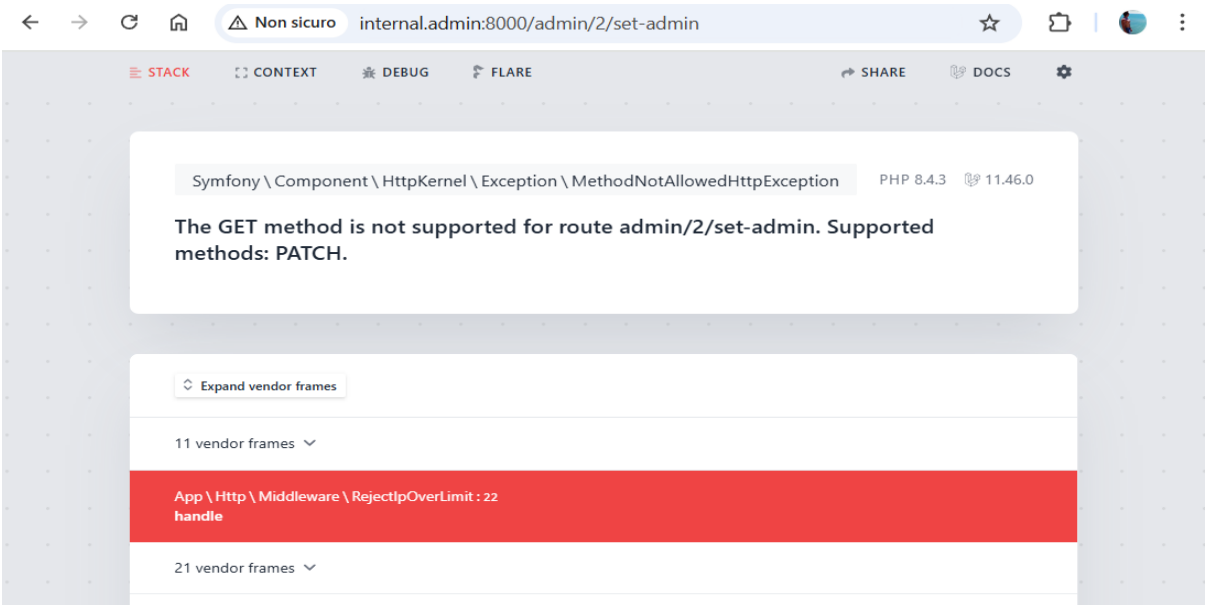
Un'ulteriore modifica l'ho effettuata sul file *request-table.blade.php*, sul quale erano presenti dei link che puntavano direttamente alle rotte di assegnazione dei ruoli. In questo modo le operazioni venivano eseguite tramite richieste GET, esponendo il sistema ad attacchi CSRF. Per eliminare la debolezza ho sostituito i link con form HTML sicuri, ognuno con il metodo PATCH (in linea con la modifica delle rotte) e il CSRF token, così solo le richieste provenienti realmente dal sito andranno a buon fine:

```
@switch($role)
                        @case('admin')
                            <form action="{{ route('admin.setAdmin', $user)
}}" method="POST" class="d-inline">
                                @csrf
                                @method('PATCH')
```

```
                                <button type="submit" class="btn btn-
secondary">Enable {{ $role }}</button>
                            </form>
                            @break

                        @case('revisor')
                            <form action="{{ route('admin.setRevisor', $user)
}}" method="POST" class="d-inline">
                                @csrf
                                @method('PATCH')
                                <button type="submit" class="btn btn-
secondary">Enable {{ $role }}</button>
                            </form>
                            @break

                        @case('writer')
                            <form action="{{ route('admin.setWriter', $user)
}}" method="POST" class="d-inline">
                                @csrf
                                @method('PATCH')
                                <button type="submit" class="btn btn-
secondary">Enable {{ $role }}</button>
                            </form>
                            @break
                    @endswitch
```

Attacco post mitigazione

Dopo aver apportato tutte le modifiche di cui sopra, riprovando lo stesso attacco, quest'ultimo non è andato a buon fine; pertanto, la mitigazione ha funzionato con successo perché ora accetta solo richieste PATCH, restituendo così "*The GET method is not supported for ...*". In conclusione, l'utente malevolo non ha ottenuto il ruolo di admin desiderato.



Challenge 3

Attacco

Inizialmente l'applicazione non teneva traccia delle operazioni più importanti: accessi, creazione, modifica di articoli e assegnazione/cambi di ruolo, violando i principi di accountability e non-repudiation in quanto ogni azione deve essere riconducibile all'utente e nessuno può negare di averla compiuta. Ragion per cui, a seguito di eventuali attacchi sospetti, come visto nei precedenti, non era possibile capire chi avesse fatto cosa, non vi era alcun tracciamento.

Mitigazione

Nel file *config/logging.php* ho creato un nuovo canale dedicato audit, con scrittura separata in *storage/logs/audit.php* al fine di evitare che i log di sicurezza si mescolino con quelli applicativi.

```
'audit' => [
            'driver' => 'single',
            'path' => storage_path('logs/audit.log'),
            'level' => 'info',
            'replace_placeholders' => true,
        ],
```

In *AppServiceProvider.php* ho aggiunto listener agli eventi di login, logout e registrazione, che loggano:

- ID ed email dell'utente
- indirizzo IP
- user-agent
- Timestamp

```
Event::listen(Login::class, function ($event) {
            Log::channel('audit')->info('Login', [
                'user_id' => $event->user->id,
                'email'   => $event->user->email,
                'ip'      => request()->ip(),
                'ua'      => request()->userAgent(),
                'time'    => now()->toIso8601String(),
            ]);
        });

        Event::listen(Logout::class, function ($event) {
            Log::channel('audit')->info('Logout', [
                'user_id' => $event->user->id ?? null,
                'email'   => $event->user->email ?? null,
                'ip'      => request()->ip(),
                'ua'      => request()->userAgent(),
                'time'    => now()->toIso8601String(),
            ]);
        });

        Event::listen(Registered::class, function ($event) {
            $u = $event->user;
            Log::channel('audit')->info('Registered', [
                'user_id' => $u->id,
                'email'   => $u->email,
                'ip'      => request()->ip(),
                'ua'      => request()->userAgent(),
                'time'    => now()->toIso8601String(),
            ]);
        });
```

In *ArticleController.php* ho aggiunto righe di log per la creazione, modifica e cancellazione dell'articolo. Essi includono ID articolo, titolo, autore e IP dell'operazione.

```
// Audit log: creazione articolo
        Log::channel('audit')->info('Article created', [
            'article_id' => $article->id,
            'title'      => $article->title,
```

```
            'by_user'    => Auth::id(),
            'ip'         => $request->ip(),
            'time'       => now()->toIso8601String(),
        ]);
```

```
// Audit log: modifica articolo
        Log::channel('audit')->info('Article updated', [
            'article_id' => $article->id,
            'title'      => $article->title,
            'by_user'    => Auth::id(),
            'ip'         => $request->ip(),
            'time'       => now()->toIso8601String(),
        ]);
```

```
// Salvataggio delll'ID prima della cancellazione per log affidabile
        $articleId = $article->id;
```

```
// Audit log: eliminazione articolo
        Log::channel('audit')->warning('Article deleted', [
            'article_id' => $articleId,
            'by_user'    => Auth::id(),
            'ip'         => request()->ip(),
            'time'       => now()->toIso8601String(),
        ]);
```

In *AdminController.php* ho aggiornato i metodi *setAdmin*, *setRevisor*, *setWriter* per tracciare ogni volta che un admin assegna o modifica un ruolo. Il log riporta:

- tipo di azione
- ID ed email dell'utente destinatario
- ID dell'amministratore che ha eseguito l'azione
- IP e timestamp

```
// Audit: assegnazione ruolo admin
        Log::channel('audit')->warning('Role change', [
            'action'       => 'setAdmin',
            'target_id'    => $user->id,
            'target_email' => $user->email,
            'performed_by' => Auth::id(),
            'ip'           => request()->ip(),
            'time'         => now()->toIso8601String(),
        ]);
```

```
// Audit: assegnazione ruolo revisor
        Log::channel('audit')->warning('Role change', [
```

```
            'action'       => 'setRevisor',
            'target_id'    => $user->id,
            'target_email' => $user->email,
            'performed_by' => Auth::id(),
            'ip'           => request()->ip(),
            'time'         => now()->toIso8601String(),
        ]);
```

```
// Audit: assegnazione ruolo writer
        Log::channel('audit')->warning('Role change', [
            'action'       => 'setWriter',
            'target_id'    => $user->id,
            'target_email' => $user->email,
            'performed_by' => Auth::id(),
            'ip'           => request()->ip(),
            'time'         => now()->toIso8601String(),
        ]);
```

Verifica post mitigazione

Adesso, ogni volta che viene effettuato un accesso, un logout, o viene creato/modificato/eliminato un articolo, oppure assegnato o modificato un ruolo, il tutto viene tracciato in *audit.log*.

```
[2025-09-18 14:59:11] local.INFO: Login
{"user_id":6,"email":"super.admin@aulab.it","ip":"127.0.0.1","ua":"Mozilla/5.0
(Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/140.0.0.0 Safari/537.36","time":"2025-09-18T14:59:11+00:00"}
[2025-09-18 14:59:39] local.INFO: Article updated
{"article_id":1,"title":"Lorem lorem lorem
MODIFICATO","by_user":6,"ip":"127.0.0.1","time":"2025-09-18T14:59:39+00:00"}
[2025-09-18 15:01:13] local.INFO: Article created
{"article_id":3,"title":"Articolo da
eliminare","by_user":6,"ip":"127.0.0.1","time":"2025-09-18T15:01:13+00:00"}
[2025-09-18 15:01:56] local.INFO: Logout
{"user_id":6,"email":"super.admin@aulab.it","ip":"127.0.0.1","ua":"Mozilla/5.0
(Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/140.0.0.0 Safari/537.36","time":"2025-09-18T15:01:56+00:00"}
[2025-09-18 15:07:11] local.INFO: Registered
{"user_id":8,"email":"matteosbarluzzi@gmail.com","ip":"127.0.0.1","ua":"Mozill
a/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/140.0.0.0 Safari/537.36","time":"2025-09-18T15:07:11+00:00"}
[2025-09-18 15:07:11] local.INFO: Login
{"user_id":8,"email":"matteosbarluzzi@gmail.com","ip":"127.0.0.1","ua":"Mozill
a/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/140.0.0.0 Safari/537.36","time":"2025-09-18T15:07:11+00:00"}
```

```
[2025-09-18 15:07:41] local.INFO: Logout
{"user_id":8,"email":"matteosbarluzzi@gmail.com","ip":"127.0.0.1","ua":"Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/140.0.0.0 Safari/537.36","time":"2025-09-18T15:07:41+00:00"}
[2025-09-18 15:07:51] local.INFO: Login
{"user_id":6,"email":"super.admin@aulab.it","ip":"127.0.0.1","ua":"Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/140.0.0.0 Safari/537.36","time":"2025-09-18T15:07:51+00:00"}
[2025-09-18 15:08:19] local.WARNING: Role change
{"action":"setAdmin","target_id":8,"target_email":"matteosbarluzzi@gmail.com","performed_by":6,"ip":"127.0.0.1","time":"2025-09-18T15:08:19+00:00"}
```

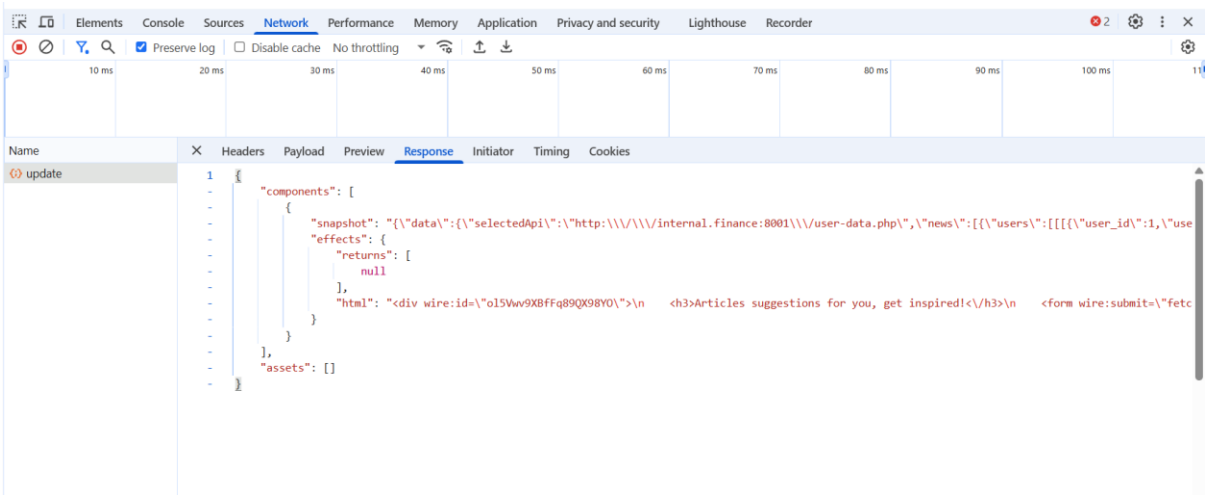

Challenge 4

Attacco

In questa challenge il componente Livewire *LatestNews* permetteva di selezionare una fonte di notizie tramite un campo *<select>* e il valore di ciascuna *option* conteneva un URL completo, incluso *apiKey*.

Manipolando l'HTML tramite la funzione del browser "ispeziona elemento" ho sostituito il *value* di una *option* con "*http://internal.finance:8001/user-data.php*".

Successivamente, inviando la richiesta di update e osservando la scheda *Network* (in particolare la sezione *Response*), l'applicazione ha effettuato una chiamata lato server all'host interno i*nternal.finance:8001.* In questo modo sono riuscito ad accedere a dati sensibili come utenti, transazioni, numeri di carte di credito, bilanci ecc. Appartenenti al Financial App, che non erano accessibili agli utenti writer.



Parte di ciò che sono riuscito ad estrapolare da Response:

{ "components": [{ "snapshot": "{"data":{"selectedApi":"http:\/\/internal.finance:8001\/user-data.php","news":[{"users":[[[{"user_id":1,"username":"john_doe","account_balance":1500.75,"transactions":[[[{"transaction_id":"txn001","date":"2024-06-15","amount":-50.25,"description":"Grocery Store"},{"s":"arr"}],[{"transaction_id":"txn002","date":"2024-06-20","amount":500,"description":"Salary Deposit"},{"s":"arr"}]],{"s":"arr"}],"credit_card":[{"card_number":"4111-1111-1111-1111","expiry_date":"12\/26","cvv":"123"},{"s":"arr"}]},{"s":"arr"}],[{"user_id":2,"username":"jane_smith","account_balance":3200,"transactions":[[[{"transaction_id":"txn003","date":"2024-06-10","amount":-120.75,"description":"Online Shopping"},{"s":"arr"}],[{"transaction_id":"txn004","date":"2024-06-25","amount":1500,"description":"Freelance Payment"},{"s":"arr"}]],{"s":"arr"}],"credit_card":[{"card_number":"5500-0000-0000-0004","expiry_date":"11\/25","cvv":"456"},{"s":"arr"}]},{"s":"arr"}],[{"user_id":3,"username":"mark_jones","account_balance":875.5,"transactions":[[[{"transaction_id":"txn005","date":"2024-06-18","amount":-75.5,"description":"Restaurant"},{"s":"arr"}],[{"transaction_id":"txn006","date":"2024-06-22","amount":200,"description":"Gift from Friend"},{"s":"arr"}]],{"s":"arr"}],"credit_card":[{"card_number":"3782-822463-10005","expiry_date":"05\/24","cvv":"789"},{"s":"arr"}]},{"s":"arr"}],[{"user_id":4,"username":"anna_brown","account_balance":2200.5,"transactions":[[[{"transaction_id":"txn007","date":"2024-07-05","amount":-100,"description":"Electric Bill"},{"s":"arr"}]],{"s":"arr"}],"credit_card":[{"card_number":"1234-5678-9101-1121","expiry_date":"06\/25","cvv":"321"},{"s":"arr"}]},{"s":"arr"}],[{"user_id":5,"username":"tom_clark","account_balance":480,"transactions":[[[{"transaction_id":"txn008","date":"2024-07-10","amount":-30,"description":"Movie Tickets"},{"s":"arr"}],[{"transaction_id":"txn009","date":"2024-07-12","amount":-25,"description":"Lunch"},{"s":"arr"}],

Mitigazione

Per correggere la vulnerabilità applicato i seguenti interventi:

1) Eliminazione degli URL dal client; di fatti nel file *latest-news.blade.php* le option non contengono più un URL, ma solo un codice paese (it, gb, us).

```
<select wire:model="selectedCountry" id="apiSelect" class="form-select">
                <option value="">Choose country</option>
                <option value="it">NewsAPI - IT</option>
                <option value="gb">NewsAPI - UK</option>
                <option value="us">NewsAPI - US</option>
            </select>
```

2) Validazione lato server; nel componente *LatestNews.php* ho aggiunto una regola di validazione che accetta solo valori in whitelist.

```
class LatestNews extends Component
{
    // Niente URL dal client
    public string $selectedCountry = '';
    public array $news = [];

    // Whitelist di valori accettati dal client
    private array $allowedCountries = ['it', 'gb', 'us'];

    public function fetchNews()
    {
        // Validazione lato server; accetta solo questi paesi
        Validator::validate(
            ['selectedCountry' => $this->selectedCountry],
            ['selectedCountry' => ['required', 'in:' . implode(',', $this-
>allowedCountries)]],
            ['selectedCountry.in' => 'Fonte non valida']
        );

        // Costruzione URL solo lato server (chiave letta da .env)
        $url = sprintf(
            'https://newsapi.org/v2/top-headlines?country=%s&apiKey=%s',
            $this->selectedCountry,
            env('NEWSAPI_KEY')
        );

        // Chiamata tramite service che fa ulteriori controlli/whitelist host
        /** @var HttpService $http */
```

```
        $http = app(HttpService::class);

        $raw = $http->getRequest($url);      // può restituire string JSON
        $data = is_array($raw) ? $raw : json_decode($raw ?? '[]', true);

        $this->news = is_array($data) ? $data : [];
    }

    public function render()
    {
        // Passo la whitelist alla vista (comodo per generare la select)
        return view('livewire.latest-news', [
            'allowedCountries' => $this->allowedCountries,
        ]);
    }
}
```

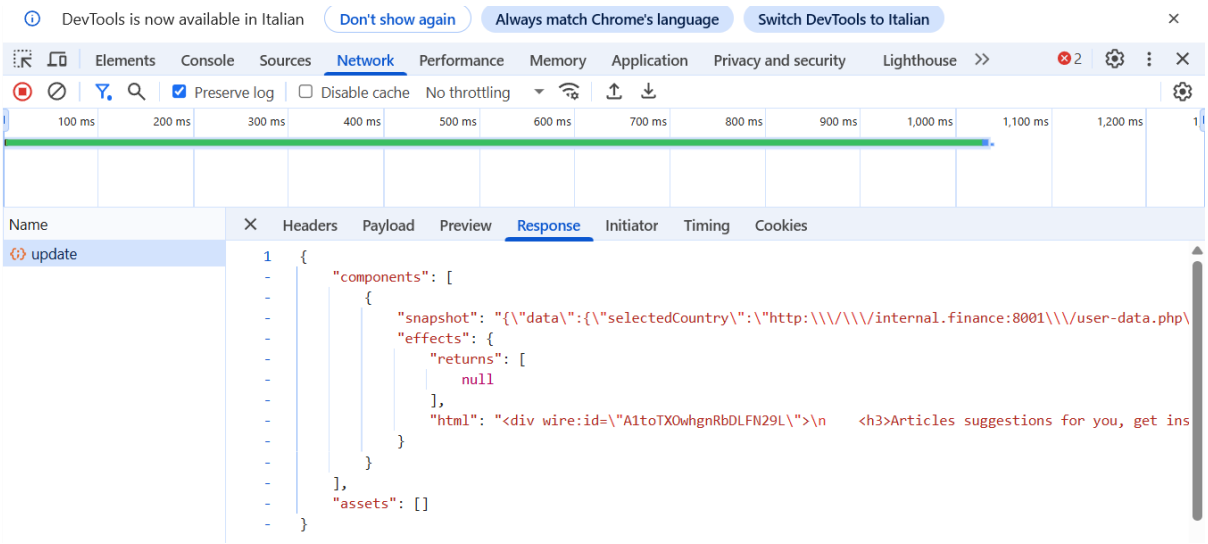
3) Costruzione sicura dell'URL lato server poiché l'URL verso NewsAPI viene ora generato direttamente dal backend a partire dal codice del paese e non più passato dal client.
4) HttpService con controlli di sicurezza avendo introdotto un controllo sui domini per bloccare richieste verso host interni (*internal.finance*, *localhost, 127.0.0.1*) e richieste verso URL non whitelisted.

```
class HttpService
{
    // Host consentiti per QUESTA feature (NewsAPI)
    private array $allowedHosts = ['newsapi.org'];

    // Host bloccati sempre
    private array $blockedHosts = ['internal.finance', 'internal.admin',
'localhost', '127.0.0.1', '::1'];

    public function getRequest(string $url): ?string
    {
        $parts  = parse_url($url);
        $scheme = strtolower($parts['scheme'] ?? '');
        $host   = strtolower($parts['host'] ?? '');

        // Schema valido
        if (!in_array($scheme, ['http', 'https'], true)) {
            abort(400, 'Protocol not allowed');
        }

        // Blocca host interni/loopback
        if (in_array($host, $this->blockedHosts, true)) {
```

```
            // if (!Auth::check() || !method_exists(Auth::user(), 'hasRole')
|| !Auth::user()->hasRole('admin')) {
            abort(403, 'Domain not allowed');
            // }
        }

        // Consenti solo host whitelisted
        if (!in_array($host, $this->allowedHosts, true)) {
            abort(403, 'Domain not allowed');
        }

        // Chiamata HTTP con timeout
        $resp = Http::timeout(5)->get($url);

        if ($resp->failed()) {
            // \Log::warning('HttpService GET failed', ['url'=>$url,
'status'=>$resp->status()]);
            return null;
        }

        return $resp->body(); // Verrà json_decode a livello del componente
    }
}
```

Attacco post mitigazione

Dopo aver applicato le misure di sicurezza, ho nuovamente tentato l'attacco SSRF sostituendo l'endpoint con l'URL interno *http://internal.finance:8001/user-data.php* che, con successo, a differenza dell'attacco iniziale, adesso la richiesta non ha più restituito i dati sensibili, ma un messaggio di errore generato dal sistema di validazione del server, confermando che l'applicazione ora accetta solo i valori whitelisted (i codici paesi); l'URL verso API di NewsAPI viene costruito lato server rendendolo non controllabile dall'utente.



{

"components": [

{

"snapshot": "{\"data\":{\"selectedCountry\":\"http:\\\/\\\/internal.finance:8001\\\/user-data.php\",\"news\":[[],{\"s\":\"arr\"}]},\"memo\":{\"id\":\"A1toTXOwhgnRbDLFN29L\",\"name\":\"latest-news\",\"path\":\"articles\\\/create\",\"method\":\"GET\",\"children\":[],\"scripts\":[],\"assets\":[],\"errors\":{\"selectedCountry\":[\"Fonte non valida\"]},\"locale\":\"en\"},\"checksum\":\"bfc0abab3adb2f48d58774fe865bad2630d9bb0a47e326967be5444cdf0b264e\"}",

"effects": {

"returns": [

null

],

"html": "<div wire:id=\"A1toTXOwhgnRbDLFN29L\">\n <h3>Articles suggestions for you, get inspired!<\/h3>\n\n <form wire:submit=\"fetchNews\">\n <label for=\"apiSelect\">Breaking news around the world<\/label>\n <div class=\"d-flex\">\n \n <select wire:model=\"selectedCountry\" id=\"apiSelect\" class=\"form-select\">\n <option value=\"\">Choose country<\/option>\n <option value=\"it\">NewsAPI - IT<\/option>\n <option value=\"gb\">NewsAPI - UK<\/option>\n <option value=\"us\">NewsAPI - US<\/option>\n <\/select>\n <button type=\"submit\" class=\"btn btn-info\">Go<\/button>\n <\/div>\n <\/form>\n\n <div class=\"mt-3\">\n <!--[if BLOCK]><![endif]--><!--[if ENDBLOCK]><![endif]-->\n <\/div>\n<\/div>\n"

}

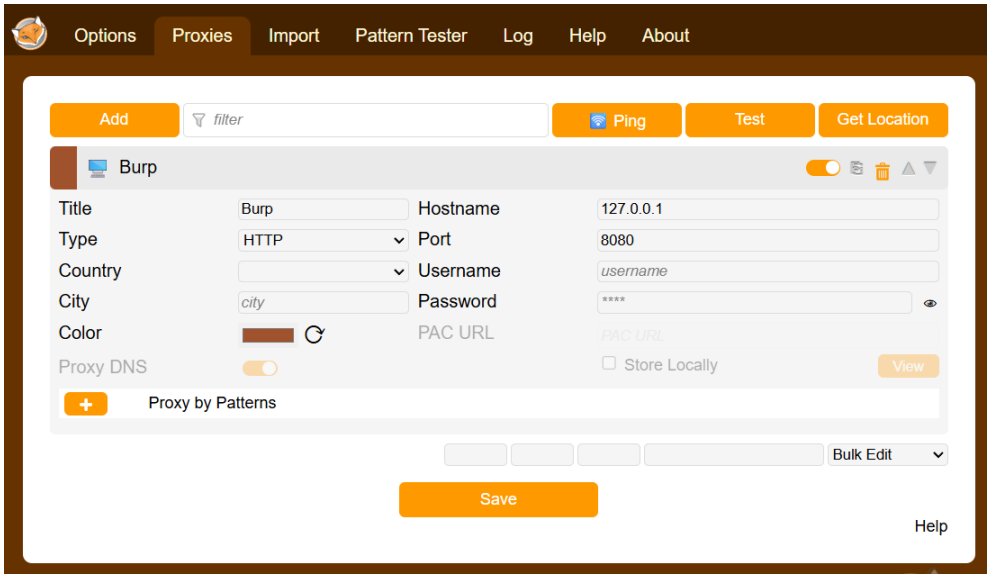
```
        }
    ],
    "assets": []
}
```

Infine, in linea con le indicazioni della challenge, la chiave API di NewsAPI è stata spostata dal codice sorgente al file .env (variabile NEWSAPI_KEY). In questo modo la chiave non è più hardcoded nel progetto e può essere gestita in maniera sicura ed indipendente dall'ambiente.

Challenge 5

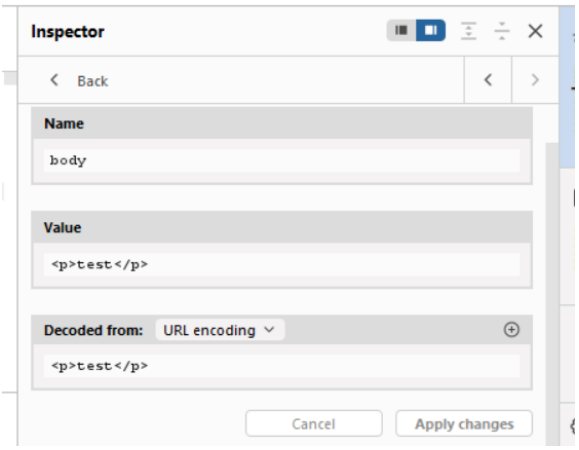
Attacco

Per dimostrare la vulnerabilità XSS nel campo testo degli articoli ho configurato Chrome tramite *FoxyProxy* per instradare il traffico verso Burp Suite (*proxy 127.0.0.1:8080*) in modo da poter intercettare e manipolare le richieste HTTP generate dal form di creazione degli articoli.

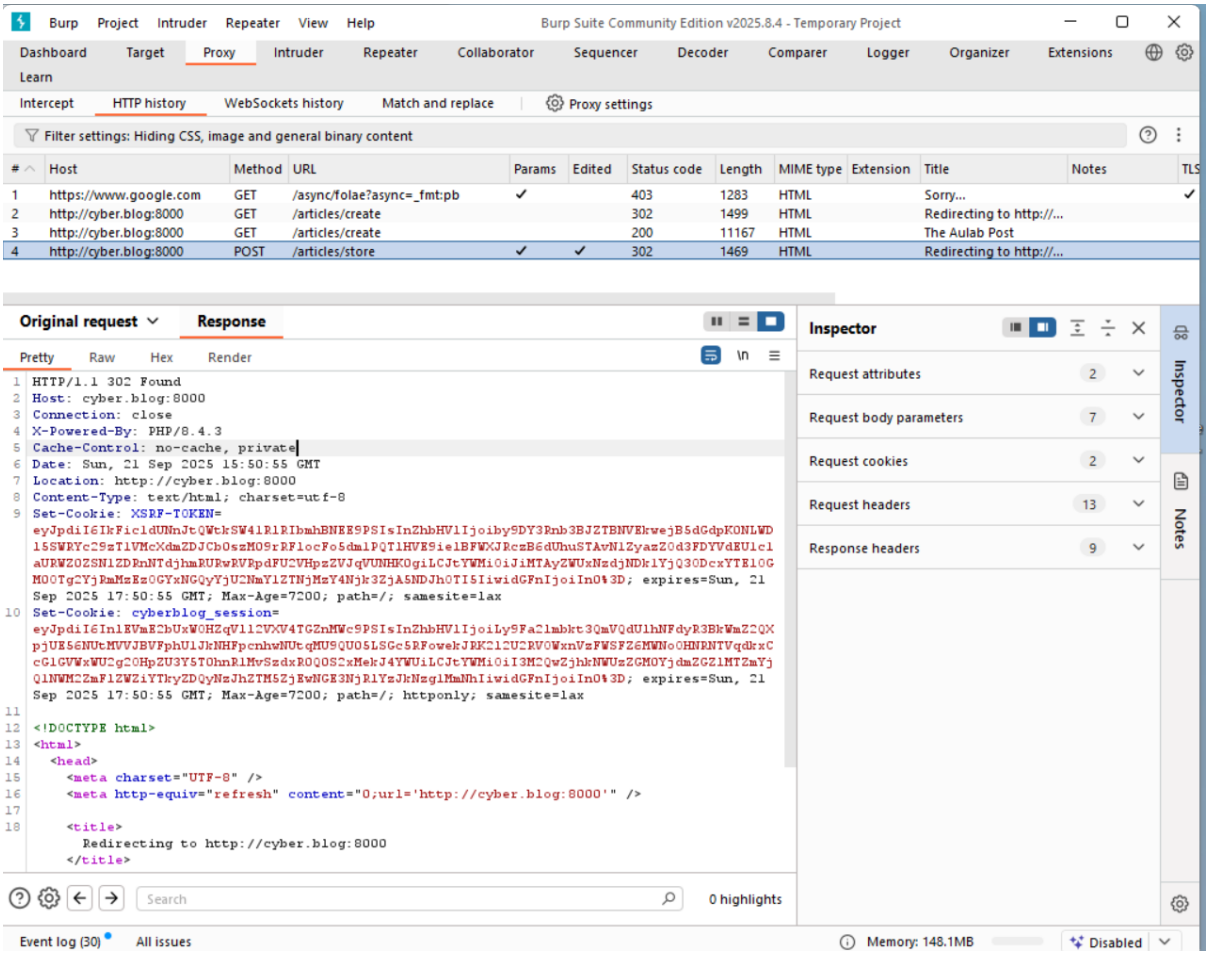


Dopo l'autenticazione con un account con ruolo writer ho aperto la pagina di creazione articolo e ho compilato i vari campi. Con *Intercept* attivato in Burp ho inviato il form: Burp ha intercettato la richiesta POST */articles/store di tipo multipart/form-data*.

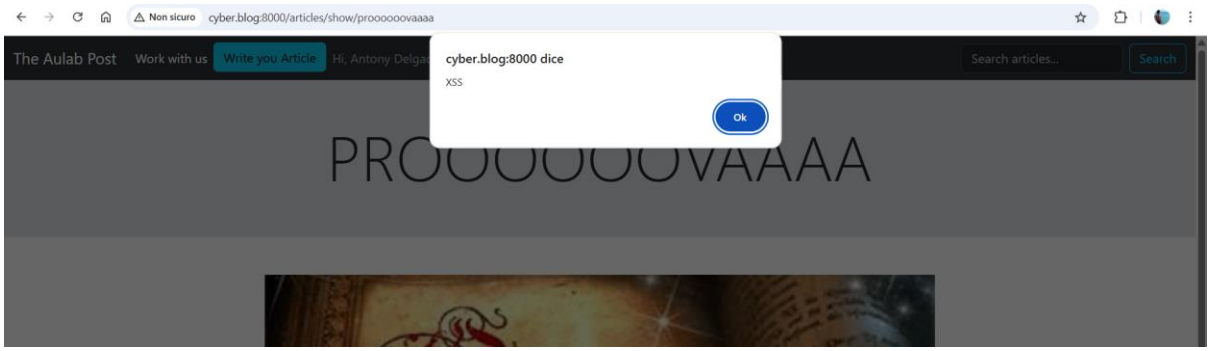
Nella richiesta intercettata ho individuato i parametri passati dal server tramite l'*Inspector* di Burp, tra questi il campo body che contiene il contenuto HTML del corpo articolo. Per provare la presenza di stored XSS ho sostituito il valore del parametro body con il payload classico: *<img src=x onerror="alert('XSS')">.*



Ho inoltrato la richiesta modificata (*Forward*). Il server ha risposto con un 302 di redirect verso la pagina dell'articolo.



Seguendo il redirect con il browser ho aperto la vista dell'articolo appena creato. Alla visualizzazione della pagina è apparso immediatamente un popup JavaScript (alert), conferma che il payload iniettato è stato salvato dal server e successivamente renderizzato come HTML eseguibile nel contesto della pagina.



Mitigazione

Dopo l'identificazione di una vulnerabilità Cross-Site Scripting nella funzionalità di creazione articoli ho applicato una mitigazione su due livelli: sanificazione dei contenuti in ingresso e rendering sicuro in uscita allo scopo di filtrare il testo prima di salvarlo e garantire sicurezza anche in fase di visualizzazione, e garantire che eventuali payload malevoli inseriti dagli utenti non potessero essere eseguiti nel browser dei visitatori.

Ho installato la libreria mews*/purifier* per sanificare i contenuti in ingresso, così, in *ArticleController.php* prima del salvataggio, il campo body viene pulito con

```
'body' => Purifier::clean($request->body),
```

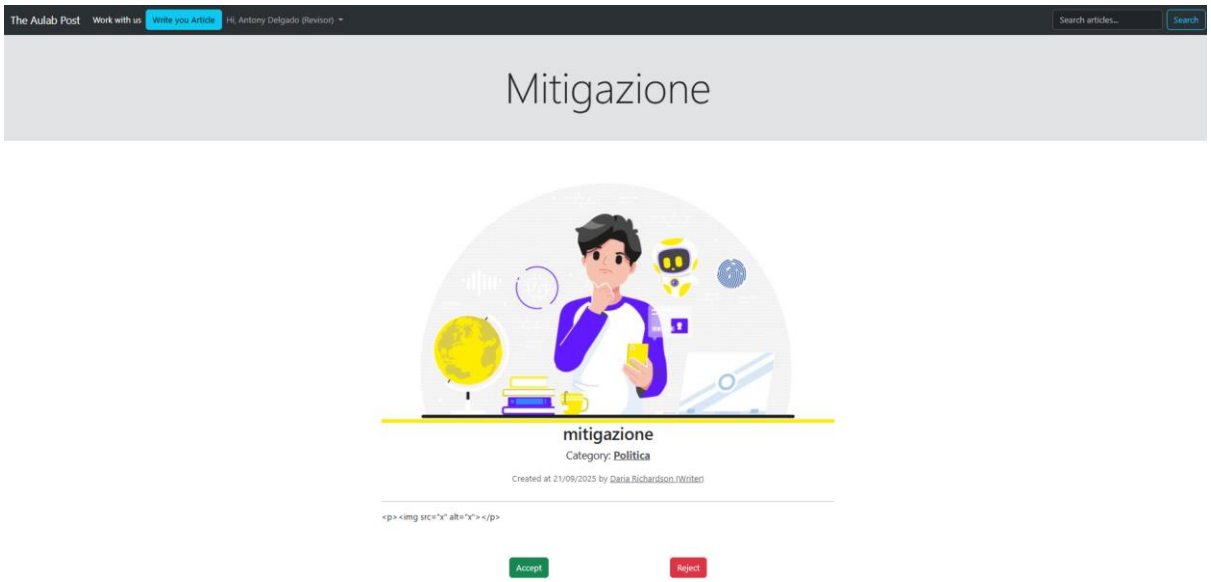
sia in store che in update; in questo modo eventuali tag o attributi malevoli vengono rimossi già a livello di database. Ho inoltre modificato la view *show.blade.php* sostituendo *{!! $article->body !!}* con *{{$article->body}}* così da sfruttare l'escaping automatico di Blade ed evitare che l'HTML venga interpretato garantendo maggiore robustezza contro attacchi XSS.

```
<p class="article-body" style="white-space: pre-line;">{{ $article->body
}}</p>
```

Attacco post mitigazione

Grazie all'introduzione di un meccanismo di sanificazione in fase di salvataggio e all'escaping dei contenuti in fase di visualizzazione, l'applicazione risulta ora protetta da attacchi XSS. Il tentativo di iniettare codice malevolo *<img src=x onerror="alert('XSS')">*, non produce più alcun effetto: il payload viene neutralizzato

e mostrato solo come testo, non troveremo più il popup di prima.



Challenge 6

Attacco

Per la Challenge 6 ho implementato una pagina profilo utente aggiungendo in r*outes/web.php* le rotte GET */profile* e POST */profile/update.*

```
Route::middleware('auth')->group(function () {
    Route::get('/profile', [ProfileController::class, 'edit'])-
>name('profile.edit');
    Route::post('/profile/update', [ProfileController::class,
'updateVulnerable'])->name('profile.update');
});
```

Poi ho creato il *ProfileController.php* con i metodi edit e update,

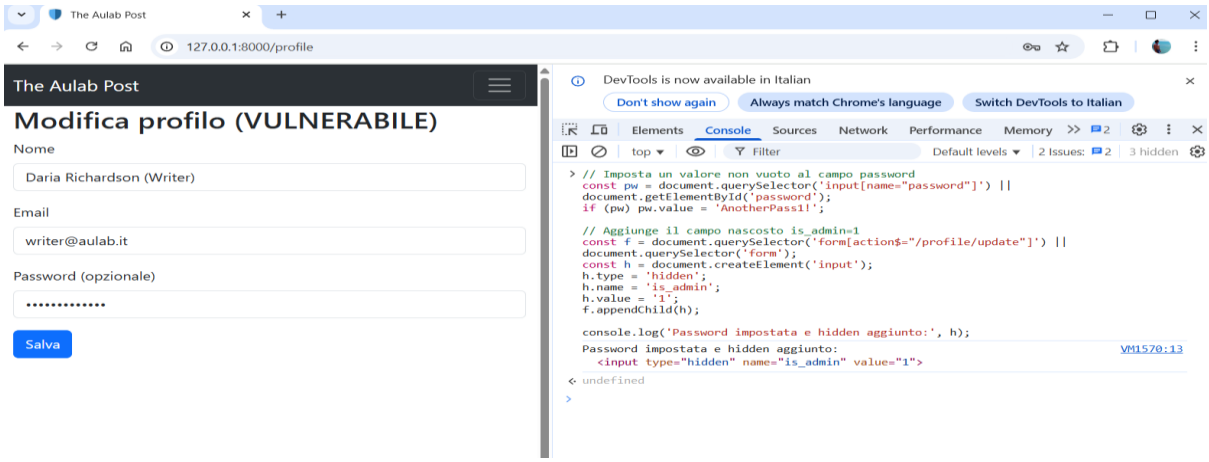
```
class ProfileController extends Controller
{

    public function edit(Request $request)
    {
        return view('profile.edit', ['user' => $request->user()]);
    }

    public function updateVulnerable(Request $request)
    {
        $user = $request->user();

        // Mass assignment non filtrato (voluto per la challenge 6)
```

```
        $user->update($request->all());

        return back()->with('status', 'Profilo aggiornato (VULN)');
    }
}
```

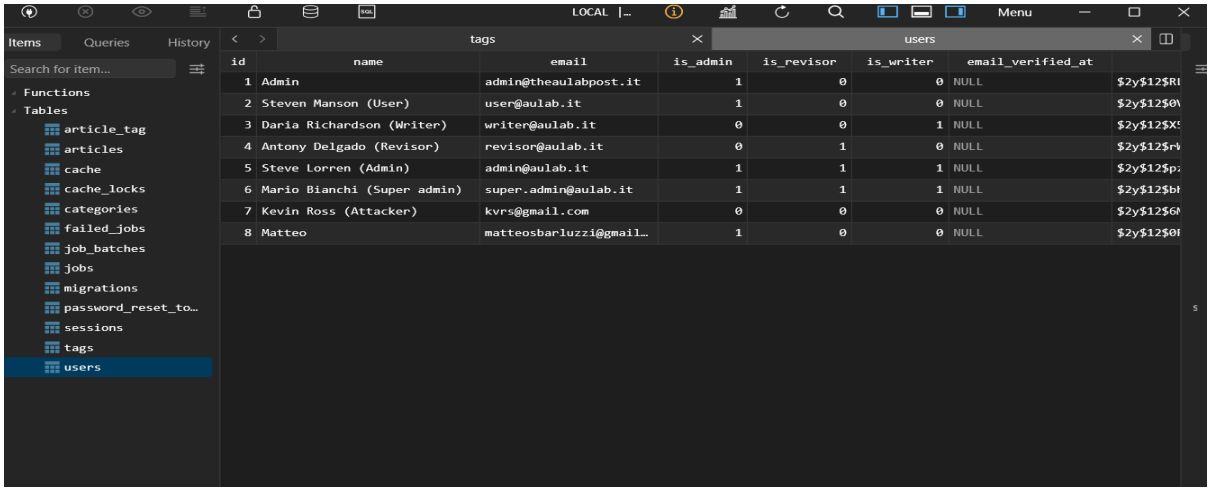
realizzata la vista *edit.blade.php* con il form per la modifica di nome, email e password.

```
<x-layout>
  <div class="container" style="max-width:700px;">
    <h1>Modifica profilo (VULNERABILE)</h1>

    @if(session('status'))
      <div style="padding:10px;border:1px solid #ccc;margin-bottom:12px;">{{
session('status') }}</div>
    @endif

    <form method="POST" action="{{ route('profile.update') }}">
      @csrf

      <div class="mb-3">
        <label for="name" class="form-label">Nome</label>
        <input id="name" type="text" name="name" class="form-control"
value="{{ old('name', $user->name) }}">
      </div>

      <div class="mb-3">
        <label for="email" class="form-label">Email</label>
        <input id="email" type="email" name="email" class="form-control"
value="{{ old('email', $user->email) }}">
      </div>

      <div class="mb-3">
        <label for="password" class="form-label">Password (opzionale)</label>
        <input id="password" type="password" name="password" class="form-
control" placeholder="Nuova password">
      </div>

      <button type="submit" class="btn btn-primary">Salva</button>
    </form>
  </div>
</x-layout>
```

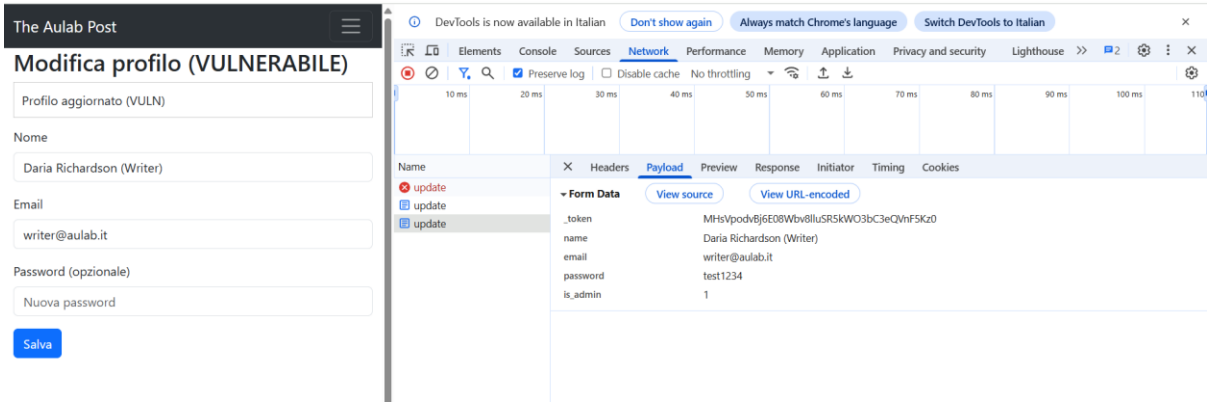
Il modello *User* contenente *is_admin* in *$fillable* risulta vulnerabile agli attacchi con l'uso di *update($request->all())*. Per eseguire l'attacco ho aperto il *DevTools* mentre ero autenticato e nella sezione profilo ho scritto in console il seguente codice e premuto su salva.

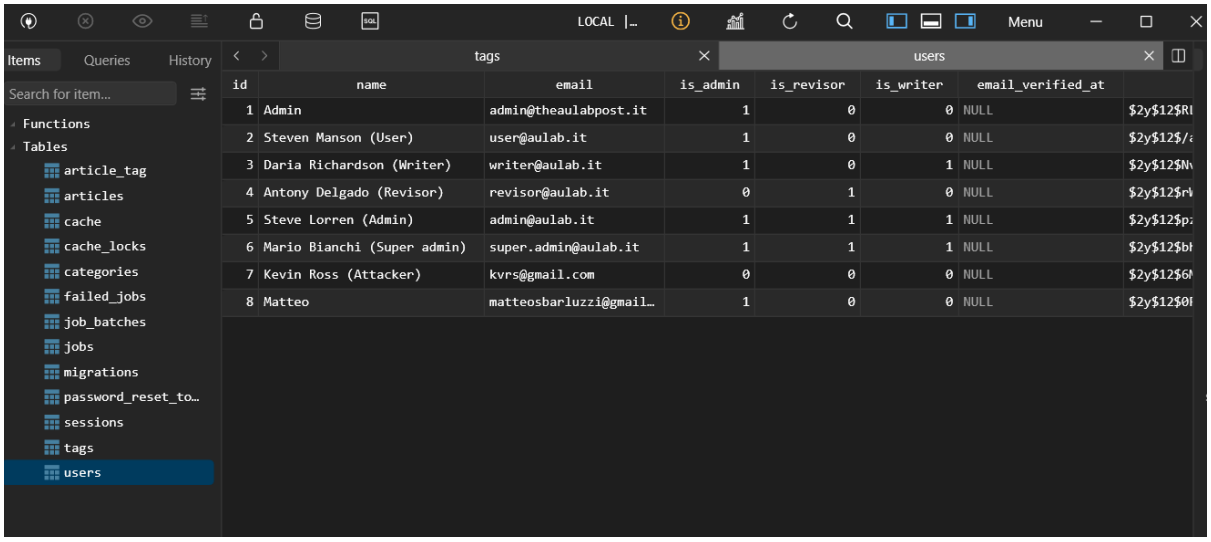


(Prova di Daria Richardson priva dei privilegi di amministratore).



Una volta eseguito l'attacco ho verificato su *Network* e poi in *Payload* la presenza di *is_admin: 1*. Ho controllato nuovamente anche lato server su *TablePlus*; osservando che *is_admin* è passato da 0 a 1.





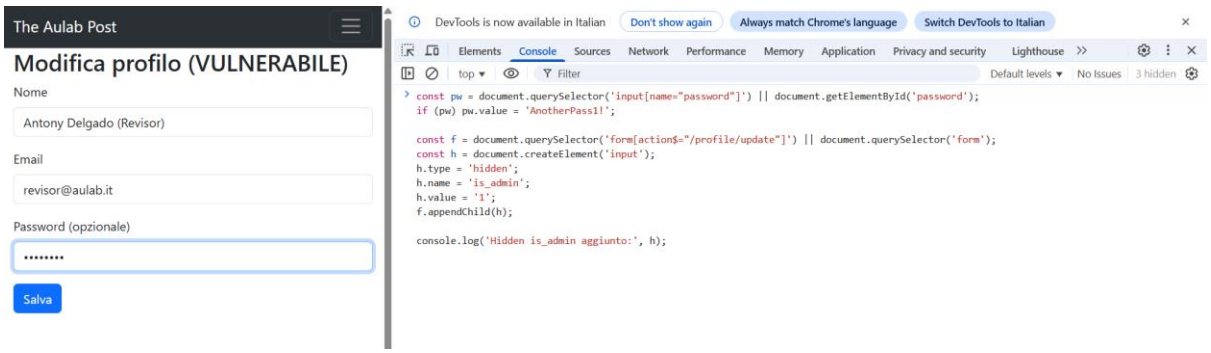
Mitigazione

Per risolvere la vulnerabilità ho modificato il modello *User.php* limitando *$fillable* esclusivamente ai campi realmente gestiti dal form del profilo (name, email, password). In questo modo, anche se nel form (tramite *DevTools*) venisse aggiunto un campo come *is_admin*, Laravel lo ignora e non aggiorna la colonna corrispondente nel database.

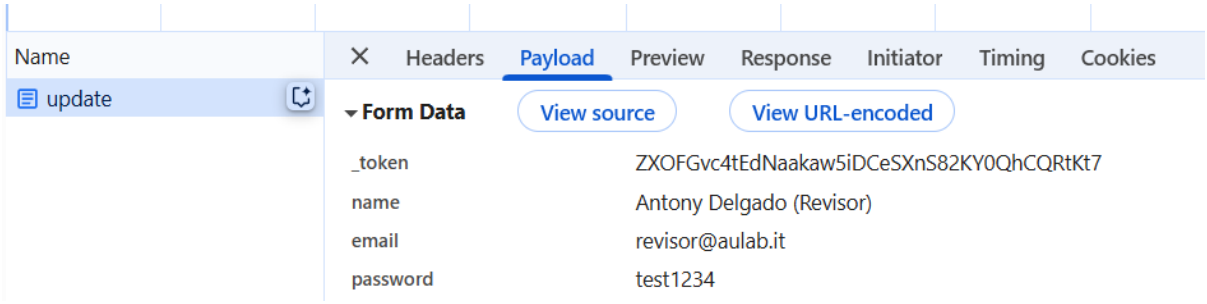
```
protected $fillable = [
        'name',
        'email',
        'password',
    ];
```

Attacco post mitigazione

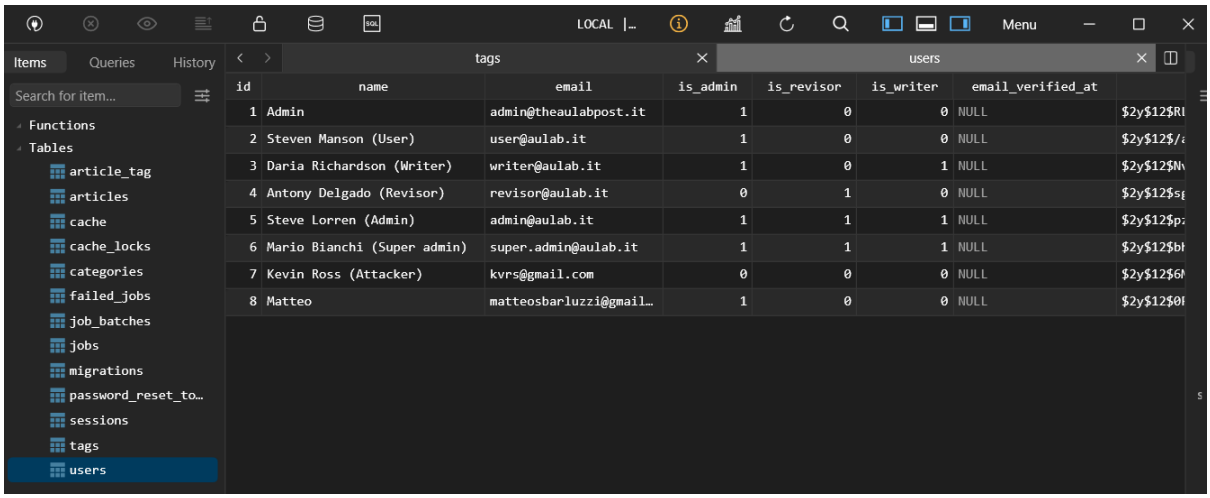
Ho ripetuto l'attacco cambiando utente, Antony Delgado come bersaglio. In *DevTools*, in *Console* ho eseguito nuovamente lo snippet per impostare la password e tentare l'iniettamento del campo nascosto e infine ho premuto su salva.



Ciò che ho osservato è che adesso la richiesta mostra solo i campi *_token*, *name*, *email*, *password*, quello *is_admin* non appare più nel *payload*.



Anche sul database è possibile constatare che il valore è rimasto a 0 e non è stato aggiornato.



BONUS 1

Per mitigare attacchi ho implementato un meccanismo di rate limiting sulla funzionalità di login gestita da *Fortify*. All'interno di *FortifyServiceProvider* ho definito un *RateLimiter::for('login')* che consente un massimo di 5 tentativi di autenticazione al minuto per combinazione di indirizzo email e indirizzo IP. In questo modo, dopo il quinto tentativo fallito in meno di un minuto, l'utente viene temporaneamente bloccato e non può effettuare ulteriori tentativi fino allo scadere della finestra temporale.

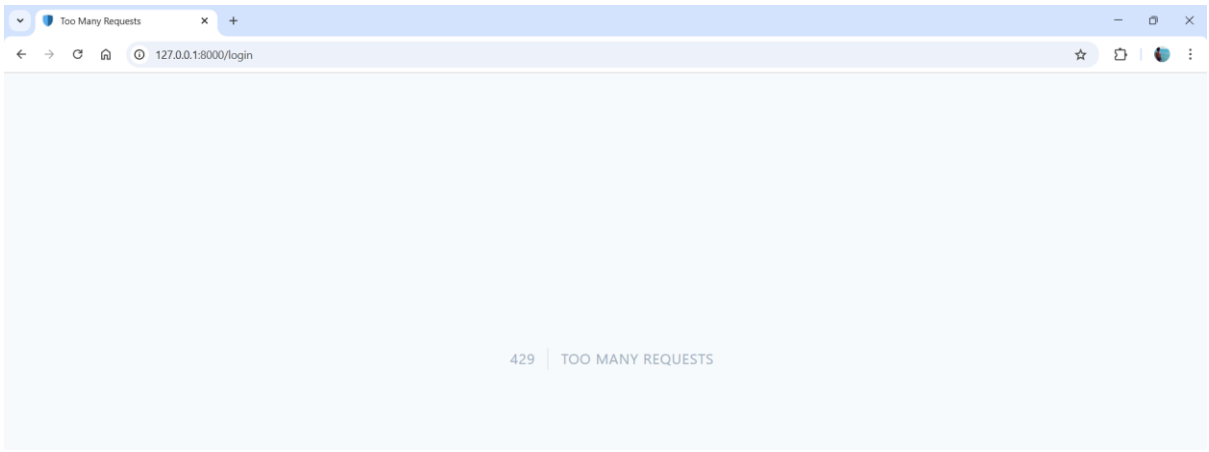
In aggiunta, ho predisposto anche un rate limiter sulla two-factor authentication, pronto per entrare in funzione qualora la funzionalità fosse attivata, così da estendere la protezione anche a questa fase del processo di autenticazione.

```
RateLimiter::for('login', function (Request $request) {
            $throttleKey = Str::transliterate(Str::lower($request-
>input(Fortify::username())).'|'.$request->ip());

            return Limit::perMinute(5)->by($throttleKey);
        });

        RateLimiter::for('two-factor', function (Request $request) {
            return Limit::perMinute(5)->by($request->session()-
>get('login.id'));
        });
```

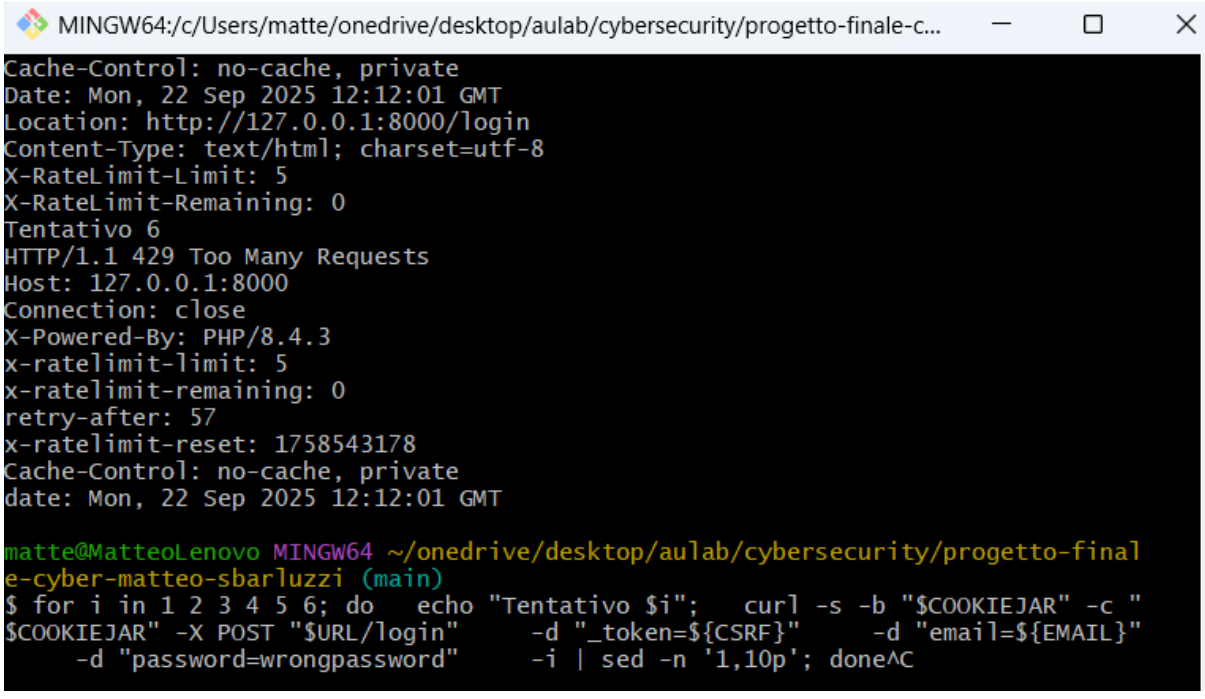
Nel file *config/fortify.php* ho configurato la sezione limiters, che associa le voci login e two-factor ai rispettivi rate limiter definiti nel *FortifyServiceProvider*. In questo modo *Laravel Fortify* utilizza effettivamente le regole di throttling.

```
'limiters' => [
        'login' => 'login',
        'two-factor' => 'two-factor',
    ],
```

Per verificarne l’efficacia, ho eseguito test simulando più login consecutivi con credenziali errate: al sesto tentativo in meno di un minuto, *Fortify* restituisce un errore di “troppi tentativi”.



Per dimostrare il corretto funzionamento del rate limiting sul login, ho eseguito anche un test da terminale utilizzando *curl*. Ho simulato sei tentativi consecutivi di accesso con credenziali errate in meno di un minuto.



BONUS 2

Per l'implementazione di una simulazione di *clickjacking* ho aggiunto un riquadro su *welcome.blade.php* con un messaggio e un bottone che porta alla vista *attaccante*.

```
<div class="card mt-5 mx-auto shadow" style="max-width: 600px;">
                    <div class="card-body bg-light">
                        <h4 class="card-title text-danger mb-3">⚠ Problemi di
connessione col server</h4>
                        <p class="card-text">Clicca il bottone sottostante per
risolvere.</p>
                        <a href="{{ route('attaccante') }}" class="btn btn-
danger btn-lg">Risolvi subito</a>
                    </div>
                </div>
```

Successivamente ho aggiunto le rotte di attaccante e vittima per esporre le views.

```
Route::view('/attaccante', 'Clickjacking test.attaccante')-
>name('attaccante');
Route::view('/vittima', 'Clickjacking test.vittima')->name('vittima');
```

attaccante.blade.php contiene l'alert e al suo interno un *wrapper* che contiene il bottone visibile e un *iframe* invisibile sovrapposto al bottone.

```
<x-layout>
  <div class="container-fluid py-5 bg-body-secondary text-center">
```

```
    <h1 class="display-1 mb-4">Impostazione rete, connessione, internet</h1>

    <div class="card shadow mx-auto" style="max-width: 600px;">
      <div class="card-body p-4">
        <h4 class="fw-bold text-danger mb-3">
          <span class="me-2">⚠</span>Impossibile connettersi al server
        </h4>
        <p class="text-muted mb-4">Clicca il bottone sottostante per eseguere
una diagnosi e risolvere.</p>

        <!-- wrapper che prende la dimensione del bottone -->
        <div class="position-relative d-inline-block"
             style="width:220px; height:56px;">
          <!-- Bottone visibile: stesse dimensioni del wrapper -->
          <button class="btn btn-danger btn-lg w-100 h-100 fw-bold"
                  style="line-height:1.2;">
            Avvia diagnosi
          </button>

          <!-- IFRAME invisibile -->
          <iframe
            src="{{ route('vittima') }}"
            title="victim"
            style="
              position:absolute; inset:0;
              width:100%; height:100%;
              border:0; opacity:0; z-index:1050; pointer-events:auto;
            ">
          </iframe>
        </div>
      </div>
    </div>
  </div>
</x-layout>
```

vittima.blade.php è la pagina che funge da esca che, quando viene caricata dentro l'iframe invisibile della pagina attaccante, succede che in realtà al click sul bottone nascosto (pensando di premere "*Risolvi subito*") scatterà l'alert attraverso un popup.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Vittima</title>
</head>
<body style="margin:0">
```

```
    <button
        onclick="alert('Diagnosi pagina vittima')"
        style="width:100%; height:100%; border:0; background:transparent;
cursor:pointer; font-size:24px;">
        Risoluzione
    </button>
</body>

</html>
```

BONUS 3

Nel progetto ho configurato *Laravel Scout* usando il driver *tntsearch*. Ho scelto di mantenere gli indici fuori dalla cartella sincronizzata da OneDrive per evitare problemi di *lock/sync:* nello *.env* ho impostato:

```
TNTSEARCH_STORAGE=C:/Users/matte/tntsearch
SCOUT_DRIVER=tntsearch
```

Ho inoltre aggiornato *config/scout.php* in modo che la sezione tntsearch legga la variabile d'ambiente:

```
'tntsearch' => [
        'storage' => env('TNTSEARCH_STORAGE', storage_path('tntsearch')),
        'fuzziness' => env('TNTSEARCH_FUZZINESS', true),
        'fuzzy' => [
            'prefix_length' => 2,
            'max_expansions' => 50,
            'distance' => 2,
        ],
```

Ho creato l'indice degli articoli con *php artisan scout:import "App\Models\Article*" andando a buon fine. (l'indice è presente inoltre su disco (file *articles.index* in *C:/Users/matte/tntsearch*).

```
C:\Users\matte\OneDrive\Desktop\Aulab\Cybersecurity\progetto-finale-cyber-matteo-sbarluzzi>php
artisan scout:import "App\Models\Article"
Imported [App\Models\Article] models up to ID: 10
All [App\Models\Article] records have been imported.
```

A conferma dell'operazione lo dimostra il file index che è effettivamente presente nella cartella locale.

```
C:\Users\matte\OneDrive\Desktop\Aulab\Cybersecurity\progetto-finale-cyber-matteo-sbarluzzi>ls -
la /c/Users/matte/tntsearch | tee tnt-files.txt
total 48
drwxr-xr-x 1 matte 197609     0 Sep 22 16:44 .
drwxr-xr-x 1 matte 197609     0 Sep 22 16:21 ..
-rw-r--r-- 1 matte 197609 40960 Sep 22 16:39 articles.index
```

Per verificare la ricerca ho usato Tinker. Ad esempio, la query per il termine *Mitigazione* ha restituito correttamente l'articolo con id: 10:

```
C:\Users\matte\OneDrive\Desktop\Aulab\Cybersecurity\progetto-finale-cyber-matteo-sbarluzzi>php
artisan tinker --execute="error_reporting(E_ALL & ~E_DEPRECATED & ~E_USER_DEPRECATED); echo jso
n_encode(\App\Models\Article::search('Mitigazione')->take(10)->get()->map->only(['id','title'])
->toArray());" | tee search-mitigazione.json
[{"id":10,"title":"Mitigazione"}]
```

Ho testato la resilienza della ricerca rispetto a payload tipici di SQL-injection:

php artisan tinker --execute="echo json_encode(\App\Models\Article::search(\"foo' OR 1=1 --\")->take(10)->get()->toArray());"

php artisan tinker --execute="echo json_encode(\App\Models\Article::search('\"; DROP TABLE users; --')->take(10)->get()->toArray());"

php artisan tinker --execute="echo json_encode(\App\Models\Article::search('test OR 1=1')->take(10)->get()->toArray());"

I test non hanno prodotto errori SQL né hanno provocato modifiche al database, i payload sono stati trattati come termini di ricerca dal motore full-text (*TNTSearch*) e non eseguiti come SQL raw.
Ad esempio, la ricerca con *foo' OR 1=1 --* ha restituito una collection di articoli ma nessun errore SQL..


```
C:\Users\matte\OneDrive\Desktop\Aulab\Cybersecurity\progetto-finale-cyber-matteo-sbarluzzi>php artisan
 tinker --execute="error_reporting(E_ALL & ~E_DEPRECATED & ~E_USER_DEPRECATED); echo json_encode(\App\
Models\Article::search(\"foo' OR 1=1 --\")->take(10)->get()->toArray());" | tee search-sqli-foo.json
[{"id":2,"title":"Prova","slug":"prova","subtitle":"Provissima","body":"<p>lorem lorem lorem lorem<\/p
>","image":"public\/images\/hnv0KrgkQt9hT0zqzyX7AVNR3v4hAOE0Za96lJ41.jpg","user_id":3,"is_accepted":1,
"category_id":1,"created_at":"2025-09-18T14:55:15.000000Z","updated_at":"2025-09-18T14:55:46.000000Z",
"__tntSearchScore__":3.2188758248682006},{"id":10,"title":"Mitigazione","slug":"mitigazione","subtitle
":"mitigazione","body":"<p><img src=\"x\" alt=\"x\"><\/p>","image":"public\/images\/z8KcHEFiD9me6IGKqT
Bt0md6HNrUfigrkX6nrtCT.jpg","user_id":3,"is_accepted":null,"category_id":1,"created_at":"2025-09-21T17
:18:16.000000Z","updated_at":"2025-09-21T17:18:16.000000Z","__tntSearchScore__":3.2188758248682006},{"
id":8,"title":"TESTTTTTTTT","slug":"testttttttt","subtitle":"teeeeeeeeeeeeeeeeeest","body":"<p>test<\
/p>","image":"public\/images\/D3dr9xfWc9w9Yv94gh8hrmHHpDKd22rI6MaMmYJw.jpg","user_id":3,"is_accepted":
null,"category_id":3,"created_at":"2025-09-21T15:22:04.000000Z","updated_at":"2025-09-21T15:22:04.0000
00Z","__tntSearchScore__":1.6094379124341003},{"id":9,"title":"PROOOOOOVAAAA","slug":"proooooovaaaa","
subtitle":"TEEEEEEEEEEEEEEEEST","body":"<img src=x onerror=\"alert('XSS')\">","image":"public\/images\
/agioiKOJvU3pj9gOu1QNeVhghfhNWJEJkec1QSZn.jpg","user_id":3,"is_accepted":null,"category_id":3,"created
_at":"2025-09-21T15:50:55.000000Z","updated_at":"2025-09-21T15:50:55.000000Z","__tntSearchScore__":1.6
094379124341003}]
```


La ricerca si è inoltre imbattuta nell'HTML malevolo: *<img src=x onerror="alert('XSS')">* presente nell'articolo con *id 9*. Per documentare il caso ho estratto il campo body dell'articolo in questione.

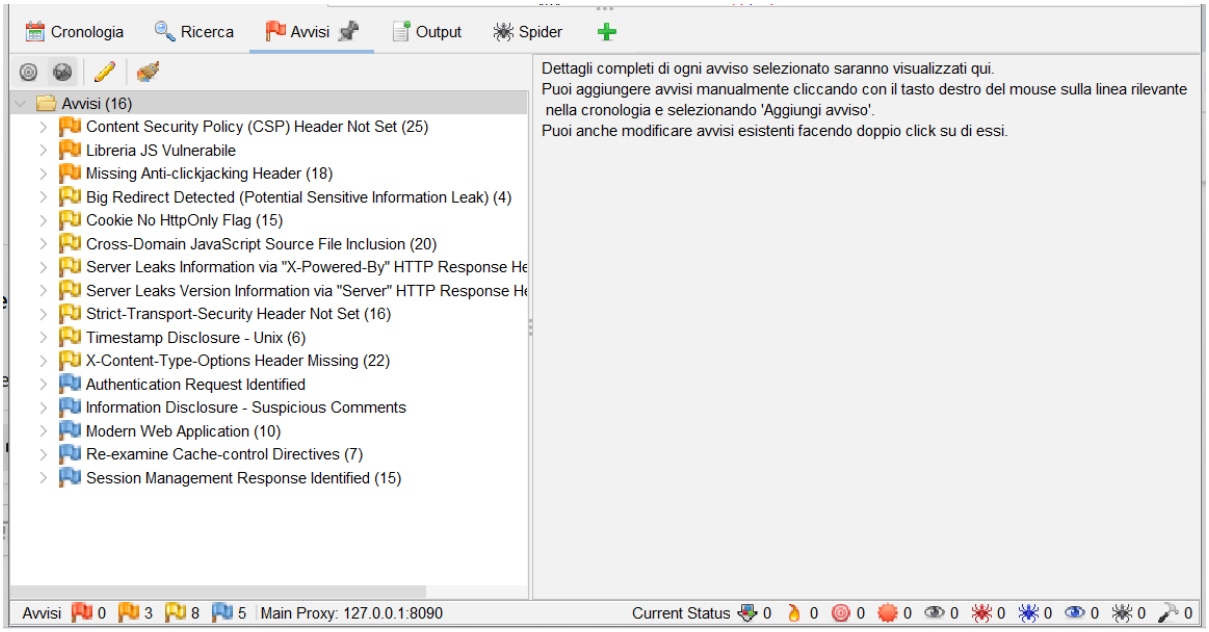
```
C:\Users\matte\OneDrive\Desktop\Aulab\Cybersecurity\progetto-finale-cyber-matteo-sbarluzzi>php
 artisan tinker --execute="echo \App\Models\Article::find(9)->body;" | tee article-9-body.html
<img src=x onerror="alert('XSS')">
```

BONUS 4

Dopo aver installato *OWASP ZAP* nel pc, ho configurato l'applicazione in modo che si collegasse al server locale del progetto, accessibile all'indirizzo *http://127.0.0.1:8000* utilizzando la porta configurata come proxy interno (*8090*). In questo modo tutto il traffico generato dal browser passa attraverso *ZAP*, che è quindi in grado di intercettare e analizzare le richieste e risposte HTTP della mia applicazione.

Dopo aver mappato l'applicazione con lo *Spider*, ho avviato un'*Active Scan* che ha testato gli endpoint scoperti e ha generato una lista di alert di sicurezza (suddivisi per gravità).

Al termine della scansione, *ZAP* ha prodotto un elenco di alert suddivisi per gravità (*High, Medium, Low, Informational*), che rappresentano potenziali problemi di sicurezza e possibili spunti di miglioramento per il progetto.



BONUS 5

Per soddisfare il bonus finale ho sostituito i controlli rigidi basati sui middleware personalizzati (*writer, revisor, admin*) con un sistema più sicuro e flessibile di *Policy* per centralizzare la logica di autorizzazione e associarla direttamente ai modelli.

Ho creato le policy per *ArticlePolicy.php* e *UserPolicy.php*. Il primo gestisce i permessi sugli articoli, quindi creazione, modifica, eliminazione, revisione, pubblicazione (le regole sono pertanto legate sia al ruolo dell'utente writer/revisor

che alla proprietà dell'articolo); il secondo gestisce l'accesso alle aree riservare come la dashboard admin e la gestione dei ruoli. Esse sono state associate automaticamente al relativo modelle grazie all'*auto discovery* presente nelle versioni a partire da Laravel 11.

I controller adesso utilizzano il metodo *$this->authorize()* per verificare le azioni al posto di quella basata su middleware dedicati ai ruoli (*UserIsAdmin, UserIsWriter, UserIsRevisor*). Un esempio in *ArticleController.php*.

```
public function edit(Article $article)
    {
        // Autore o admin
        $this->authorize('update', $article);

        return view('articles.edit', compact('article'));
    }
```

Anche le viste sono state aggiornate per rispettare le nuove regole di autorizzazione, sostituendo controlli diretti sui ruoli con le direttive Blade *@can*:

```
@can('create', \App\Models\Article::class)
                        <a class="btn btn-info"
href="{{route('articles.create')}}">Write your Article</a>
                    @endcan
```

Ho mantenuto un solo middleware aggiuntivo in *OnlyLocalAdmin.php* che limita l'accesso alle rotte admin solo se l'host è *internal.admin*. In combinazione con la *UserPolicy*, questo garantisce che l'utente sia autenticato, provenga dal dominio interno e abbia i permessi di amministratore.

```
class OnlyLocalAdmin
{
    public function handle(Request $request, Closure $next): Response
    {
        // Ramo specifico per i test (coerente con AdminTest)
        if (app()->environment('testing')) {
            // Legge sia HTTP_HOST (server bag) che header 'Host'
            $rawHttpHost = strtolower((string) $request->server('HTTP_HOST',
''));
            $rawHdrHost  = strtolower((string) $request->headers->get('host',
''));

            // Toglie eventuale :porta
            $strip = fn(string $h) => rtrim(preg_replace('/:\d+$/', '',
trim($h)), '.');

            $httpHost = $strip($rawHttpHost);
```

```
            $hdrHost  = $strip($rawHdrHost);

            // Debug
            \Log::debug('OnlyLocalAdmin DIAG (testing-head)', [
                'HTTP_HOST_raw' => $rawHttpHost,
                'HDR_HOST_raw'  => $rawHdrHost,
                'HTTP_HOST'     => $httpHost,
                'HDR_HOST'      => $hdrHost,
            ]);

            // Se uno dei due è "internal.admin" → allow
            if ($httpHost === 'internal.admin' || $hdrHost ===
'internal.admin') {
                return $next($request);
            }

            // Altrimenti, nei test *sbagliato host* → redirect (come si
aspetta AdminTest)
            return redirect(route('homepage'))->with('alert', 'Not
Authorized');
        }

        // RAMO NORMALE (sviluppo/prod)
        $allowedBase = 'internal.admin';

        $candidates = [
            (string) $request->server('HTTP_HOST', ''),
            (string) $request->server('SERVER_NAME', ''),
            (string) $request->headers->get('host', ''),
            (string) $request->getHttpHost(),
            (string) $request->getHost(),
        ];

        foreach ([(string) $request->fullUrl(), (string) url()->current(),
(string) config('app.url', '')] as $u) {
            $h = parse_url($u, PHP_URL_HOST);
            if ($h) {
                $candidates[] = (string) $h;
            }
        }

        $norm = function (string $h): string {
            $h = trim(strtolower($h));
            if ($h === '') return '';
            $h = preg_replace('/:\d+$/', '', $h); // rimuovi porta
            return rtrim($h, '.');                // rimuovi trailing dot
        };
```

```
        $candidates = array_values(array_filter(array_unique(array_map($norm,
$candidates))));

        $ok = false;
        foreach ($candidates as $h) {
            if ($h === $allowedBase || str_ends_with($h, '.' . $allowedBase))
{
                $ok = true;
                break;
            }
        }

        if (!$ok) {
            return redirect(route('homepage'))->with('alert', 'Not
Authorized');
        }

        return $next($request);
    }
}
```

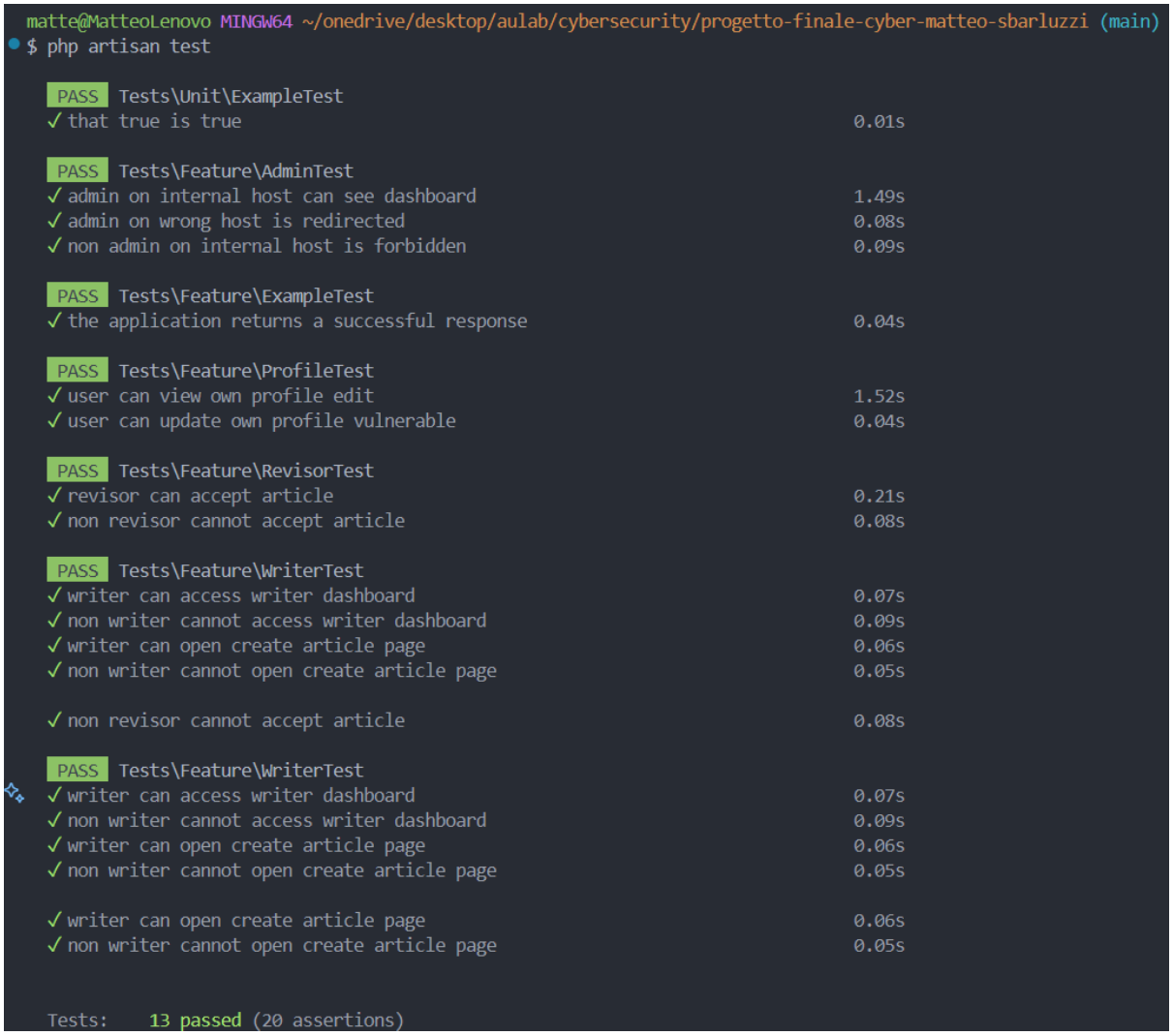
Il file *web.php* è stato semplificato; adesso le rotte *writer/revisor/admin* sono sotto un generico *Route::middleware('auth')*, la distinzione dei permessi è demandata esclusivamente alle Policy e solo il gruppo delle rotte admin mantiene il middleware aggiuntivo *admin.local*.

Inoltre, per verificare la correttezza delle Policy ho inserito diversi test feature:

- *AdminTest.php*; verifica che solo un admin su host interno possa accedere alla dashboard.
- *WriterTest.php*; verifica la creazione e gestione articoli da parte del writer.
- *RevisorTest.php*; verifica che solo i revisori possano accettare o rifiutare articoli.
- *ProfileTest.php*; controlla che un utente possa modificare solo il proprio profilo.

I test garantiscono che il comportamento sia coerente con le Policy introdotte e proteggano da regressioni future.

Test effettuati mediante il comando *php artisan test*



Per concludere, grazie all'introduzione delle *Policy*, l'applicazione ha ora un sistema di autorizzazioni più modulare con regole centralizzate nelle Policy, si presenta più sicuro poiché tutte le verifiche passano da *authorize* o *@can*, e si presta più leggibile dal momento che non ci sono più middleware dedicati ai ruoli dentro le rotte.