Проектна задача по предметот Етичко Хакирање на ФИНКИ со тема:

**OWASP Топ 10 ранливости демонстрирани во Laravel**

Изработено од:

Бисера Галеска 211023

Бојана Аризанковска 211018

СОДРЖИНА

[ВОВЕД 4](#_Toc201874948)

[ПОТРЕБЕН ХАРДВЕР И СОФТВЕР 4](#_Toc201874949)

[Хардвер 4](#_Toc201874950)

[Софтвер 4](#_Toc201874951)

[Подготовка на развојната околина 5](#_Toc201874952)

[OWASP Топ 10 ранливости 6](#_Toc201874953)

[A01: Broken Access Control 6](#_Toc201874954)

[Чести недостатоци во контролата на пристап (Access Control) : 6](#_Toc201874955)

[Како да се справиме? 7](#_Toc201874956)

[Пример 7](#_Toc201874957)

[Заштита од страна на Laravel 8](#_Toc201874958)

[A02:2021 – Cryptographic Failures 10](#_Toc201874959)

[Чести недостатоци 10](#_Toc201874960)

[Како да се справиме? 10](#_Toc201874961)

[Пример 11](#_Toc201874962)

[Заштита од страна на Laravel 12](#_Toc201874963)

[A03:2021-Injection 12](#_Toc201874964)

[Чести ситуации 12](#_Toc201874965)

[Како да се справиме? 12](#_Toc201874966)

[Пример 13](#_Toc201874967)

[Заштита од страна на Laravel 14](#_Toc201874968)

[A04:2021-Insecure Design 14](#_Toc201874969)

[Чести сценарија 15](#_Toc201874970)

[Како да се справиме? 15](#_Toc201874971)

[Пример 15](#_Toc201874972)

[Заштита од страна на Laravel 16](#_Toc201874973)

[A05:2021-Security Misconfiguration 17](#_Toc201874974)

[Чести сценарија 17](#_Toc201874975)

[Како да се справиме? 18](#_Toc201874976)

[Пример 18](#_Toc201874977)

[Заштита од страна на Laravel 19](#_Toc201874978)

# ВОВЕД

OWASP Топ 10 ги идентификува најкритичните безбедносни ризици со кои се соочуваат веб-апликациите (owasp.org). Овој проект има за цел да создаде Laravel веб-апликација која намерно ги демонстрира овие ранливости, нудејќи им на програмерите практично искуство за учење.

Преку овој проект, ќе научите:

* Да препознавате чести безбедносни стапици во веб-развојот,
* Како Laravel се справува со заканите, и
* Да ги примените најдобрите практики за ефикасна заштита на Вашите апликации.

Занемарувањето на практиките за безбедно кодирање може да доведе до кражба на податоци, оштетување на угледот, правни последици и прекин на услугите. Па затоа, со едукација на програмерите за овие ранливости, можеме значително да го намалиме бројот на небезбедни веб-апликации, правејќи го дигиталниот свет побезбеден за сите.

# ПОТРЕБЕН ХАРДВЕР И СОФТВЕР

## Хардвер

|  |  |  |
| --- | --- | --- |
| Компонента | Минимум | Препорачано |
| CPU | 2 јадра со VT‑x/AMD‑V | 4 јадра |
| RAM | 8 GB | 16 GB |
| Меморија | 20 GB слободни | 40 GB SSD |
| OS | Windows 10 / macOS 12 / Linux 5.x | Последната стабилна верзија |

Софтвер

|  |  |  |
| --- | --- | --- |
| ЦЕЛ | Алатка | Верзија |
| Container runtime | **Docker**/ Podman | 24+ |
| Оркестрација | Docker Compose | v2 |
| PHP | PHP | >= 8.1 |
| PHP Framework | Laravel | >= 10.x |
| Authentication features | Laravel Breeze | >= Laravel 9.x |
| База на Податоци | MySQL | Latest |
| Package manager | Composer | Latest |
| Веб прелистувач | Chrome/Firefox | Latest |
| IDE | VS Code, PhpStorm, … | Latest |

Опционално: VirtualBox / VMware ако работите со целосни виртуелни машини

***Безбедносно предупредување*:** *Овој проект намерно воведува ранливости. Апликацијата треба да се извршува само во изолирани средини. Не ја изложувајте на јавни мрежи.*

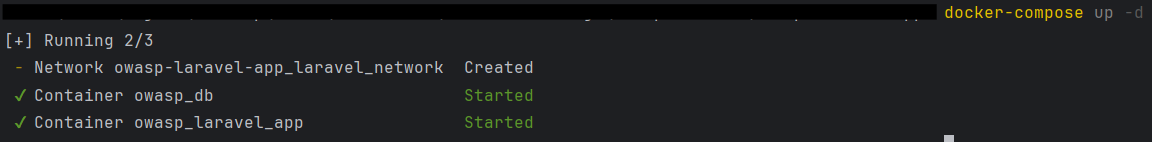
# Подготовка на развојната околина

1. Инсталирајте Docker Desktop
2. Отворете терминал и клонирајте го git репозиториумот:

<https://github.com/BojanaAri/Ethical-Hacking-Project.git>

cd owasp-laravel-app

1. Креирај ги и започни ги Докер контејнерите:

docker compose up -d –build

1. Пополнете ја базата на податоци:

docker compose exec owasp-laravel-app php artisan migrate --seed

# OWASP Топ 10 ранливости

## A01: Broken Access Control

Access control – политика која изјаснува кои акции/дејства корисиникот може да ги извршува во рамки на системот, врз основа на неговите доделени дозволи. Кога овие контроли се погрешно конфигурирани, може да доведе до сериозни нарушувања на безбедноста, вклучувајќи откривање на информации, модификација и уништување на податоци или извршување на функции надвор од дефинираните граници на корисникот.

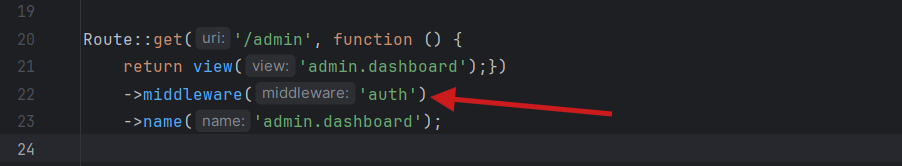
### Чести недостатоци во контролата на пристап (Access Control) :

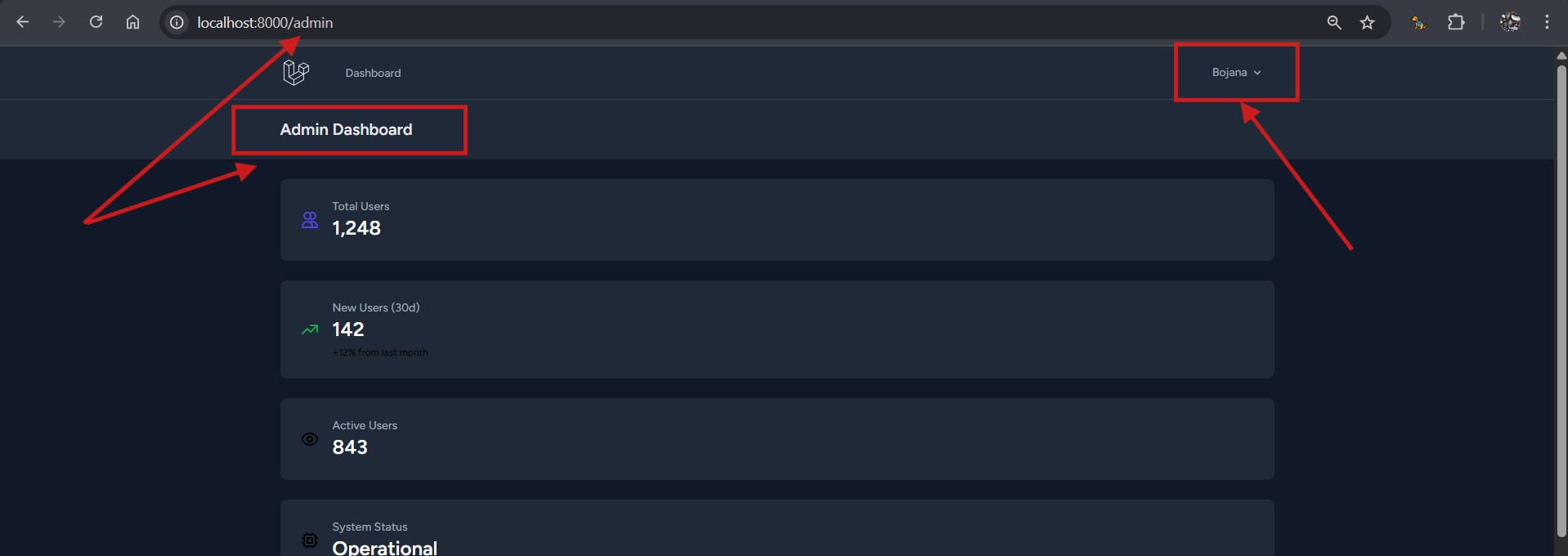
* Прекршување на најмалата привилегија / deny by default – се случува каде што пристапот треба да се дозволи само за одредени можности, улоги или корисници, но е достапен за сите. Во суштина, ако нешто не е експлицитно дозволено, треба да се одбие.
* Заобиколување на проверките (Bypassing Checks) - напаѓачите можат да ги заобиколат контролите за пристап со директно манипулирање со компонентите на апликацијата. Ова вклучува неовластено менување на параметрите (модифицирање на URL параметрите), принудно прелистување (директен пристап до ограничени URL адреси), менување на внатрешната состојба на апликацијата, модифицирање на HTML или користење на алатки за напад за манипулирање со API барањата..
* Небезбедни директни референци на објекти (IDOR) - напаѓачот може да ја прегледа или уреди сметката или податоците на друг корисник едноставно со промена на уникатен идентификатор во URL-то или API-барањето.
* Недостаток на API controls: Недостаток на автентикација за дејства како POST, PUT и DELETE.
* Ескалација на привилегии - Дејствување како помоќен корисник без соодветно овластување.
* Неовластено менување на метаподатоци: Менување на JWT, колачиња или скриени полиња за добивање зголемен пристап.
* Погрешни конфигурации на CORS - Овозможување неовластен пристап до API-ја од недоверливи извори.
* Присилно прелистување - Пристап до ограничени страници без автентикација или соодветни привилегии.

### Како да се справиме?

* Стандард да биде одбиј (Deny) - Ако ресурсот не е експлицитно јавен, секој пристап треба да биде одбиен по стандард.
* Централизирани и повеќекратно употребливи механизми - Имплементирајте механизми за контрола на пристап еднаш и применувајте ги конзистентно низ целата апликација. Минимизирајте ја употребата на споделување ресурси со вкрстено потекло (CORS) за да ги намалите потенцијалните вектори на напад.
* Спроведување на сопственост на записи - дизајнирај контроли за пристап такви што ќе ја налагаат сопственоста на записот, наместо само да прифаќаат дека корисникот може да креира, чита, ажурира или брише билокој запис.
* Имплементирајте ограничувања на деловната логика - Уникатните бизнис ограничувања специфични за апликацијата (на пр., максимална количина на нарачка) треба да се спроведени во доменските модели.
* Безбеден датотечен систем - Оневозможете листање на директориуми на веб-серверот и осигурете се дека чувствителните датотеки како што се .git или резервните датотеки не се достапни во рамките на веб-коренот.
* Сеопфатно евидентирање и известување (Log access control failures) - Имплементирајте соодветни известувања за администраторите кога е потребно
* Ограничување на брзината - Применете ограничувања на брзината на пристапот до API и контролерот.
* Робустно управување со сесии - Веднаш по одјавувањето, идентификаторите на сесии со состојба на серверот треба да станат невалидни. За JWT токени без состојба, користете краткотрајни токени за да го минимизирате прозорецот на можности за напаѓачите. За JWT со подолг век на траење, почитувајте ги стандардите на OAuth за поништување на токени.

### Пример

****Дадениот код на сликата е ранлив поради недостатокот на middleware што ќе провери дали најавениот корисник има дозвола за пристап на страницата /admin. Поради тоа, билокој корисник може да пристапи на страницата /admin без разлика дали има дозвола или не.



### Заштита од страна на Laravel

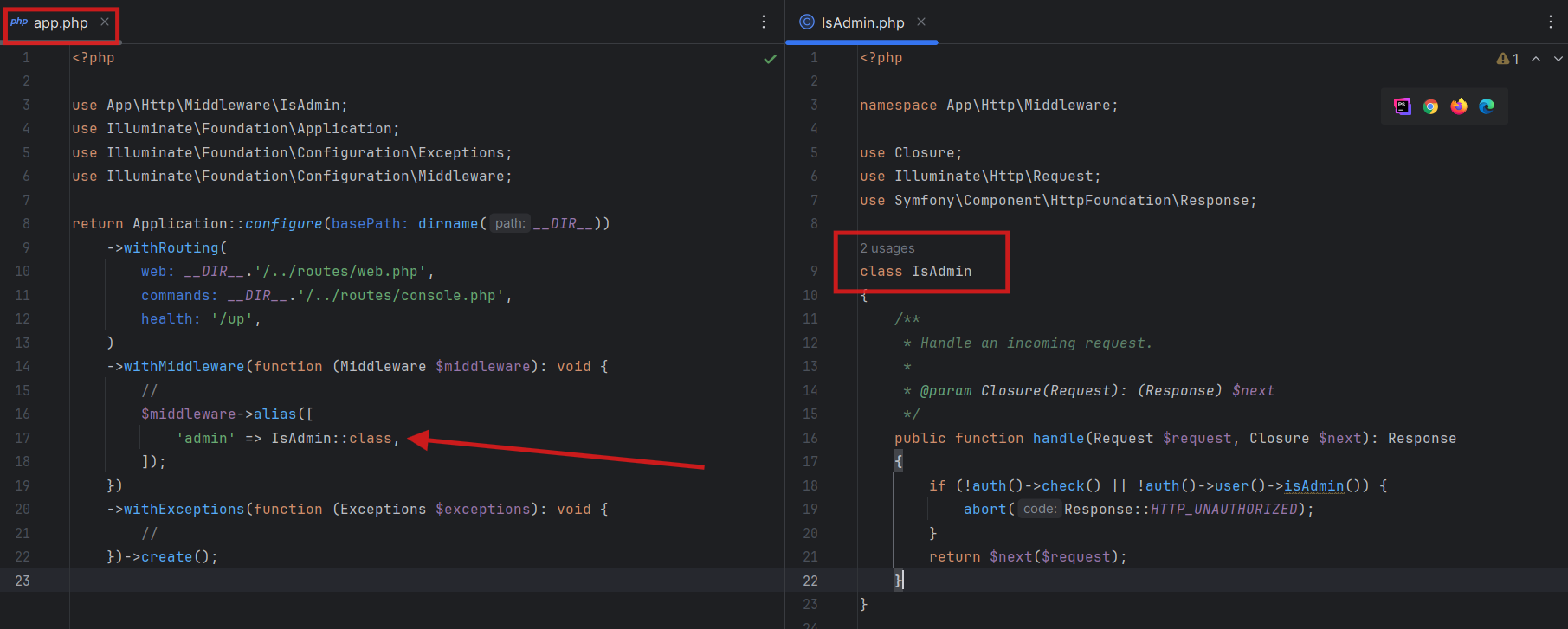
1. **Користење Middleware**

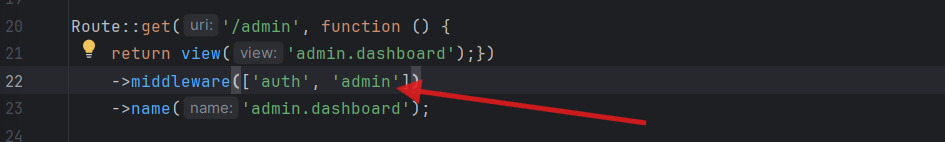
Вградената можност што Laravel ја нуди за безбедносни закани е креирање на нов middleware – IsAdmin со помош на функцијата php artisan make:middleware IsAdmin. Целта на овој middleware е да претставува посредник за заштита на рутата – применете го на сите рути за конзистентност!

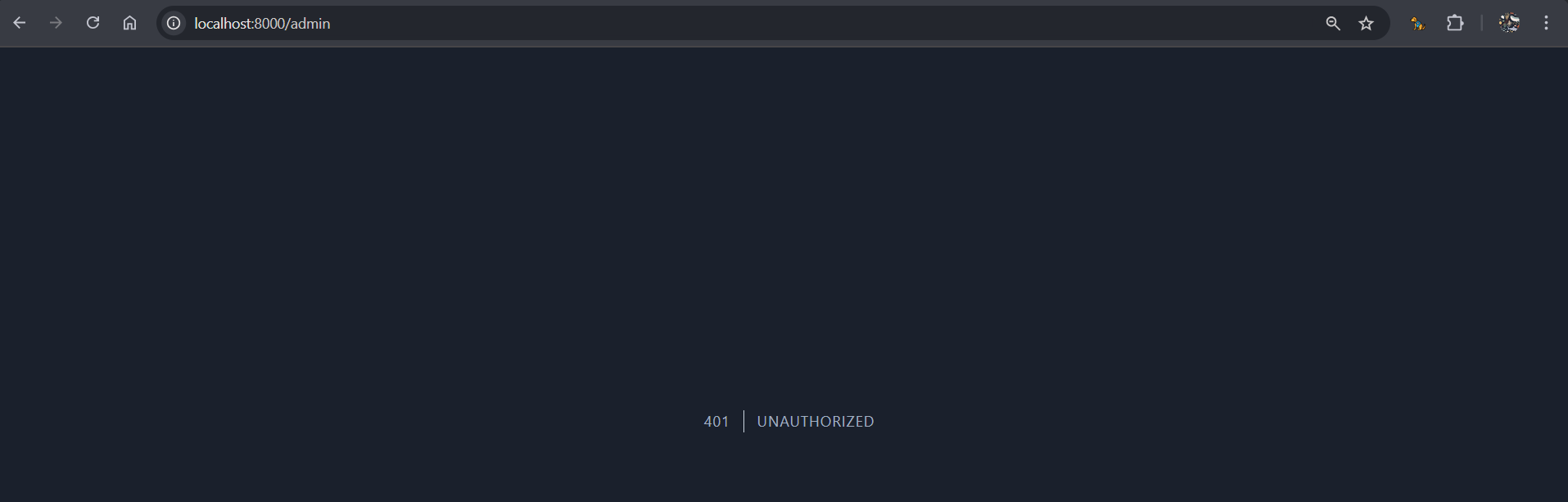
* Handle() методот - го прифаќа тековното барање и closure што го претставува следниот посреднички софтвер/контролер. Мора да врати одговор.

Проверка на услов - if (!auth()->check() || !auth()->user()->is\_admin) – дали корисникот не е најавен или дали корисникот не е админ.

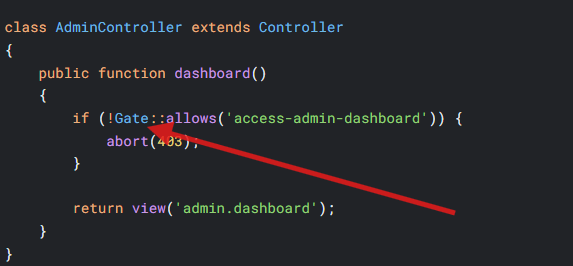
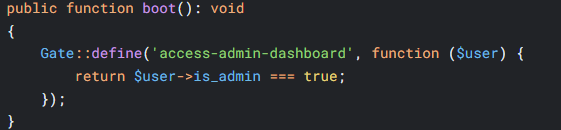
Овој middleware мора да се регистрира во app.php каде ќе му дадеме надимак “admin”, додека пак во рутата што го дефинира пристапот на страницата /admin се додава како middleware ‘admin’. Сега, кога ќе пристапи на корисник без привилегии, серверот ќе врати 401 Unauthorized.

****

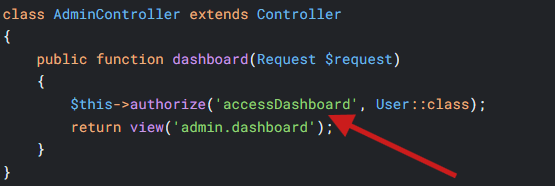
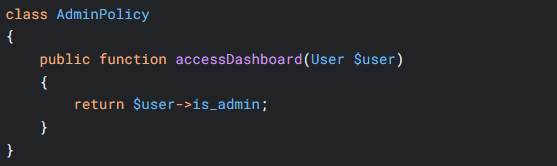
****

****

1. **Дефинирање на порти (Gate) –**

Во AuthServiceProvider.php се дефинира портата што потоа може да се користи во билокој контролер каде што е потребно со помош на !Gate::allows(‘name’)

1. **Користење полиси**

****Полисата се креира со помош на командата php artisan make:policy AdminPolicy --model=User каде потоа ќе имплементираме функција што проверува дали корисникот е админ според атрибутот is\_admin во табелата users. Тогаш во потребниот контролер доволно е да се наведе вградената функција authorize каде како параметри прима името на функцијата и моделот.

## A02:2021 – Cryptographic Failures

Во својот систем, првото нешто што треба да одлучите е потребите за заштита на податоците во транзите и при мирување. Податоци на кои им е потребна дополнителна заштита се најчесто податоци кои се заштитени од законите за приватност (како лозинки, кредитни картички, матичен број,...).

### Чести недостатоци

* Пренос на податоци во чист текст (clear text) - Испраќање чувствителни податоци без енкрипција (на пр., преку HTTP или со непринудени надградби на TLS).
* Слаба криптографија - Користење на застарени алгоритми, протоколи или слаби (или ре-искористени) криптографски клучеви и небезбедно складирање на клучеви.
* Недостаток на спроведување (enforcement)- Не се спроведува никаква енкрипција преку безбедносни заглавија.
* Несоодветна валидација - Неуспех во валидацијата на сертификатите на серверот.
* Неточни практики на енкрипција - Злоупотреба на вектори за иницијализација, небезбедни режими на работа (на пр., ECB) или користење на енкрипција кога е потребно автентицирано енкрипција.
* Лошо изведување на клучеви - Користење на лозинки директно како клучеви без соодветни функции за изведување на клучеви (password base key derivation function).
* Застарени хешови - Користење на застарени хеш функции (MD5, SHA1) или шеми за дополнување (PKCS #1 v1.5).
* Протекување на информации - Експлоатирачки криптографски пораки за грешки или информации од страничниот канал.

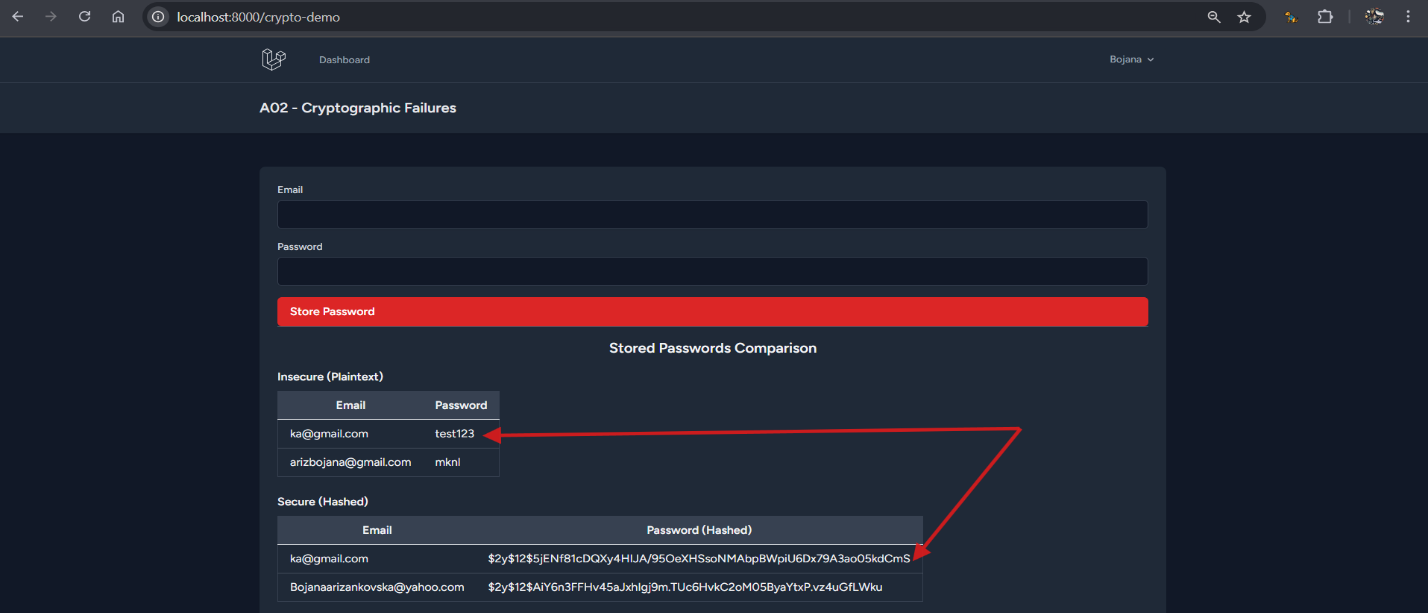
### Како да се справиме?

* Чувајте само неопходни чувствителни податоци и доколку е можно, токенизирајте ги или скратете ги.
* Шифрирајте ги сите чувствителни податоци и во мирување и во пренос користејќи силни, ажурирани алгоритми, протоколи (како TLS со тајност нанапред) и правилно управување со клучеви
* Спроведувајте енкрипција (на пр., со HSTS).
* Оневозможете кеширање за одговори на чувствителни податоци.
* Избегнувајте застарени протоколи за чувствителни податоци.
* Зачувајте ги лозинките користејќи силни, salted, адаптивни функции за хеширање (Argon2, scrypt, bcrypt, PBKDF2).
* Користете соодветни вектори за иницијализација и секогаш претпочитајте автентицирано криптирање.
* Генерирајте криптографски клучеви случајно и сменете ги лозинките во клучеви преку соодветни функции за деривација.
* Обезбедете криптографски непредвидлива безбедна случајност (randomness).
* Одделно верификувајте ги сите безбедносни конфигурации

### Пример

Со цел да се презентира оваа ранливост, креиран е нов контролер CryptographicFailuresTestController со нов поглед (view) crypto-demo.blade.php што за цел има да ја зачува лозинката безбедно и небезбедно и истата разлика да ја прикаже на самата страна.

* Во првиот случај се запишува во табелата insecure\_passwords со обичен текст (plain text) – РАНЛИВО!
* Додека во вториот случај се запишува во табелата secure\_passwords со хеширање (по стандард Laravel Breeze го користи bcrypt).



На сликата е прикажано како е зачувана лозинката со обичен текст, а како се хеширан.

### Заштита од страна на Laravel

* bcrypt() / Hash::make() – Безбедно salted хеширање (Bcrypt/Argon2) за лозинки
* CSRF Protection – Се осигурува дека барањата од формите се легитимни користејќи signed tokens
* APP\_KEY – Ги штити енкриптираните колачиња, сесии и податоци со AES-256
* Crypt::encrypt() / Crypt::decrypt() - Безбедно ги шифрира произволните вредности
* Правило за валидација на лозинки - Laravel не чува лозинки во обичен текст — дури и Breeze користи хеширани лозинки одма.

## A03:2021-Injection

Апликацијата станува ранлива на напади со инјектирање кога директно користи невалидирани, нефилтрирани или несредени податоци доставени од корисникот во динамични барања, команди или повици на interpreter. Вообичаените типови на напади со инјектирање вклучуваат SQL, NoSQL, OS команда, ORM, LDAP и инјектирање на Expression Language (EL) или Object Graph Navigation Library (OGNL).

### Чести ситуации

* Податоците доставени од корисник не се валидирани, филтрирани и санитизирани.
* Динамичките барања или непараметаризираните повици се користат без соодветен context-aware escaping.
* Непријателските податоци манипулираат со параметрите за пребарување на ORM за да извлечат неовластени записи.
* Злонамерните податоци се директно споени во SQL барања, команди или складирани процедури.

*Детекција*: Најефикасниот метод е преглед на изворниот код, заедно со автоматско тестирање на сите влезни параметри (заглавија, URL, колачиња, JSON, SOAP, XML). Интегрирањето на алатките SAST, DAST и IAST во CI/CD pipelines помага да се откријат недостатоци пред production.

### Како да се справиме?

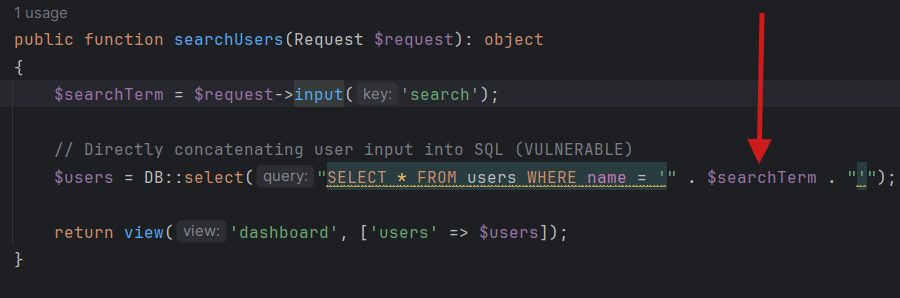
Спречувањето на инјектирањето бара чување на податоците одвоени од командите и барањата:

* Користи безбеден АРI, кој избегнува целосно користење на интерпретерот, обезбедува параметризиран интерфејс или мигрира кон алатки за мапирање на релации на објекти (ORM).
  + Забелешка: Дури и кога се параметризирани, складираните процедури сè уште можат да воведат SQL инјекција ако PL/SQL или T-SQL спојуваат барања и податоци или извршуваат непријателски податоци со EXECUTE IMMEDIATE или exec().
* Користете позитивна валидација на влез од страна на серверот (positive server-side input validation). Ова не претставува целосна одбрана бидејќи многу апликации бараат специјални знаци, како што се текстуални области или API-ја за мобилни апликации.
* За сите преостанати динамички барања, escape специјалните знаци користејќи ја специфичната синтакса за избегнување за тој интерпретер (SQL structures such as table names, column names, and so on cannot be escaped, and thus user-supplied structure names are dangerous).
* Ограничи го откривањето на податоци - Користете контроли како LIMIT во SQL барањата за да го минимизирате влијанието на успешната инјекција со спречување на масовно откривање на податоци.

### Пример

Проблемот со наведениот код е директното користење на податокот доставен од корисник без никаква проверка. Што може да се случи?

Влезен напад: ' OR 1=1 –

Прашалникот сега ќе стане : SELECT \* FROM users WHERE name = '' OR 1=1 --' што ќе ги врати сите корисници!

#### Решение

Laravel е безбеден по стандард – користи Laravel’s query builder и Eloquent ORM што автоматски ги валидира влезните податоци и спречува директно SQL инјектирање (освен ако намерно не напишете небезбеден суров SQL).

Безбедната верзија прикажана на сликата:

* Го валидира влезниот формат на сите корисници
* Има параметезирани прашања така што го користи вградениот query builder на Laravel наместо raw SQL (пример ' OR '1'='1)
* Го има истото корисничко искуство, но со соодветна безбедност

### Заштита од страна на Laravel

* Query Builder / Eloquent ORM – користи готови изрази за одбрана од SQL Injection
* DB::select("... WHERE name = ?", [$name]) - Parameter binding built-in
* Blade’s {{ $variable }} - Escapes output за одбрана од XSS
* validate() input – Спречува масовно доделување и инјектирање
* @csrf – Спречува од CSRF напад

## A04:2021-Insecure Design

Небезбеден дизајн се однесува на безбедносни слабости што произлегуваат од архитектонски или дизајнерски недостатоци, а не од грешки во имплементацијата. Иако добро дизајнираниот систем сè уште може да пати од недостатоци во имплементацијата, небезбедниот дизајн, по својата природа, не може да се поправи со совршена имплементација, бидејќи потребните безбедносни мерки никогаш не биле промислени (неуспех соодветно да се процени ризикот).

* Управување со барања и ресурси - Оваа фаза вклучува темелно собирање и преговарање на бизнис барањата, вклучувајќи ги потребите за заштита на доверливоста на податоците, интегритетот, достапноста и автентичноста. Мора да се состават технички безбедносни барања - функционални и нефункционални, и треба да се планира и преговара соодветен буџет за сите безбедносни активности во текот на фазите на дизајнирање, градење и тестирање.
* Безбеден дизајн - култура и методологија која постојано ги оценува заканите и обезбедува робустен дизајн и тестирање на кодот против познати методи на напад. Ова вклучува интегрирање на моделирање на закани во процесите на развој, прецизно дефинирање на точни и неуспешни состојби во корисничките приказни и темелно анализирање на претпоставките и условите. Документацијата на овие наоди во корисничките приказни е од суштинско значење, заедно со учење од минатите грешки и поттикнување подобрувања.
* Безбеден животен циклус на развој (SDLC) - фундаментален за безбеден софтвер. Ова вклучува искористување на безбедни шеми на дизајн, методологии за „поплочен пат“ (претходно одобрени, безбедни патеки), безбедни библиотеки на компоненти и соодветни алатки. Ангажирањето на специјалисти за безбедност од почетокот на проектот до одржувањето е од битно значење. Рамки како OWASP Software Assurance Maturity Model (SAMM) можат да помогнат во структурирањето на овие напори за безбеден развој.

### Чести сценарија

* Откривање на кориснички идентификатори во URL-адреси
* Верување на влезните податоци од frontend (на пр., скриени полиња или идентификатори на формулари)
* Непроверка на сопственоста на ресурсите

### Како да се справиме?

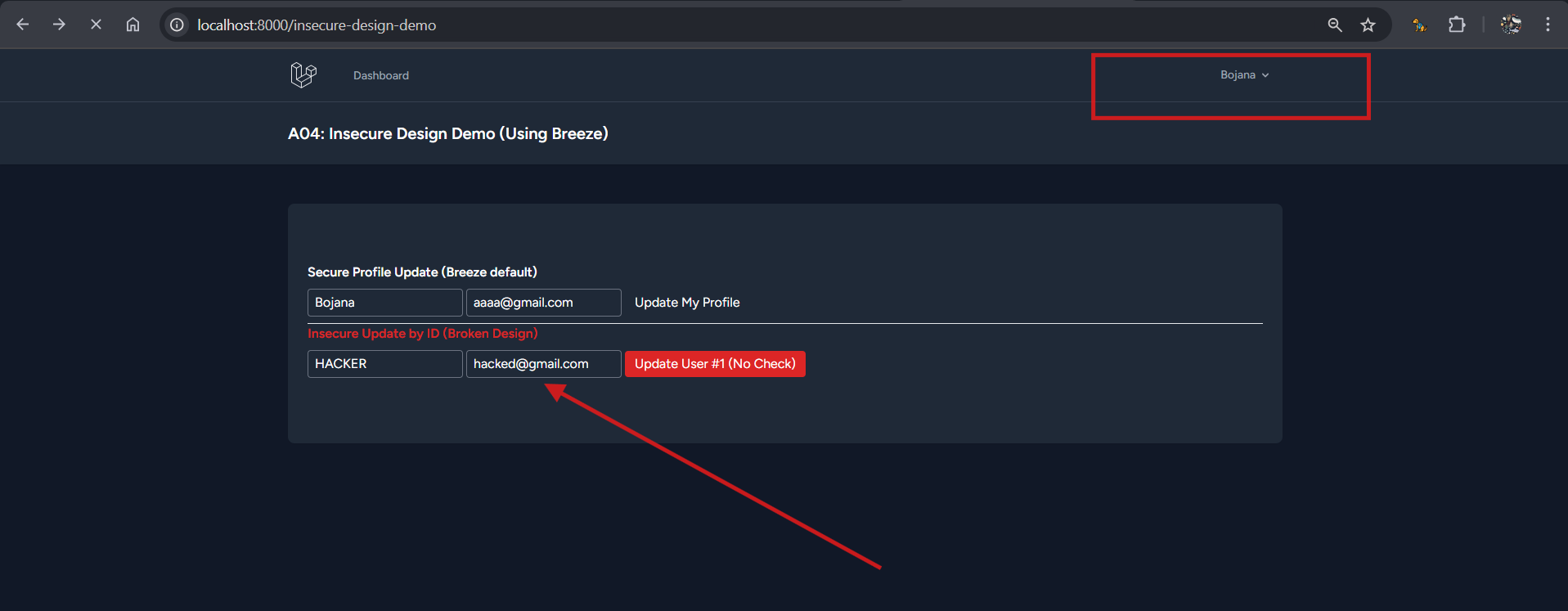
* Воспоставете и користете безбеден животен циклус на развој со професионалци од AppSec за да помогнете во евалуацијата и дизајнирањето на контроли поврзани со безбедноста и приватноста
* Воспоставете и користете библиотека со безбедни шеми на дизајн или компоненти подготвени за употреба
* Користете моделирање на закани за критична автентикација, контрола на пристап, бизнис логика и key flows.
* Интегрирајте го безбедносниот јазик и контролите во корисничките приказни
* Интегрирајте проверки на веродостојноста на секое ниво од вашата апликација
* Напишете тестови за единици и интеграција за да потврдите дека сите критични текови се отпорни на моделот на закана.
* Одвојте ги нивоата на системските и мрежните слоеви во зависност од потребите за изложеност и заштита
* Ограничете ја потрошувачката на ресурси по корисник или услуга

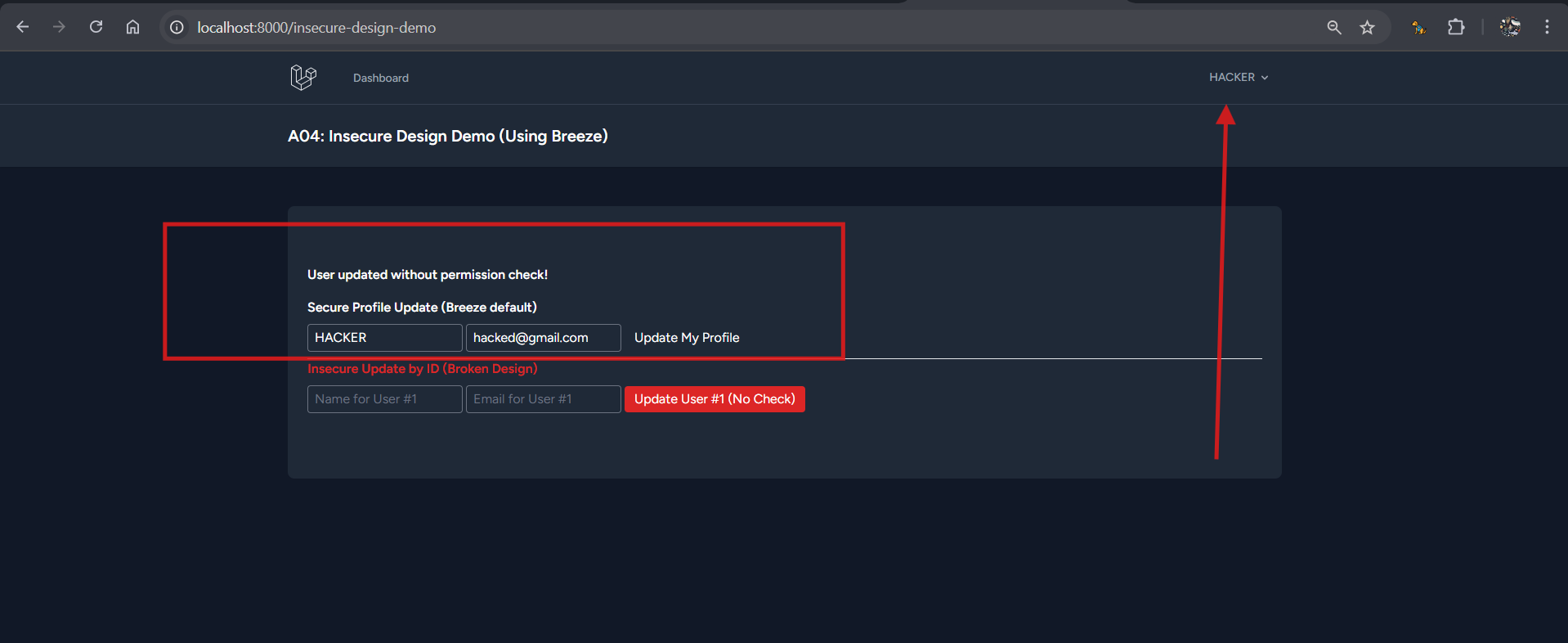
### Пример

Со цел да се прикаже ранливоста, додаден нов метод во ProfileController - insecureUpdate() што при прифаќа кориснички ID во URL-то, (пр. POST /update-any-profile/1) и директно го ажурира името и е-поштата на тој корисник без да провери кој е најавен.

Проблемот - Секој автентициран корисник може да го промени профилот на билокој друг корисник со погодување на ID.







### Заштита од страна на Laravel

Laravel Breeze при конфигурација, во Profile Controller-от автоматски креира безбедна функција за ажурирање на лозинки (освен ако намерно не напишете ранлив код). Воедно, небезбедниот дизајн се однесува на безбедносни слабости што произлегуваат од архитектонски грешки, а не од грешки во имплементацијата.

За да спречите OWASP A04:2021 во Laravel (или која било апликација), мора да ги примените принципите на безбеден дизајн — што значи дека мора да вградите безбедност во Вашата архитектура и бизнис логика од самиот почеток.



* Policies & Gates - Овозможуваат да спроведувате безбедносни правила на логичко ниво
* Form Request Validation - Чисто раздвојување на логиката за валидација и контролерот
* Custom Middleware - посредни меѓу нивоа
* Route model binding + policies - Автоматски осигурува дека корисникот има пристап до моделот
* authorizeResource() - Додава проверки на политиките на RESTful контролерите

## A05:2021-Security Misconfiguration

Апликацијата може да се соочи со потенцијални ранливости ако има недостаток на безбедноста низ билокој дел од апликацијата или пак погрешно конфигурирани дозволи, особено во услугите во облак.

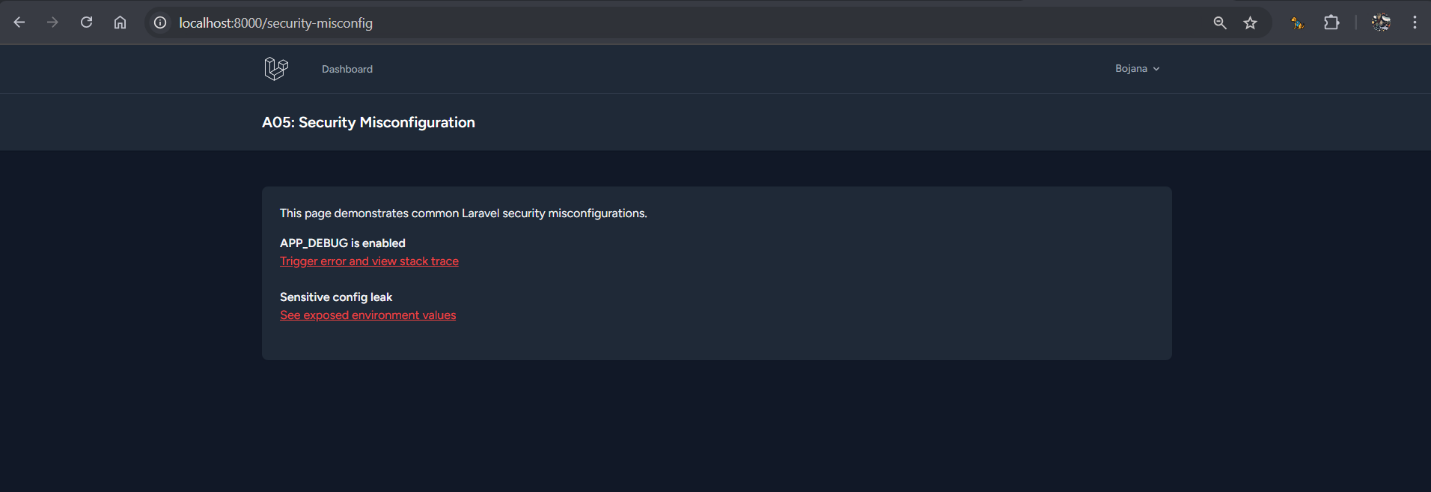
### Чести сценарија

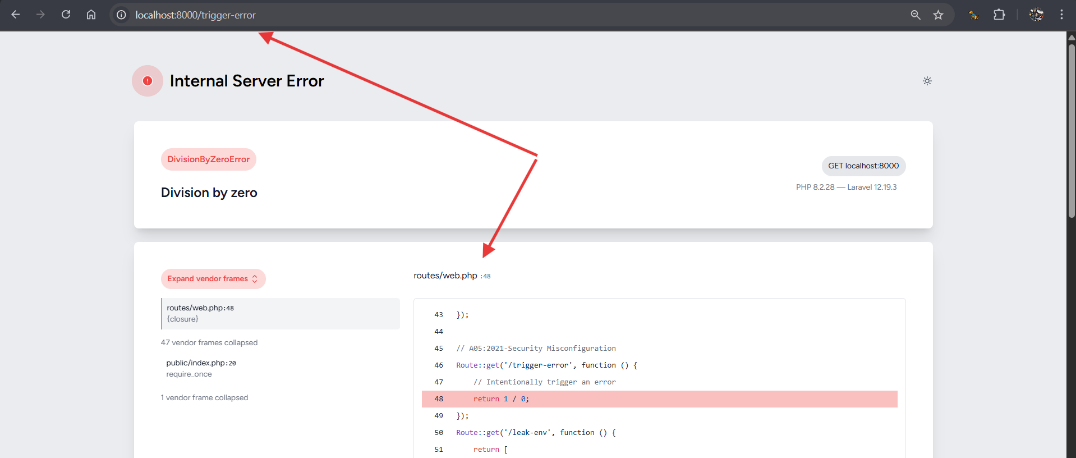
* Користење на стандардни лозинки, кориснички сметки,...
* Изложување на debug mode во продукција
* Заборавени логови или грешки кои содржат сензитивни информации во продукција
* Неправилно конфигуриран CORS, заглавија, пораки за грешки
* Користење на застарени или небезбедни софтверски компоненти
* Премногу дозволи (на пр. 777)

### Како да се справиме?

* Имплементирајте процес на повторувачко зајакнување: Ова го олеснува и брзо распоредувањето на нови, безбедно заклучени средини. Развојните, QA и производствените средини треба да имаат идентични конфигурации, кои се разликуваат само во акредитивите. Автоматизирајте го овој процес за да го поедноставите поставувањето на нови безбедни средини.
* Одржувајте минимална платформа: Инсталирајте само основни функции, компоненти и документација. Отстранете или избегнувајте инсталирање на неискористени функции и рамки за да ја намалите површината на нападот.
* Редовно прегледувајте и ажурирајте ги конфигурациите: Како дел од вашиот процес на управување со закрпи, постојано прегледувајте и ажурирајте ги конфигурациите врз основа на безбедносни белешки, ажурирања и закрпи. Обрнете големо внимание на дозволите за складирање во облак, како што се дозволите за S3 кофа.
* Усвојте сегментирана архитектура на апликацијата: Ова обезбедува ефикасно и безбедно раздвојување помеѓу компонентите или закупувачите преку техники како сегментација, контејнеризација или групи за безбедност во облак (ACL).
* Испраќајте безбедносни директиви до клиентите: Имплементирајте безбедносни заглавија и други директиви за подобрување на безбедноста од страна на клиентот.
* Автоматизирајте ја верификацијата на конфигурацијата: Воспоставете автоматизиран процес за редовно потврдување на ефикасноста на конфигурациите и поставките во сите средини.

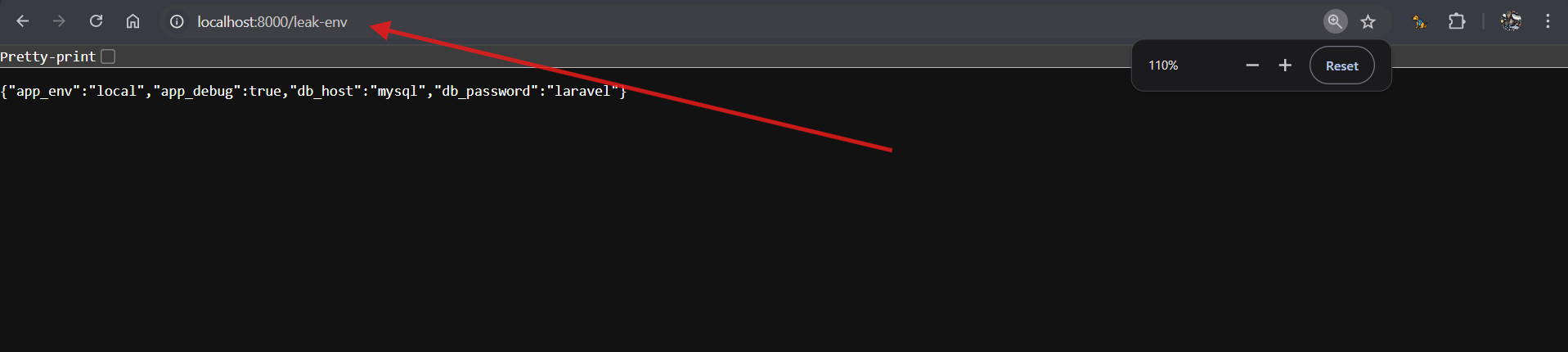
### Пример

Со цел да прикаже оваа ранливост, креирана е страна која намерно воведува погрешни безбедносни конфигурации.

* Намерно сменета конфигурација во продукција за дебагирање APP\_DEBUG=true Посетата на /trigger-error ќе прикаже:
  + - Целосна трага на стекот
    - Патеки на датотеки
    - Верзија на Laravel
    - Исечоци од код
    - Променливи на околината во некои случаи
* Заборавена или откриена датотеката .env

Доколку е изложено, ова открива:

* + - Кој е host на базата на податоци
    - Акредитиви за базата
    - Тип на околина



* Заборавени логови 

### Заштита од страна на Laravel

* APP\_DEBUG=false in .env - Спречува изложување на детали за грешки
* APP\_ENV=production - Оневозможува небезбедни однесувања на развивачите
* .env е игнориран во .gitignore - Спречува протекување на акредитиви во контролата на изворниот код
* config:cache, route:cache - Оптимизира и ја зајакнува конфигурацијата
* Public directory (/public) - Laravel прикажува само дозволени датотеки
* Custom error pages (/resources/views/errors/) - Спречува протекување на генерички грешки