

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](#_Toc202122558)

[ПОТРЕБЕН ХАРДВЕР И СОФТВЕР 4](#_Toc202122559)

[Хардвер 4](#_Toc202122560)

[Софтвер 4](#_Toc202122561)

[Подготовка на развојната околина 5](#_Toc202122562)

[OWASP Топ 10 ранливости 6](#_Toc202122563)

[A01: Broken Access Control 6](#_Toc202122564)

[Чести недостатоци во контролата на пристап (Access Control) : 6](#_Toc202122565)

[Како да се справиме? 7](#_Toc202122566)

[Пример 7](#_Toc202122567)

[Заштита од страна на Laravel 8](#_Toc202122568)

[A02:2021 – Cryptographic Failures 10](#_Toc202122569)

[Чести недостатоци 10](#_Toc202122570)

[Како да се справиме? 10](#_Toc202122571)

[Пример 11](#_Toc202122572)

[Заштита од страна на Laravel 12](#_Toc202122573)

[A03:2021-Injection 12](#_Toc202122574)

[Чести сценарија 12](#_Toc202122575)

[Како да се справиме? 12](#_Toc202122576)

[Пример 13](#_Toc202122577)

[Заштита од страна на Laravel 14](#_Toc202122578)

[A04:2021-Insecure Design 14](#_Toc202122579)

[Чести сценарија 15](#_Toc202122580)

[Како да се справиме? 15](#_Toc202122581)

[Пример 15](#_Toc202122582)

[Заштита од страна на Laravel 16](#_Toc202122583)

[A05:2021-Security Misconfiguration 17](#_Toc202122584)

[Чести сценарија 17](#_Toc202122585)

[Како да се справиме? 18](#_Toc202122586)

[Пример 18](#_Toc202122587)

[Заштита од страна на Laravel 19](#_Toc202122588)

[A06:2021 - Vulnerable and Outdated Components 20](#_Toc202122589)

[Чести недостатоци: 20](#_Toc202122590)

[Како да се справиме? 20](#_Toc202122591)

[Пример 21](#_Toc202122592)

[A07:2021 – Identification and Authentication Failures 22](#_Toc202122593)

[Чести недостатоци: 22](#_Toc202122594)

[Како да се справиме? 22](#_Toc202122595)

[Пример 23](#_Toc202122596)

[A08:2021 – Software and Data Integrity Failures 24](#_Toc202122597)

[Чести недостатоци: 24](#_Toc202122598)

[Како да се справиме? 25](#_Toc202122599)

[Пример 26](#_Toc202122600)

[A09:2021 – Security Logging and Monitoring Failures 28](#_Toc202122601)

[Чести недостатоци: 29](#_Toc202122602)

[Како да се справиме? 29](#_Toc202122603)

[Пример: 31](#_Toc202122604)

[A10:2021 – Server-Side Request Forgery (SSRF) 32](#_Toc202122605)

[Чести недостатоци: 32](#_Toc202122606)

[Како да се справиме? 33](#_Toc202122607)

[Пример: 34](#_Toc202122608)

[Евалуација на трети решенија за скенирање ранливости во код и нивна ефикасност 35](#_Toc202122609)

[БРИШЕЊЕ НА ДЕФИНИРАНАТА ОКОЛИНА 37](#_Toc202122610)

[ЗАКЛУЧОК 37](#_Toc202122611)

# ВОВЕД

OWASP Топ 10 e индустриски стандарден документ што ги идентификува најкритичните безбедносни ризици со кои се соочуваат веб-апликациите, ги рангира и дава препораки за нивно справување. Ризиците се рангираат според фреквенцијата на откриените безбедносни дефекти, сериозноста на откриените ранливости и големината на нивното потенцијално влијание. 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** | 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

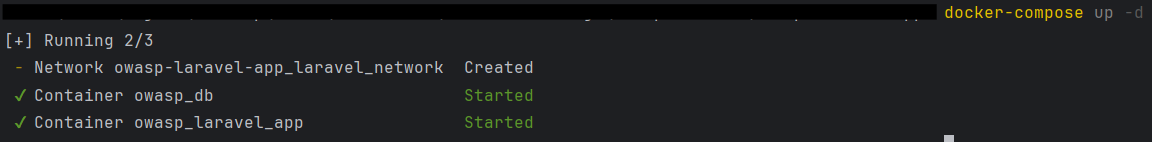
1. Инсталирајте Docker Desktop
2. Отворете терминал и клонирајте го git репозиториумот:

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

cd owasp-laravel-app

1. Креирај ги и започни ги Докер контејнерите:

При прво започнување на контејнерите: docker compose up -d –build

Секое последователно: docker compose up -d

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.

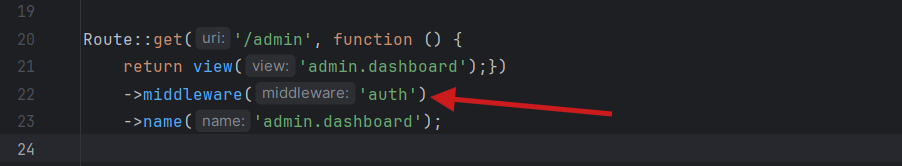
### Чести недостатоци во контролата на пристап (Access Control) :

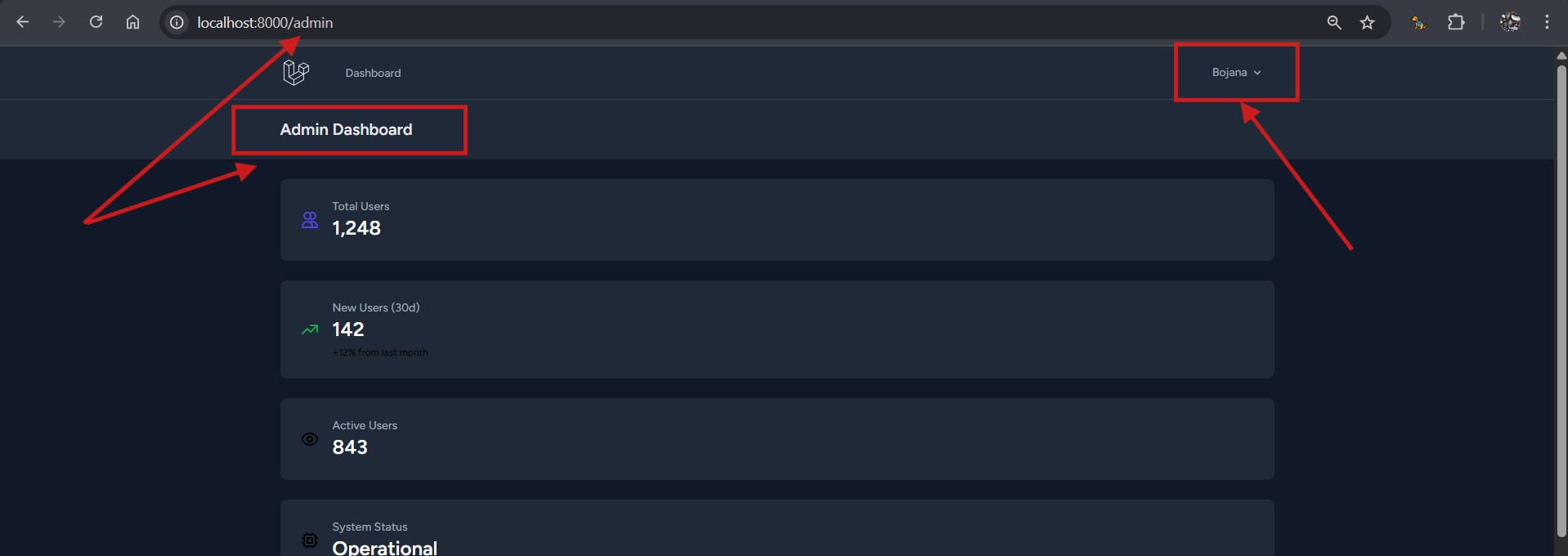
* Прекршување на најмалата привилегија / deny by default – се случува каде што пристапот треба да се дозволи само за одредени можности, улоги или корисници, но е достапен за сите. Во суштина, ако нешто не е експлицитно дозволено, треба да се одбие.
* Заобиколување на проверките (Bypassing Checks) - напаѓачите 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

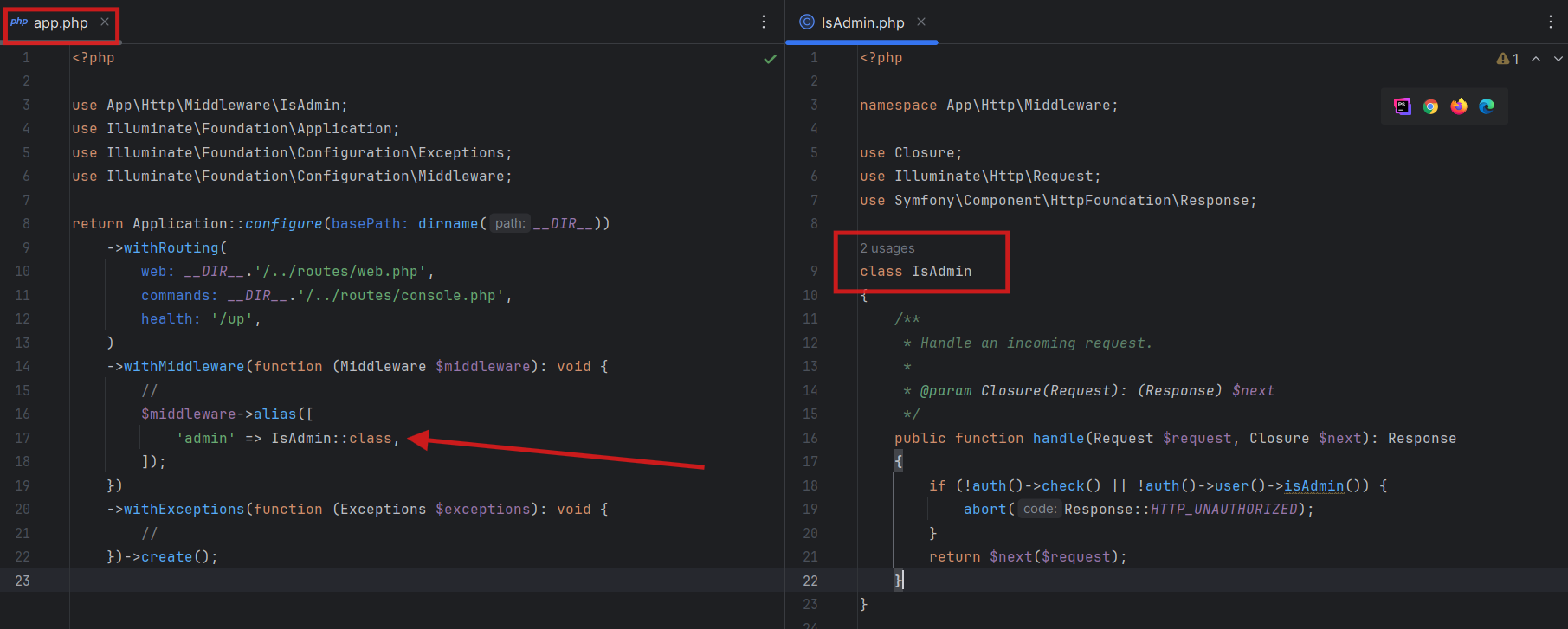
1. **Користење Middleware**

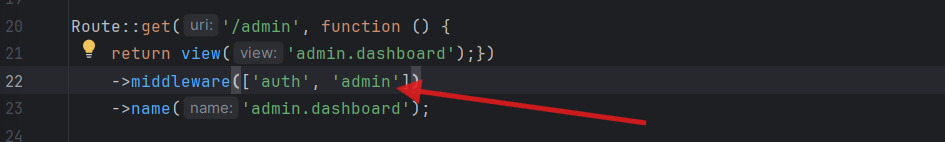
Вградената можност што Laravel ја нуди за безбедносни закани е креирање на нов middleware – IsAdmin со помош на функцијата php artisan make:middleware IsAdmin. Целта на овој middleware е да претставува посредник за заштита на рутата – применете го на сите рути за конзистентност!

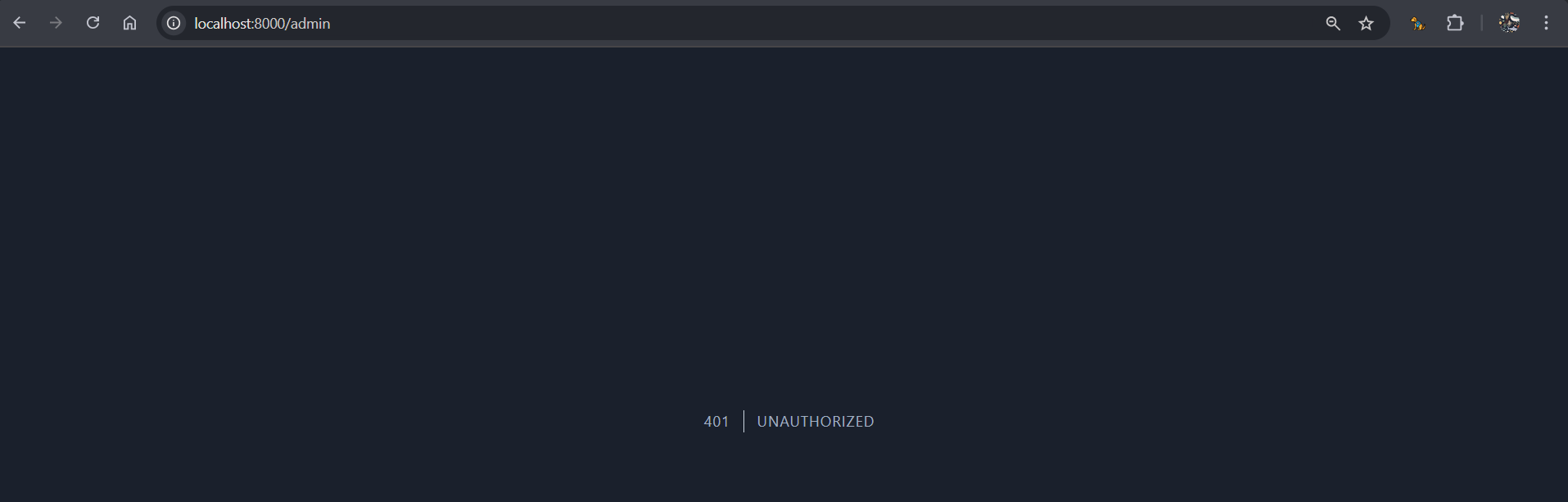
* Handle() методот - го прифаќа тековното барање и closure што го претставува следниот посреднички софтвер/контролер. Мора да врати одговор.

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

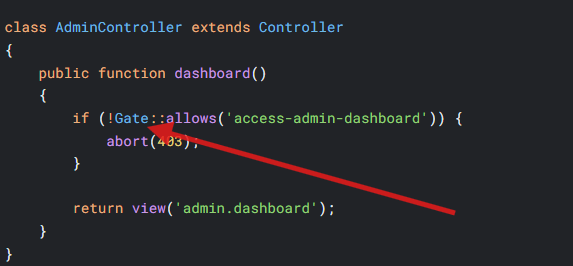
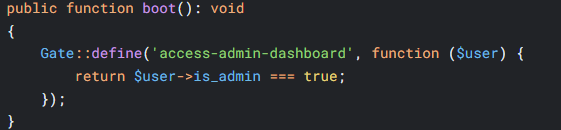
Овој middleware мора да се регистрира во app.php каде ќе му дадеме надимак “admin”, додека пак во рутата што го дефинира пристапот на страницата /admin се додава како middleware ‘admin’. Сега, кога ќе пристапи на корисник без привилегии, серверот ќе врати 401 Unauthorized.

****

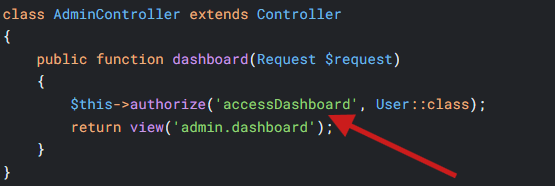
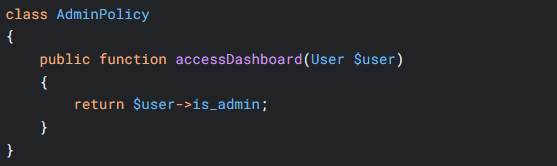
****

****

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.

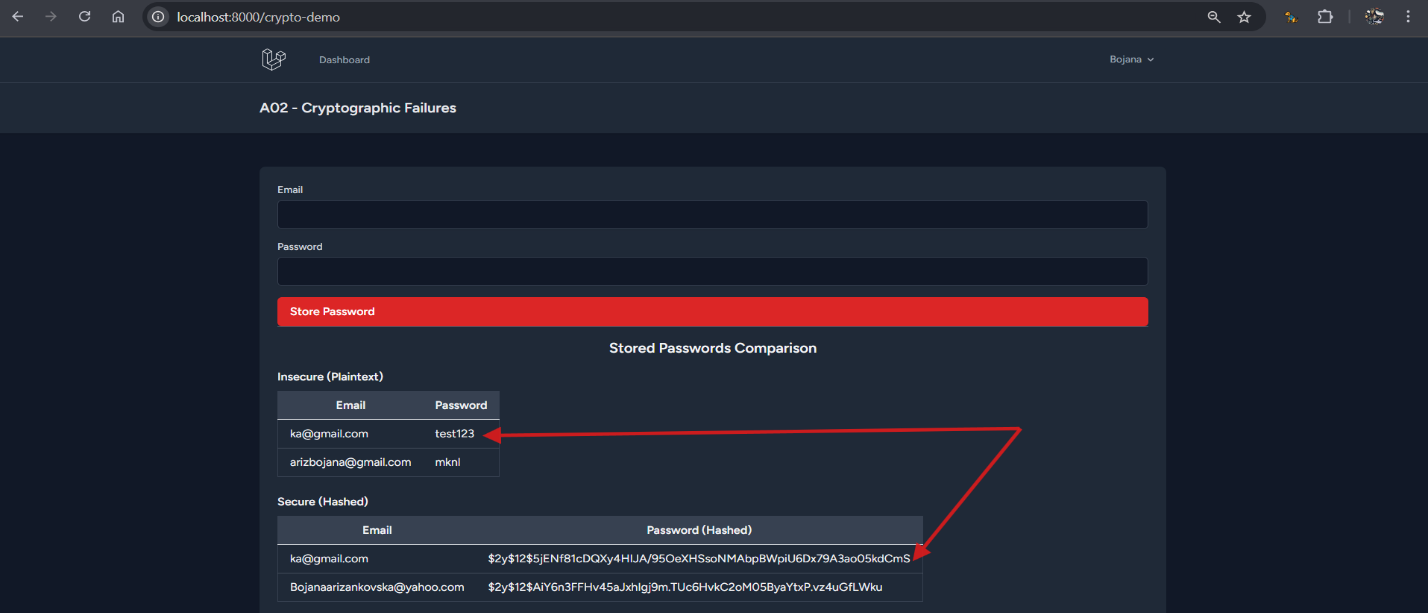
### Како да се справиме?

* 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 сценарија

* 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?

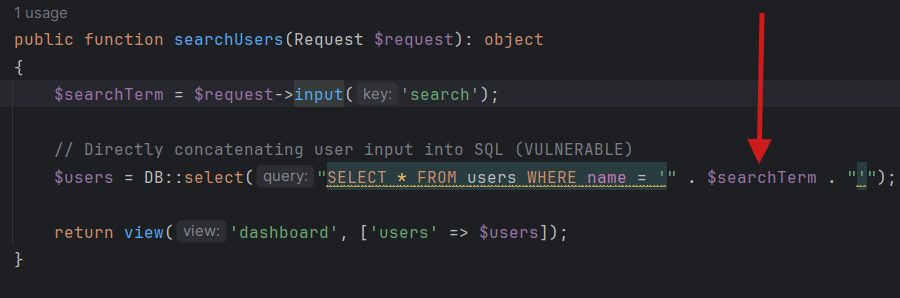
Спречувањето на инјектирањето бара чување на податоците одвоени од командите и барањата:

* 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?

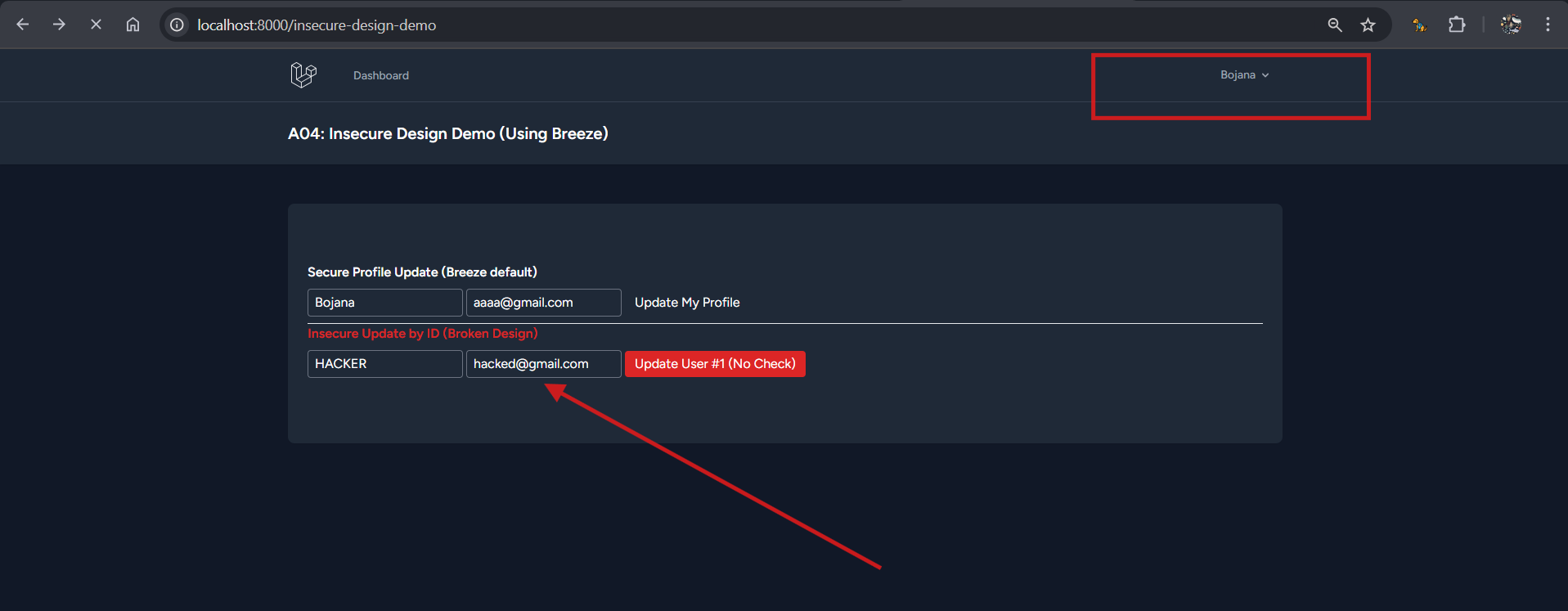
* Воспоставете и користете безбеден животен циклус на развој со професионалци од AppSec за да помогнете во евалуацијата и дизајнирањето на контроли поврзани со безбедноста и приватноста
* Воспоставете и користете библиотека со безбедни шеми на дизајн или компоненти подготвени за употреба
* Користете 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

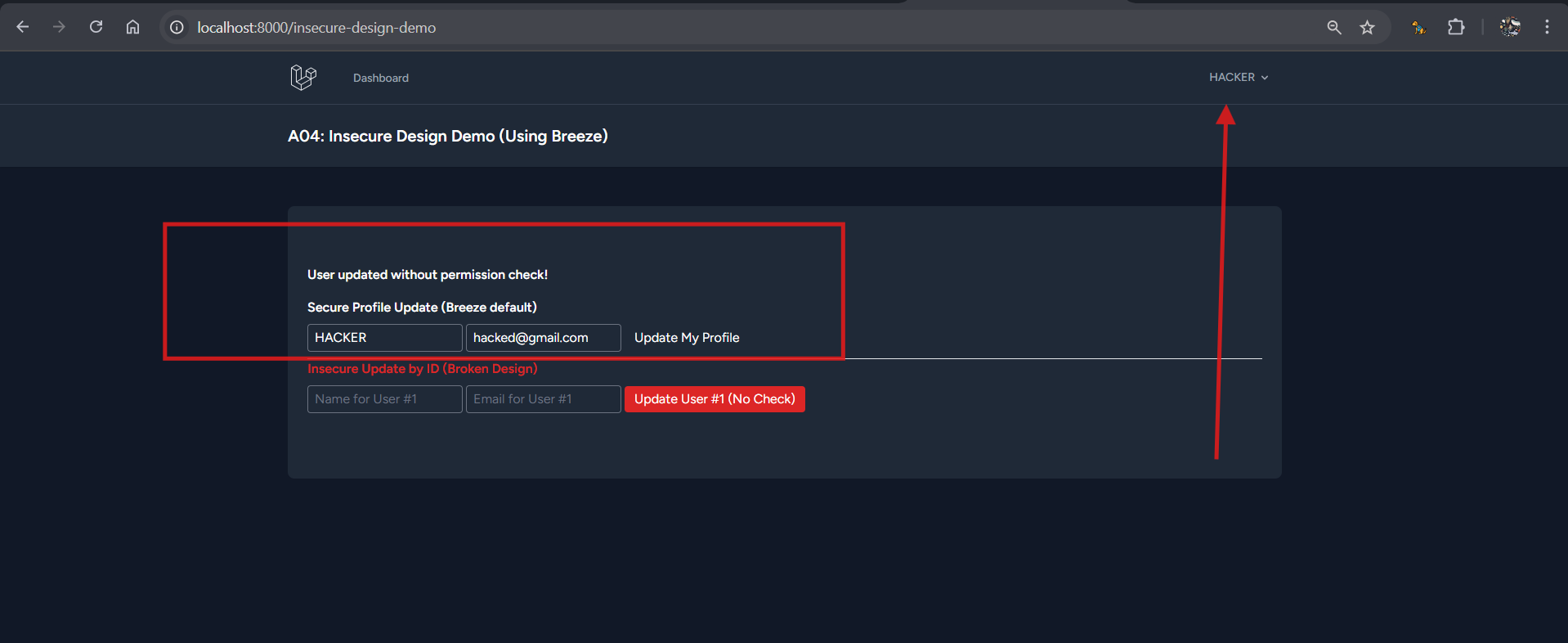
### Пример

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.

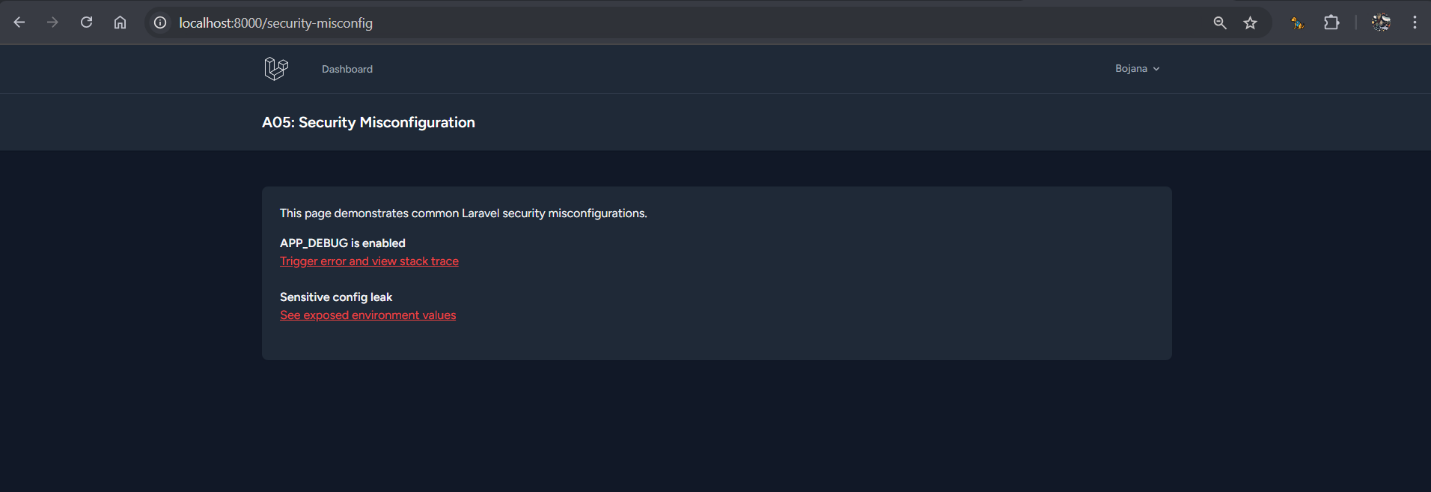
### Чести сценарија

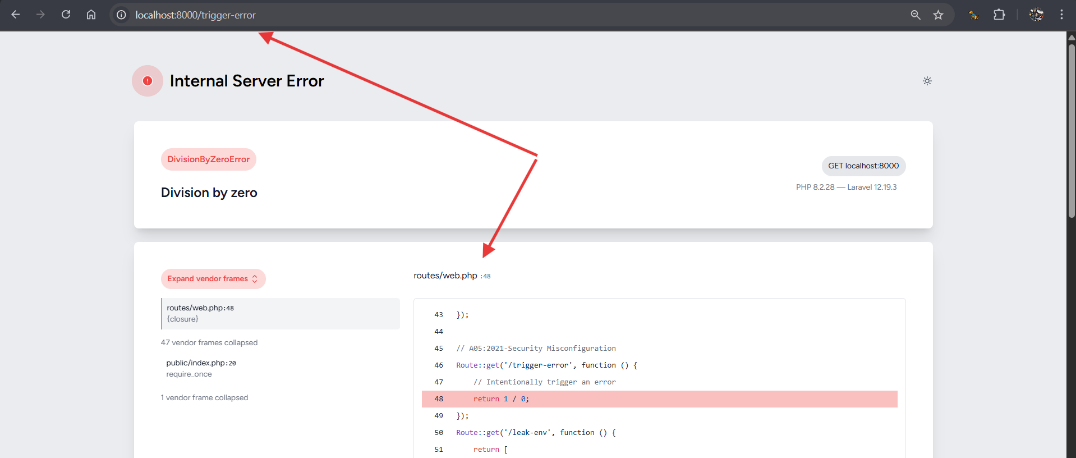
* Користење на стандардни лозинки, кориснички сметки,...
* Изложување на debug mode во продукција
* Заборавени логови или грешки кои содржат сензитивни информации во продукција
* Неправилно конфигуриран CORS, заглавија, пораки за грешки
* Користење на застарени или небезбедни софтверски компоненти
*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

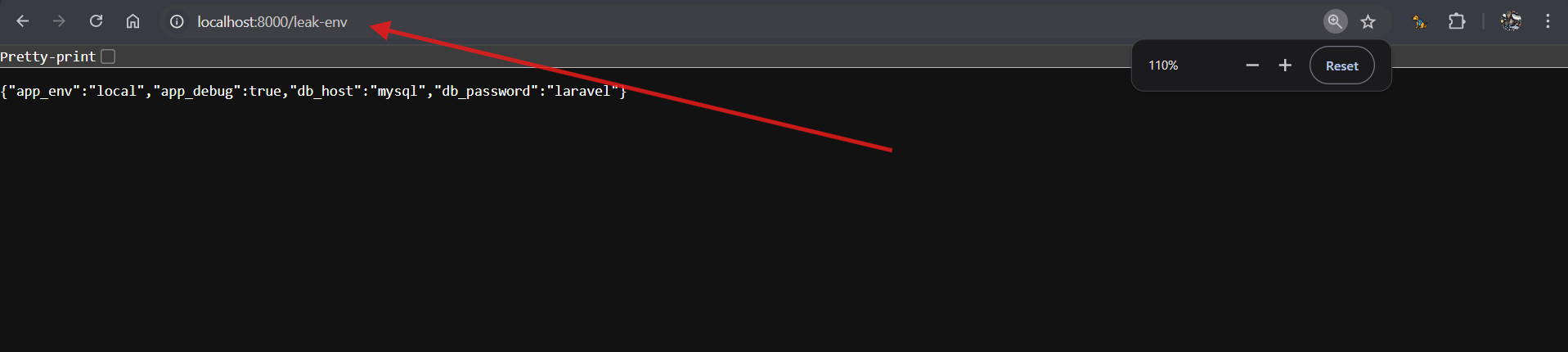
### Пример

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

## A06:2021 - Vulnerable and Outdated Components

Ранливите и застарени компоненти претставуваат значаен ризик за безбедноста на апликациите. Оваа категорија ги опфаќа сите компоненти што ги користи апликацијата, вклучувајќи библиотеки, frameworks и други софтверски модули, кои имаат познати безбедносни пропусти или се застарени и повеќе не добиваат безбедносни ажурирања. Напаѓачите често ги искористуваат овие ранливости бидејќи тие се јавно познати и честопати имаат достапни експлоити.

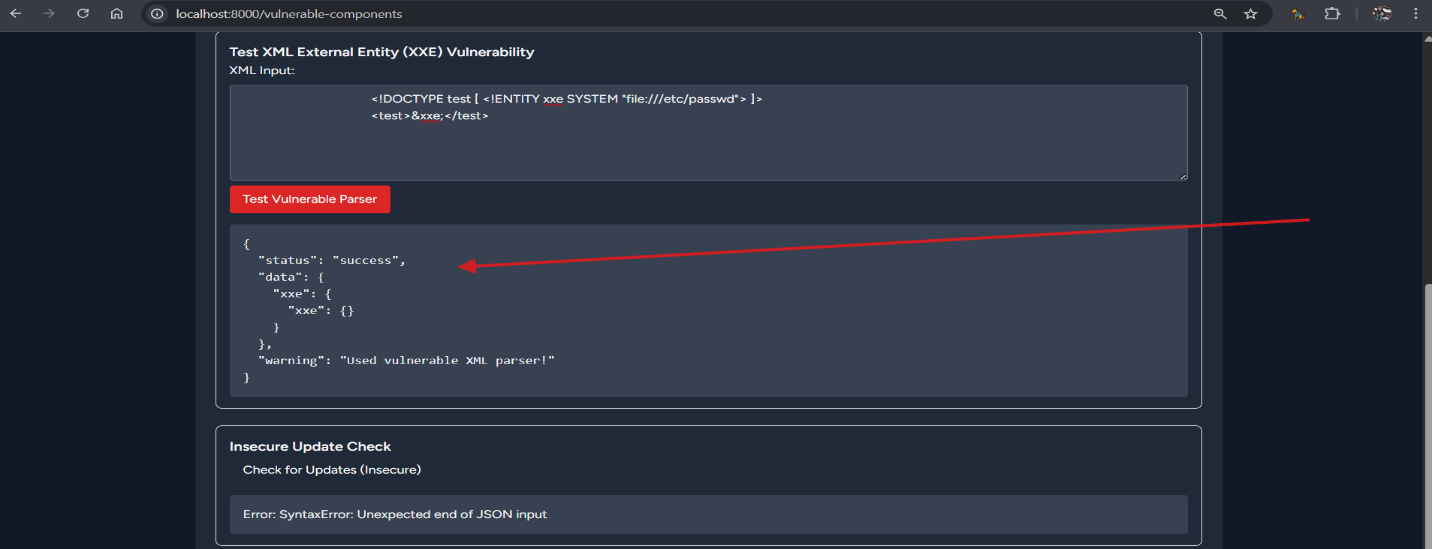
### Чести недостатоци:

* Неажурирани компоненти: Користење на стари верзии на библиотеки, frameworks, оперативни системи или веб-сервери кои содржат познати CVE (Common Vulnerabilities and Exposures).
* Искористување на познати CVE: Напаѓачите лесно наоѓаат и искористуваат ранливости кои се јавно објавени за застарени компоненти.
* Недоволно следење: Недостаток на редовно скенирање и следење на компонентите за нови безбедносни пропусти.
* Искористување на функции со застарена безбедност: Употреба на функционалности во компонентите кои се означени како депрецирани (deprecated) или небезбедни.
* Небезбедно преземање и управување со компоненти: Преземање компоненти од недоверливи извори или невалидирање на интегритетот на преземените пакети.
* Insecure deserialization: Неправилно ракување со сериализирани податоци, што може да доведе до далечинско извршување на код или други напади.

### Како да се справиме?

* Редовни ажурирања: Континуирано ажурирајте ги сите компоненти – оперативни системи, веб-сервери, бази на податоци, frameworks и библиотеки – до најновите стабилни верзии.
* Користење на безбедни верзии: Пред имплементација, проверете ги компонентите за познати CVE. Користете алатки за анализа на безбедноста на зависностите.
* Отстранување на неискористени компоненти: Минимизирајте ја површината за напад со отстранување на сите неискористени библиотеки, функционалности и зависности.
* Извори со доверба: Преземајте компоненти само од официјални и доверливи извори.
* Валидација на интегритетот: Користете хеш проверки или дигитални потписи за да го потврдите интегритетот на компонентите по преземањето.
* Сегментација на мрежата: Изолирајте ги компонентите во мрежата за да го ограничите потенцијалниот домет на напад.
* Автоматизирани алатки: Интегрирајте алатки за анализа на безбедноста (SCA - Software Composition Analysis) во вашиот CI/CD pipeline за автоматско откривање на ранливи компоненти.
* Заштита од Insecure Deserialization: Избегнувајте сериализација на чувствителни податоци. Користете безбедни формати за податоци (на пр., JSON) и валидирајте ги влезните податоци

### Пример



## A07:2021 – Identification and Authentication Failures

Ранливостите во идентификацијата и автентикацијата (Authentication Failures) се јавуваат кога апликациите неправилно ги имплементираат механизмите за проверка на идентитетот на корисниците. Ова може да доведе до компромисирање на лозинки, сесиски токени или други акредитиви, овозможувајќи им на напаѓачите да се претстават како легитимни корисници или да извршат ескалација на привилегии.

### Чести недостатоци:

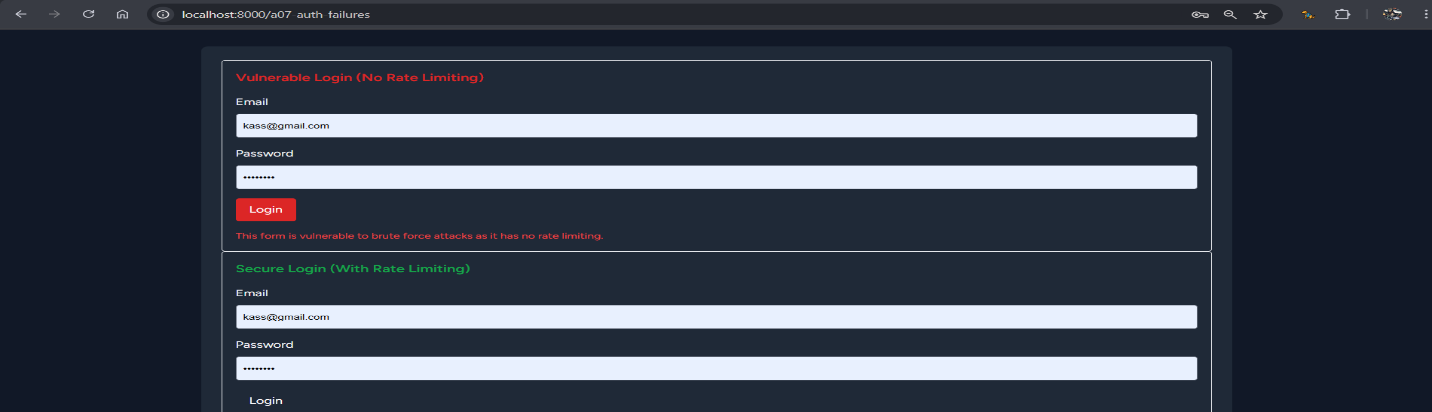
* Брут-форс напади (Brute Force Attacks) – Отсуство на ограничување на обидите за најава, што овозможува напаѓачите да испробаат голем број комбинации на лозинки.
* Слаби лозинки – Користење на едноставни или предвидливи лозинки (на пр., "123456", "password") или недостаток на политики за сложеност.
* Небезбедно складирање на лозинки – Лозинките се чуваат во чист текст или со слаби алгоритми за хеширање (на пр., MD5, SHA-1).
* Незаштитени сесиски токени – Токените за сесии се пренесуваат преку нешифрирани канали или имаат долг век на траење.
* Отсуство на повеќефакторна автентикација (MFA) – Потпирање само на лозинки без дополнителни фактори за верификација.
* Искористување на стандардни акредитиви – Користење на предодредени кориснички имиња и лозинки (на пр., "admin/admin").
* Несоодветна обработка на грешки – Прикажување на премногу детални пораки за грешки (на пр., "Погрешна лозинка" vs. "Погрешно корисничко име или лозинка").

### Како да се справиме?

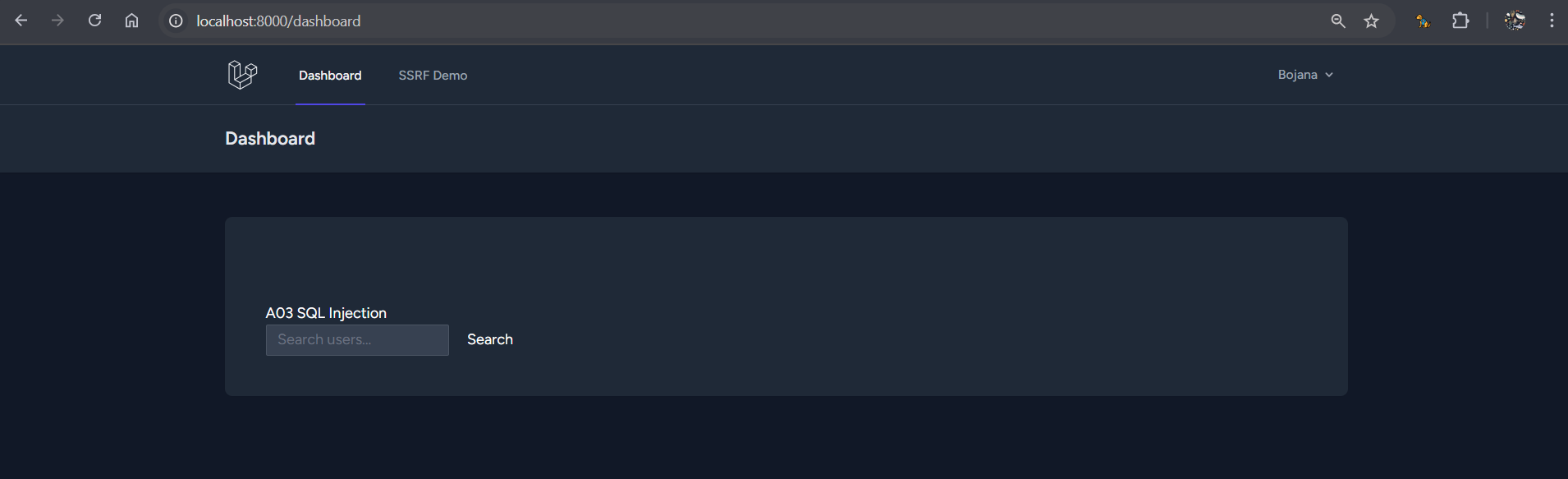
* Имплементирајте ограничување на обитите (Rate Limiting) – Спречете брут-форс напади со ограничување на бројот на обиди за најава (на пр., 5 обиди во рок од 1 минута).
* Присилете ја употребата на силни лозинки –
* Минимална должина од 12 знаци.
* Барајте комбинација на големи/мали букви, бројки и специјални знаци.
* Проверете дали лозинката е компромитирана (преку алатки како Have I Been Pwned).
* Безбедно складирање на лозинки –
  + Користете модерни алгоритми за хеширање (bcrypt, Argon2).
  + Додајте сол (salt) за спречување на rainbow table напади.
* Заштитете ги сесиските токени –
  + Користете HTTPS за сите комуникации.
  + Ограничете го времето на валидност на токените и овозможете нивно поништување при одјава.
* Воведете повеќефакторна автентикација (MFA) –
  + Комбинирајте лозинка со SMS, TOTP (Google Authenticator) или хардверски токени.
* Избегнувајте стандардни акредитиви –
  + Забранете употреба на предодредени кориснички имиња/лозинки.
  + Регуларно проверувајте и бришете тест-сметки.
* Генерички пораки за грешки –
  + Прикажувајте неспецифични пораки (на пр., "Невалидни акредитиви") за да избегнете фазерување (username enumeration).

### Пример

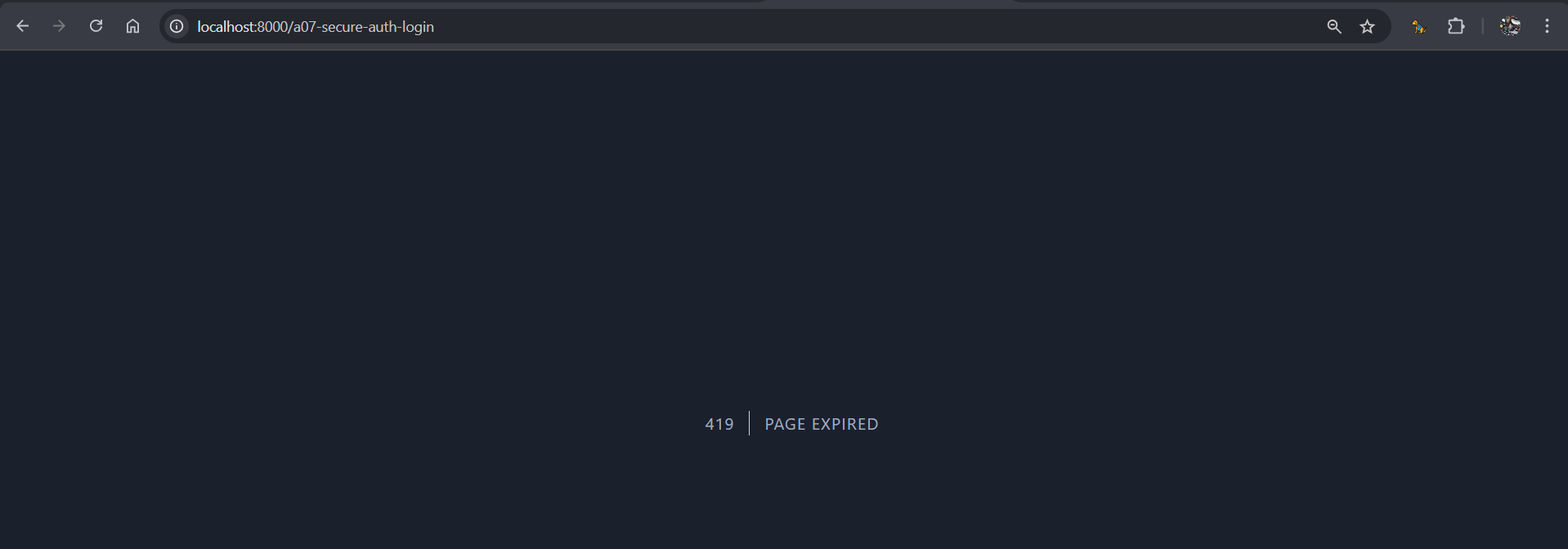
Во демонстрационата апликација, споредбата помеѓу ранливиот и безбедниот пристап е јасно видлива:

1. Ранлив метод (vulnerableLogin) – Нема rate limiting, што овозможува брут-форс напади. Лозинките се споредуваат директно без дополнителни заштити.
2. Безбеден метод (secureLogin) –
   * Користи вградена Laravel функција RateLimiter за спречување на брут-форс.
   * Генерира уникатен клуч за throttling ($throttleKey) врз основа на е-пошта и IP адреса.
   * Го бриши клучот при успешна најава.
3. Регистрација со слаби лозинки (weakPasswordRegister) – Дозволува лозинки со само 4 знаци и ги складира во чист текст.
4. Регистрација со силни лозинки (securePasswordRegister) –
   * Го користи правилото Password::min(8)->mixedCase()->numbers()->symbols()->uncompromised().
   * Лозинките се хеширани пред складирање (Hash::make).

Со извршување на првата форма (неправилно имплементирани механизми за проверка на идентитетот на корисниците ) – пренасочени сме на /dashboard – РАНЛИВО!!



Со извршување на втората форма – системот не ни дозволува пристап на ресурсите.



## A08:2021 – Software and Data Integrity Failures

Ранливостите поврзани со интегритетот на софтверот и податоците настануваат кога претпоставките за интегритетот на софтверот, кодот или критичните податоци се нарушени. Ова вклучува недостаток на проверки за интегритет при ажурирање на софтвер, критични податоци, и CI/CD процеси кои овозможуваат внесување на малициозен код. Една од најголемите закани е кога апликациите преземаат невалидирани, нехеширани или непотпишани компоненти, ажурирања или податоци од недоверливи извори и ги извршуваат.

### Чести недостатоци:

* Непроверени софтверски ажурирања: Апликациите преземаат и извршуваат ажурирања, библиотеки или додатоци од јавни извори без соодветна верификација (на пр., дигитален потпис или безбеден хаш).
* Небезбедна десеријализација: Обработка на сериализирани податоци од недоверливи извори, што може да доведе до инјекција на објекти, далечинско извршување на код (RCE), напади за ускратување на услугата (DoS) или заобиколување на автентикацијата.
* CI/CD Pipeline ранливости: Небезбедни CI/CD (Continuous Integration/Continuous Delivery) процеси кои овозможуваат напаѓачот да инјектира малициозен код во пакети за испорака, изворни репозиториуми или да модифицира конфигурации за изградба.
* Недоволни проверки на интегритетот на податоците: Критичните податоци (пр. банкарски информации, медицински записи) се складираат или пренесуваат без соодветни проверки на интегритетот, како што се хаш или контролни суми (checksums), што може да доведе до нивна незабележана модификација.
* Неправилна употреба на криптографски хашови: Употреба на слаби хаш алгоритми, несоодветни соли (salts) или недостаток на хаширање за чувствителни податоци.
* Застарени/ранливи компоненти: (Иако посебна категорија A06, често се преклопува со интегритетот на софтверот бидејќи ранливите компоненти ја компромитираат целокупната интегритет).

### Како да се справиме?

* Верификација на софтверски ажурирања и компоненти:
  + Секогаш користете дигитални потписи и/или силни криптографски хашови (како SHA-256) за верификација на интегритетот и автентичноста на сите софтверски ажурирања, библиотеки и зависности пред да ги извршите или инсталирате. Ова гарантира дека софтверот не е променет од страна на напаѓач.
  + Преземајте ги компонентите само од доверливи, официјални извори.
  + Присилно користете HTTPS/TLS со соодветна валидација на сертификатот за сите преземања за да спречите MitM (Man-in-the-Middle) напади.
* Избегнувајте десеријализација на недоверливи податоци:
  + Никогаш не десеријализирајте податоци од недоверливи извори. Ако мора да обработувате структурирани податоци од кориснички влез, користете безбедни, стандардни формати како JSON, кои се дизајнирани за размена на податоци и немаат способност за инстанцирање на објекти или извршување на код.
  + Ако десеријализацијата е неизбежна, имплементирајте строга валидација на влезот и ограничете ги класите што може да се десеријализираат (white-listing).
* Обезбедување на CI/CD Pipeline:
  + Имплементирајте строга контрола на пристап до CI/CD алатките и изворните репозиториуми.
  + Користете дигитални потписи за commit-ови во Git за да потврдите кој го направил commit-от.
  + Редовно скенирајте ги контејнерските слики и пакетите за ранливости.
  + Имплементирајте процеси за верификација на интегритетот на изградените артефакти пред распоредување.
* Проверки на интегритетот на податоците:
  + За критични податоци, чувајте ги хаш вредностите заедно со податоците и редовно проверувајте ги за да откриете неовластени измени.
  + Користете автентицирана енкрипција (пр. AES-GCM) при енкриптирање чувствителни податоци, бидејќи таа не само што ги енкриптира податоците, туку и обезбедува нивниот интегритет.
* Имплементирајте механизам за опоравување: Во случај на откриен компромис на интегритетот, имајте подготвен план за брзо враќање на претходна, позната добра состојба.

### Пример

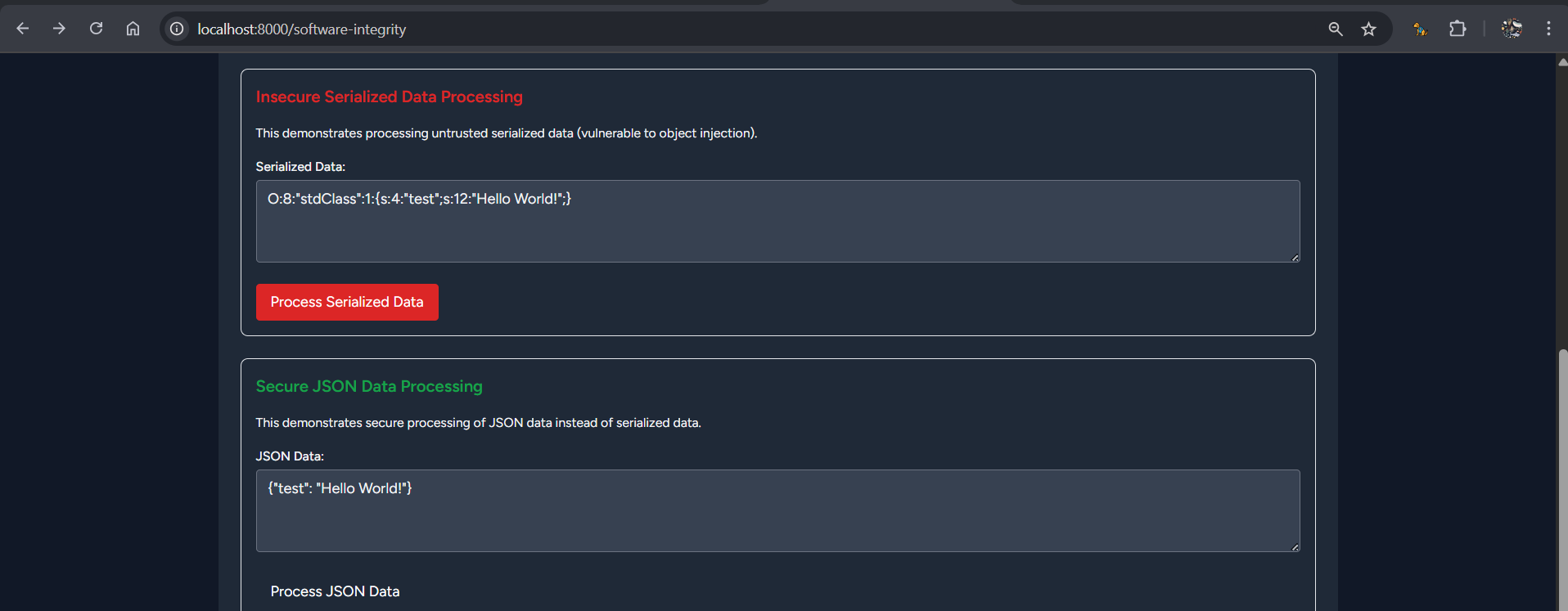
Во Laravel апликацијата, демонстрацијата на A08:2021 Software and Data Integrity Failures ги вклучува следниве примери:

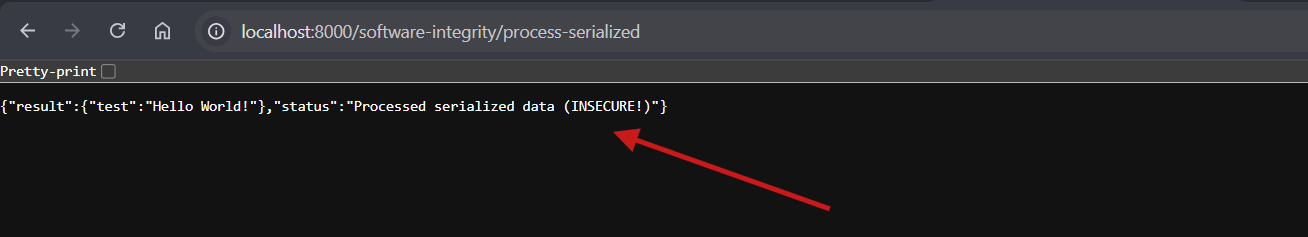
1. Небезбедни софтверски ажурирања:

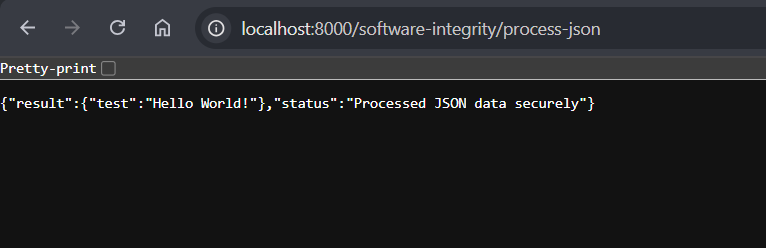
* insecureUpdate() (Ранливо):
  + Ризик: Овој метод дозволува преземање на софтверски ажурирања од било која URL адреса дадена од корисникот ($request->input('update\_url')).
  + Најголема опасност: Користи Http::withoutVerifying() за да го оневозможи SSL верификацијата (што го прави ранлив на MitM) и, уште полошо, директно го извршува преземениот код со eval($updateScript). Ова овозможува далечинско извршување на код ако напаѓачот успее да хостира малициозен скриптен фајл и да ја натера апликацијата да го преземе и изврши.
  + Пример за напад: Напаѓач може да постави сервер со скрипта која брише податоци од базата или инсталира backdoor, а потоа да ја наведе жртвата да ја повика оваа ендпоинт со URL до малициозната скрипта.
* secureUpdate() (Безбедно):
  + Заштита: Овој метод демонстрира правилен пристап со имплементирање на проверки за интегритет и автентичност.
  + Механизми:
    - Користи фиксни, доверливи URL-и за ажурирање и потпис.
    - Вклучува верификација на контролна сума (checksum) користејќи hash('sha256', $updateContent). Ова осигурува дека преземената содржина не е изменета.
    - Вклучува верификација на дигитален потпис ($this->verifySignature()). Во реална апликација, ова би потврдило дека ажурирањето е потпишано од доверлив извор, обезбедувајќи автентичност.
    - Користи Http::withOptions(['verify' => true]) за да обезбеди валидација на SSL сертификатот, заштитувајќи од MitM напади за време на преземањето.
  + Придобивки: Дури и ако напаѓачот успее да пренасочи преземање, проверките на хашот и потписот ќе откријат каква било манипулација, спречувајќи го извршувањето на малициозен код.

2. Небезбедна обработка на сериализирани податоци:

* processSerializedData() (Ранливо):
  + Ризик: Овој метод директно ја користи PHP функцијата unserialize() на кориснички внес ($request->input('serialized\_data')).
  + Опасност: PHP unserialize() е озлогласено ранлива на PHP Object Injection (POI). Напаѓачот може да испрати специјално изработен сериализиран стринг кој, при десеријализација, ќе инстанцира објекти со малициозни својства или ќе повика магични методи (\_\_destruct(), \_\_wakeup() итн.) кои можат да доведат до бришење датотеки, извршување на код или други критични акции.
  + Пример за напад: Напаѓач може да инјектира објект од постоечка класа во апликацијата (на пр. класа за логирање или обработка на датотеки) и да манипулира со нејзините својства за да предизвика несакани несакани ефекти кога објектот е десеријализиран или деструиран.
* processJsonData() (Безбедно):
  + Заштита: Овој метод користи json\_decode() за обработка на структурирани податоци од кориснички внес.
  + Предност: JSON (JavaScript Object Notation) е безбеден формат за размена на податоци кој е дизајниран да носи само податоци, а не логика или објекти што можат да бидат експлоатирани за инјекција. Со json\_decode(), апликацијата ги обработува податоците како што се наменети, без ризик од инстанцирање на малициозни објекти или извршување на скриена логика.
  + Најдобра практика: Заради безбедност и леснотија на интероперабилност, секогаш претпочитајте JSON пред PHP serialize() за обработка на кориснички податоци.







## A09:2021 – Security Logging and Monitoring Failures

Неуспесите во безбедносното евидентирање (logging) и следење (monitoring) се една од најчестите и најкритични безбедносни слабости. Тие директно влијаат на способноста на апликацијата и системот да откриваат, истражуваат и се опоравуваат од безбедносни инциденти. Без соодветно евидентирање и ефективно следење, нападите може да останат незабележани со месеци или дури години, дозволувајќи им на напаѓачите да предизвикаат значителна штета. Оваа категорија се фокусира на пропустите како што се несоодветно евидентирање на клучни настани, чување чувствителни податоци во логовите, недостаток на мониторинг и алармирање, и несоодветна заштита на самите лог-датотеки.

### Чести недостатоци:

* Недоволно или отсутно евидентирање на настани: Неуспех да се евидентираат критични безбедносни настани, како што се неуспешни и успешни најави, промени на привилегии, чувствителни трансакции, грешки при контрола на пристап, промени во конфигурацијата и неуспешни обиди за манипулација. Ова го прави откривањето на напади речиси невозможно.
* Евидентирање на чувствителни податоци: Запишување чувствителни информации (на пр., лозинки, податоци од кредитни картички, лични идентификациски броеви, сесиски токени) директно во лог-датотеките во чист текст или со недоволна заштита. Ова ги прави логовите мета за напад и може да доведе до изложување на податоци.
* Ранливости на инјектирање во лог-датотеки (Log Injection): Директно внесување на кориснички внес во лог-пораките без соодветна санитација. Напаѓачот може да инјектира нови лог-записи или да ги искриви постоечките, со што го отежнува следењето или да маскира злонамерни активности.
* Недостаток на ефективно следење и алармирање: Отсуство на автоматизирани системи за следење кои ги анализираат логовите за сомнителни активности и генерираат аларми во реално време. Рачното прегледување на големи количини логови е неефикасно и неефективно.
* Несоодветна заштита на лог-датотеките: Логовите не се заштитени од неовластен пристап, модификација или бришење. Ова им овозможува на напаѓачите да ги менуваат или уништат доказите за нивните активности.
* Неправилно управување со лог-волумени: Прекумерно евидентирање што резултира со огромни количини на податоци, што го отежнува нивното прегледување и анализа.
* Неуспех да се интегрираат логовите: Недостаток на централизирано собирање логови од сите компоненти на системот, што ја отежнува холистичката анализа на безбедносните инциденти.

### Како да се справиме?

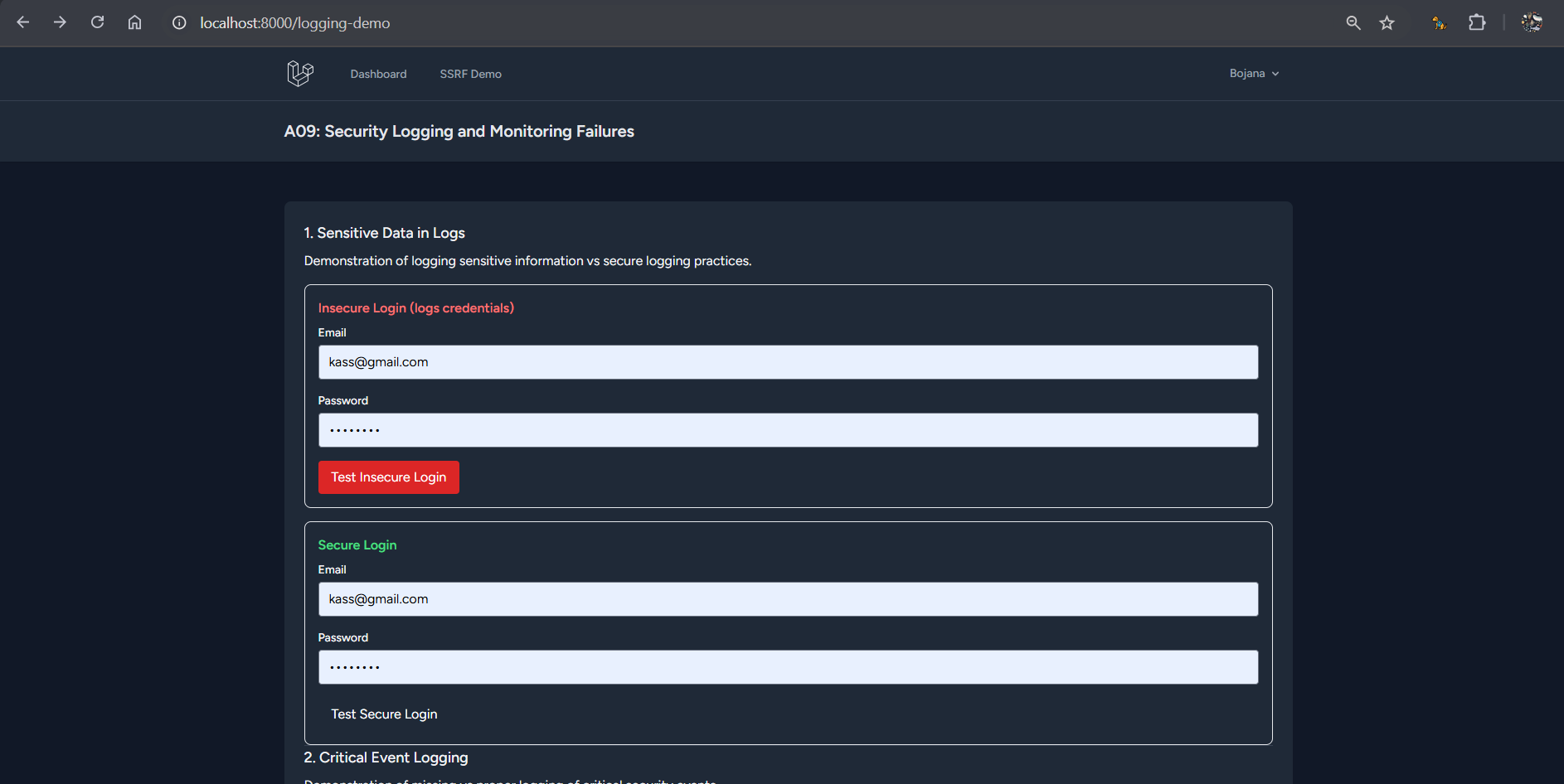
* Евидентирајте ги сите критични безбедносни настани: Снимете ги сите успешни и неуспешни обиди за најава, неуспеси при контрола на пристап, промени на административни дозволи, промени на податоци, грешки на серверот и други значајни настани. Вклучете доволно контекст (корисник, IP адреса, временска ознака, исход).
* Избегнувајте евидентирање на чувствителни податоци: Никогаш не запишувајте лозинки, токени за сесии, лични информации (PII) или други чувствителни податоци во лог-датотеките. Користете техники за маскирање или енкрипција доколку е апсолутно неопходно да се евидентираат делови од чувствителни податоци.
* Санитизирајте го влезот пред евидентирање: Секогаш филтрирајте и санитизирајте ги сите податоци добиени од корисникот пред да ги запишете во лог-датотеките, особено отстранувајќи ги контролните знаци како нови линии и повратни знаци (newlines и carriage returns), за да спречите инјектирање.
* Имплементирајте централизирано евидентирање и следење: Користете SIEM (Security Information and Event Management) или други централизирани алатки за собирање, агрегирање и анализа на логовите од сите компоненти на апликацијата и инфраструктурата.
* Заштитете ја доверливоста, интегритетот и достапноста на логовите:
  + Доверливост: Ограничете го пристапот до лог-датотеките само на овластен персонал. Енкриптирајте ги логовите во мирување и во транзит.
  + Интегритет: Користете криптографски хашови или дигитални потписи за да се осигурате дека логовите не се манипулирани. Чувајте ги логовите на WORM (Write Once, Read Many) медиуми доколку е можно.
  + Достапност: Обезбедете дека логовите се достапни кога е потребно, со соодветни резервни копии и системи за обновување.
* Воспоставете алармирање: Поставете правила за алармирање за сомнителни настани, како што се повеќекратни неуспешни обиди за најава, неовластени пристапи, модификации на критични податоци или активности од невообичаени локации.
* Редовно прегледувајте ги логовите: Дури и со автоматизирано следење, од суштинско значење е редовно да се прегледуваат логовите рачно за аномалии што автоматизираните правила можеби ги пропуштиле.
* Користете соодветни лог-нивоа: Користете стандардни лог-нивоа (DEBUG, INFO, WARNING, ERROR, CRITICAL) за да овозможите лесна филтрација и анализа на логовите.
* Конфигурирајте посебни лог-канали: За критични безбедносни настани, конфигурирајте посебни лог-канали кои се одвоени од општите апликациски логови. Ова го олеснува следењето и заштитата на најважните безбедносни информации.

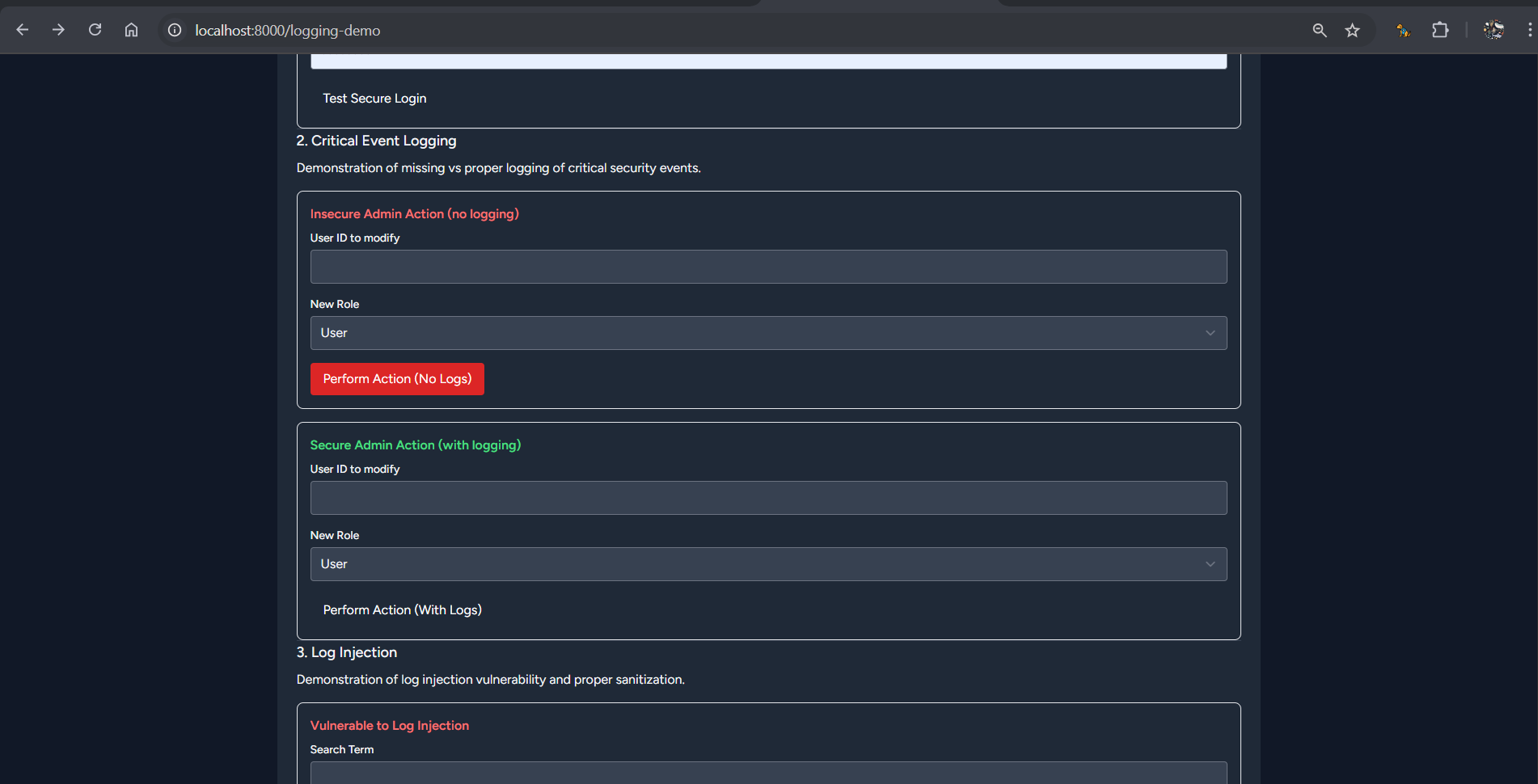
### Пример:

Во Laravel апликацијата, демонстрацијата на A09:2021 ги вклучува следниве примери:

* Логирање чувствителни податоци: Примерот insecureLogin покажува како небезбедно се логираат корисничките акредитиви (email и password) директно во лог-датотеката. Спротивно на тоа, secureLogin демонстративно логира само нечувствителни информации како е-пошта, IP адреса и кориснички агент, како и ID на корисникот по успешна најава, без изложување на лозинката.
* Недоволно/соодветно логирање на критични настани: Методот insecureAdminAction извршува административна акција без никакво логирање, што би го отежнало следењето на злонамерни активности. За разлика од него, secureAdminAction детално логира секоја административна акција со сите релевантни детали, вклучувајќи го ID на администраторот, целниот корисник, IP адресата и направените промени, користејќи посебен security лог-канал.
* Лог инјекција: Методот logInjection покажува како несоодветната санитација на кориснички внес може да дозволи инјектирање на нови линии и лажни лог-записи во лог-датотеката. Примерот safeLogging ја демонстрира правилната санитација со отстранување на контролните знаци пред евидентирањето, спречувајќи ја оваа ранливост.

Овие примери јасно ги илустрираат ризиците од лоши практики и најдобрите практики за евидентирање и следење кои се клучни за робусна безбедносна положба на апликацијата.





## A10:2021 – Server-Side Request Forgery (SSRF)

Server-Side Request Forgery (SSRF) е безбедносна ранливост каде што напаѓачот може да ја натера серверската апликација да испрати барање до дестинација по избор на напаѓачот. Наместо напаѓачот директно да ги таргетира системите на серверот, серверот дејствува како посредник. Ова им овозможува на напаѓачите да пристапат до внатрешни ресурси кои вообичаено не се достапни од надворешната мрежа или да извршат дејства во име на серверот.

### Чести недостатоци:

* Директна употреба на кориснички внес: Апликацијата директно користи URL или дел од URL (како хостнејм или патека) обезбеден од корисникот, без соодветна валидација или санација.
* Недоволна валидација: Недостаток на проверка на внесените URL-адреси за дозволени шеми (e.g., http://, https://), порти или IP адреси.
* Пристап до внатрешни услуги: Успешна SSRF експлоатација може да доведе до пристап до чувствителни внатрешни системи, како што се:
  + Metadata APIs на Cloud провајдери: Откривање на акредитиви, токени и други чувствителни конфигурациски податоци на серверот.
  + Внатрешни REST APIs или бази на податоци: Пристап до податоци или извршување на операции во други внатрешни сервиси.
  + File shares или интранет апликации: Преземање или манипулација со внатрешни датотеки и ресурси.
* Port Scanning на внатрешни мрежи: Напаѓачите можат да ја користат ранливата апликација за да скенираат порти на внатрешни сервери, идентификувајќи активни услуги и потенцијални цели за понатамошни напади.
* Blind SSRF: Во некои случаи, напаѓачот не добива директен одговор од барањето, но може да заклучи дали нападот бил успешен врз основа на временски задоцнувања или други индиректни индикатори.

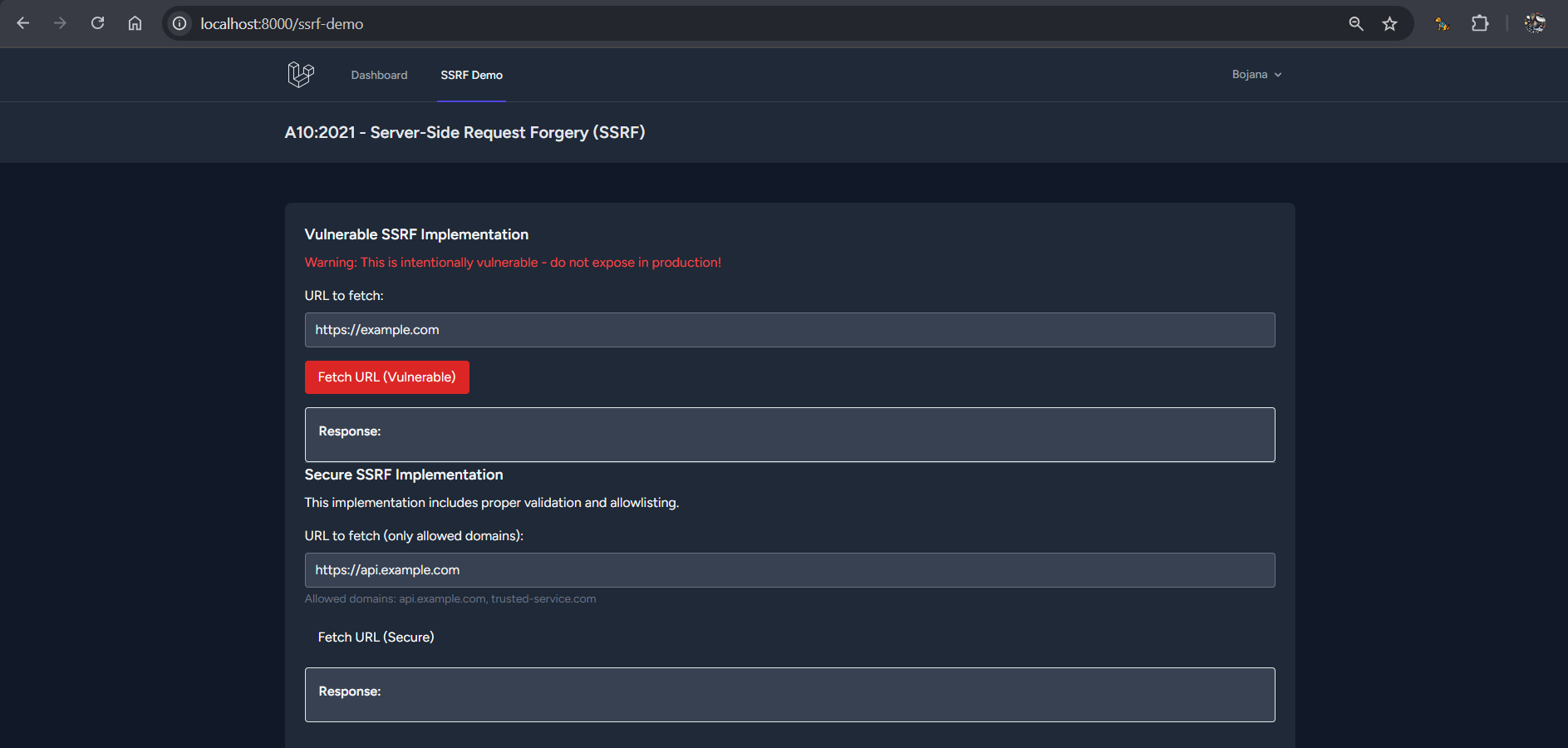
### Како да се справиме?

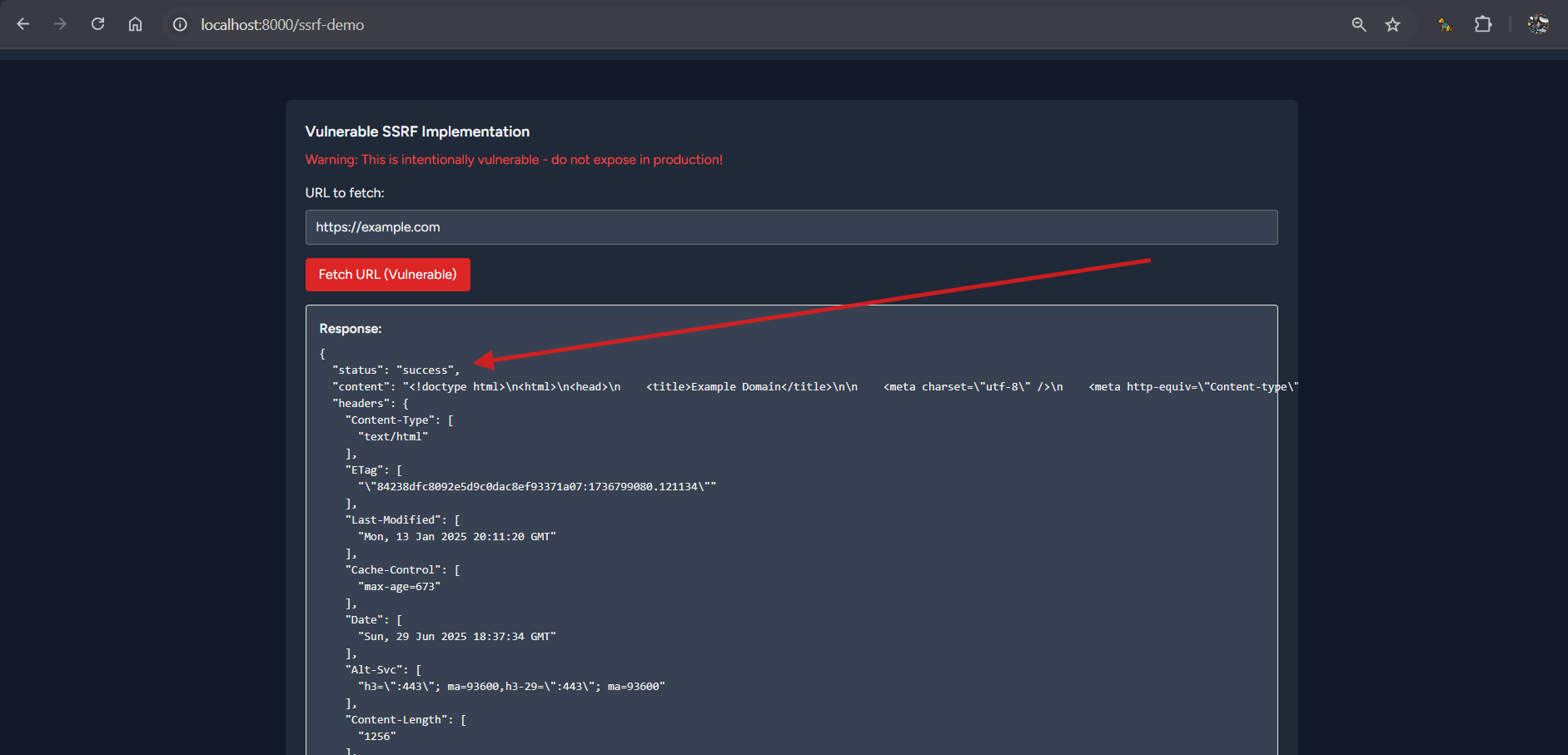
* Validate & Sanitize Input:
  + Секогаш валидирајте го корисничкиот внес за URL-адреси од страна на серверот. Користете вградени функции или библиотеки за валидација на URL-адреси.
  + Санитизирајте ги сите компоненти на URL-адресата (scheme, host, port, path) за да се осигурате дека тие се во очекуван формат и да спречите инјектирање на несакани знаци.
* Allowlisting (Бела листа) на домени и IP адреси:
  + Дефинирајте строга листа на дозволени домени и/или IP адреси до кои апликацијата смее да прави барања. Секој домен или IP адреса што не е на белата листа треба да биде експлицитно одбиен.
  + Избегнувајте blocklisting (црна листа) бидејќи може лесно да се заобиколи (на пр., со користење на алтернативни IP формати, DNS записи или пренасочувања).
* Оневозможете пренасочувања (Redirects):
  + Конфигурирајте го клиентот за HTTP барања на серверот да не ги следи автоматски пренасочувањата. Доколку се потребни пренасочувања, проверете ја дестинацијата на пренасочувањето во однос на белата листа пред да ја дозволите.
* Ограничување на протоколи и порти:
  + Дозволувајте само безбедни протоколи како http:// и https://. Оневозможете протоколи како file://, gopher://, ftp:// кои можат да се користат за пристап до локални датотеки или други услуги.
  + Ограничете ги дозволените порти (на пр., само 80 и 443 за веб-услуги) за да спречите скенирање на порти на внатрешни сервери.
* Имплементирајте тајмаути (Timeouts):
  + Поставете разумни тајмаути за HTTP барањата за да спречите напади кои ја блокираат апликацијата (denial-of-service) или да спречите премногу долго чекање за одговор од нереагирачки внатрешни услуги.
* Користете специфични библиотеки:
  + Пристапете до URL-адресите користејќи библиотеки или функции кои се дизајнирани со безбедност на ум и кои овозможуваат лесна конфигурација на горенаведените безбедносни контроли (како Http фасадата во Laravel).
* Log & Monitor:
  + Евидентирајте ги сите обиди за SSRF, дури и оние блокирани, за да можете да ги идентификувате потенцијалните напади и да ја прилагодите вашата безбедносна политика.

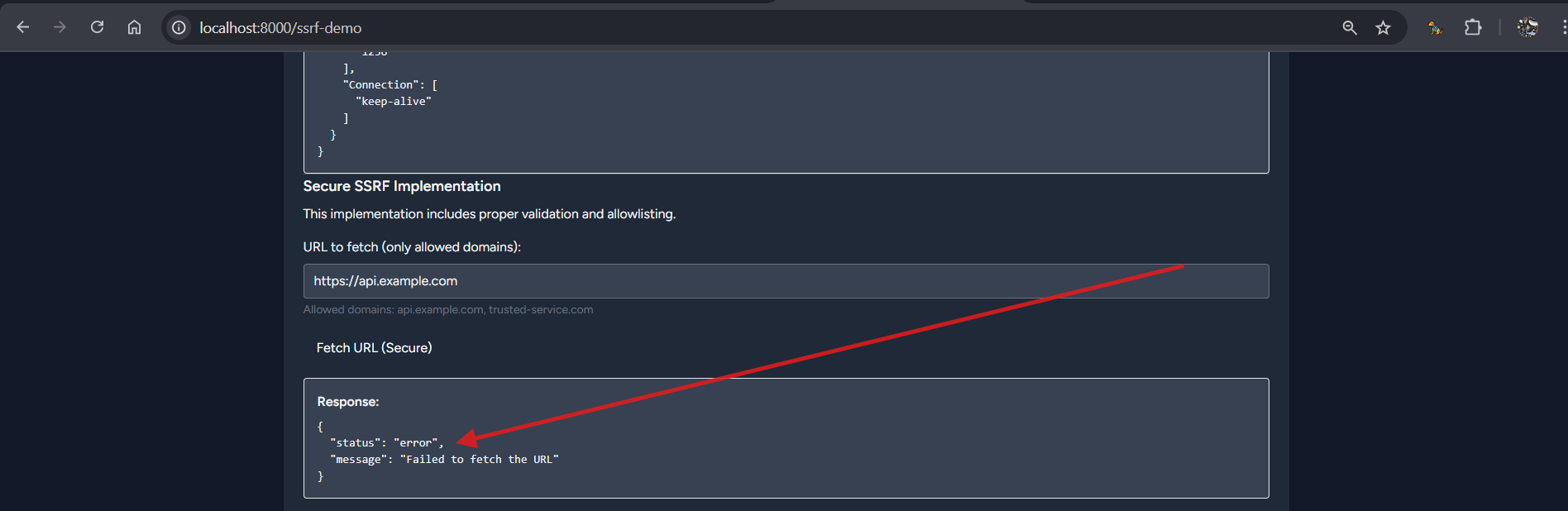
### Пример:

Во Laravel апликацијата, демонстрацијата на A10:2021 ги вклучува следниве примери:

* Vulnerable SSRF Endpoint (fetchUrl): Овој ендпоинт го зема URL-то директно од корисничкиот внес и прави HTTP барање без никаква валидација или контрола. Ова овозможува напаѓачот да испраќа барања до која било URL адреса што серверот може да ја достигне, вклучувајќи ги и внатрешните ресурси (e.g., http://localhost, file:///etc/passwd, http://169.254.169.254/latest/meta-data/).
* Secure SSRF Endpoint (fetchUrlSecure): Овој ендпоинт покажува како да се справите со SSRF со имплементирање на следниве безбедносни мерки:
  + Валидација на URL: Laravel-овата валидација required|url се користи за да се осигура дека влезот е валидна URL адреса.
  + Домен Allowlisting: Се проверува дали хост доменот на бараниот URL е дел од претходно дефинирана листа на дозволени домени (api.example.com, trusted-service.com). Барањата до други домени се одбиваат.
  + Ограничување на порти: Дозволени се само стандардни HTTP/HTTPS порти (80 и 443).
  + Тајмаут: Барањето има поставено тајмаут од 5 секунди за да се спречат долги чекања.
  + SSL верификација: Се осигурува дека SSL сертификатите се валидирани, додавајќи уште еден слој на безбедност.







## Евалуација на трети решенија за скенирање ранливости во код и нивна ефикасност

Дури и со најдобри намери, програмерите може да пропуштат ранливости во нивниот код. Поради таа причина,постојат скенери за апликации кои автоматски откриваат безбедносни недостатоци. Тие помагаат да се идентификуваат погрешни конфигурации и застарени зависности на OWASP што би можеле да бидат искористени од напаѓачите. Со користење на овие скенери, тимовите можат проактивно да пронајдат и поправат потенцијални слабости пред тие да станат сериозен проблем.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| АЛАТКА | ПРЕДНОСТИ | НЕДОСТАТОЦИ | ИНСТАЛАЦИЈА | НАЈДОБАР СЛУЧАЈ ЗА УПОТРЕБА |
| Enlightn (Laravel-Specific) | Скенер за безбедност и перформанси специфичен за Laravel.  Детектира над 150 типови на ранливости, вклучувајќи проблеми со SQLi, XSS, CSRF  Проверува за погрешни конфигурации специфични за Laravel. | Се фокусира само на  Laravel/PHP екосистеми. | * composer require enlightn/enlightn --dev * php artisan enlightn | Laravel апликации |
| OWASP ZAP  (Dynamic Application Security Testing (DAST) tools) | Симулира напади врз вашата апликација што работи преку HTTP.  Детектира SQL инјекција, XSS, CSRF, погрешни безбедносни конфигурации | * Нема пристап до изворниот код. * Потребна е работна околина (на пр., Docker) * Потребна е експертиза за толкување на резултатите | docker run -it owasp/zap2docker-weekly or use Desktop GUI. | Тестирање на веб-апликации за време на извршување од перспектива на напаѓач |
| Snyk | Одлично за познати CVE во пакети и статички проблеми како SQLi.  Идентификува застарени, ранливи пакети во composer.lock | * Помалку фокусиран на ранливостите на апликацискиот код. * Потребна е интеграција во облак за да може да се искористат сите функции. | 1. npm install -g snyk 2. snyk auth 3. snyk test. | Брза CI интеграција; открива ранливи зависности. |
| SonarQube | Сеопфатно статичко тестирање на безбедноста на апликациите (SAST).  Комерцијално и платформа за анализа на статички код со отворен код | * Троши многу ресурси * Долго време на стартување * Потребна е конфигурација за интеграција. | docker run -d -p 9000:9000 sonarqube:9-community | In-house security/code quality enforcement. |
| PHPStan | Статичка анализа со правила фокусирани на безбедноста.  Lightweight, rule-based, and IDE-friendly | * Потребно е подесување за да се избегнат лажни позитиви * Нема анализа за време на извршување – Нема да открие проблемии што доаѓаат од динамички барања | composer require --dev phpstan/phpstan phpstan/extension-installer phpstan-security/phpstan-security | Открива вообичаени грешки, грешки во типот и небезбедни шеми на код.  Предупредува за проблеми со масовно доделување и некои небезбедни шеми на дизајн |

## БРИШЕЊЕ НА ДЕФИНИРАНАТА ОКОЛИНА

Се` што треба да направите е да ги стопирате Докер контејнерите со следнава команда извршена во терминал: docker compose down –volumes

## ЗАКЛУЧОК

Овој елаборат детално ги претстави најкритичните безбедносни ризици за веб-апликациите според OWASP Топ 10, со посебен фокус на нивната демонстрација и превенција во Laravel околина. Разгледавме различни ранливости, од проблеми со контрола на пристап и криптографски пропусти, преку инјекции и небезбеден дизајн, до слаби компоненти, проблеми со автентикација, нарушувања на интегритетот на податоците, и неуспеси во логирањето и следењето, па сè до SSRF напади.

Клучното учење од овој проект е дека безбедноста на апликациите не е еднократна задача, туку континуиран процес кој бара интегриран пристап во целиот животен циклус на развојот. Laravel, како робустен PHP framework, нуди многу вградени заштити, но тие не се доволни без свесност, најдобри практики и прецизна имплементација од страна на програмерите.

Со практично демонстрирање на овие ранливости, проектот има за цел да ја подигне свеста и да ги опреми програмерите со знаењето и алатките потребни за изградба на побезбедни веб-апликации. Занемарувањето на овие безбедносни принципи може да има катастрофални последици, вклучувајќи загуба на податоци, финансиска штета и нарушување на угледот. Затоа, инвестирањето во безбеден дизајн, кодирање и следење е од суштинско значење за заштита на дигиталниот екосистем.

Се надеваме дека овој проект ќе послужи како вреден ресурс за сите кои сакаат да ја продлабочат својата експертиза во етичкото хакирање и безбедноста на веб-апликациите.