Универзитет „Св. Кирил и Методиј“ во Скопје
ФАКУЛТЕТ ЗА ИНФОРМАТИЧКИ НАУКИ И КОМПЈУТЕРСКО ИНЖЕНЕРСТВО

Дополнителни активности за проект по предметот

Бази на податоци

Анализа на безбедност на апликација и база - Crime Tracker

Љубица Ристовска 212016

Скопје, 27.08.2024

Целта е да се анализираат следните теми, да се тестира апликацијата Crime tracker и да се направи рефакторирање на кодот соодветно за темата на анализа.

Вовед

Безбедноста на податоците е една од најважните работи што треба да се испланираат при креирање на база на податоци за некоја институција. Бидејќи проектот е за менаџирање на документи, односно криминални случаи, податоците треба да се заштитени од надворешен пристап, дополнително заштитени и според пермисии на пристап во однос на тоа дали ги пристапува полицаец или началник. Целта на дополнителниот дел на овој проект е да се анализираат сите можни аспекти на бази на податоци поврзани со безбедноста.

Главен дел

Чување на лозинки во база и начин на автентикација

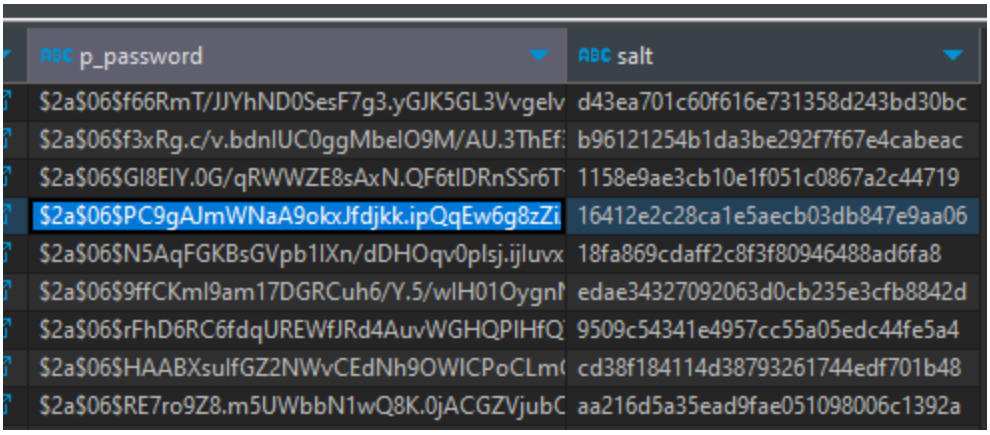
Како почеток апликацијата ни дава login screen. Тука се променети три нови работи со цел зголемување на безбедноста.

1. При најава, односно внесување на број на значка и лозинка, апликацијата доколку има грешка не посочува дали е конкретно за корисничкото име(значката) или за лозинката. Ова е со цел апликацијата да не открие кои кориснички имиња(значки) се валидни. Пораката што ја прикажува е „Невалидни креденцијали“.
2. Многу е важно лозинките во база да не се чуваат како ‘cleartext’ односно треба да се чуваат хеширани.
3. Доколку двајца полицајци имаат иста лозинка доколку се хешираат добиваме исти хеш. Ова е небезбедно и со цел да нема исти хеш чуваме и ‘salt’ за да исти лозинки имаат различни хешови.

Со цел да се постигне ова откако веќе се извршени скрипта за додавање на податоци, можеме да направиме скрипта што додава нова колона ‘salt’, генерира random стрингови и

ги внесува како податок. Потоа лозинките што се во cleartext ги земаме, хешираме и ги поставуваме како лозинка. Скриптата е следнава:

```
DO $$
DECLARE
  policeman_record RECORD;
  generated_salt TEXT;
  hashed_password TEXT;
BEGIN
  FOR policeman_record IN SELECT pe_id, badge_no, p_password FROM policeman LOOP
    -- Generate a random salt
    generated_salt := md5(random()::text);
    -- Hash the password with the salt using the crypt function
    hashed_password := crypt(policeman_record.p_password || generated_salt, gen_salt('bf'));
    -- Update the policeman's password and salt in the database
    UPDATE policeman
    SET p_password = hashed_password,
      salt = generated_salt
    WHERE pe_id = policeman_record.pe_id;
  END LOOP;
END $$;
-- Encrypt passwords for the officer table
DO $$
DECLARE
  officer_record RECORD;
  generated_salt TEXT;
  hashed_password TEXT;
BEGIN
  FOR officer_record IN SELECT pe_id, o_badge_no, o_password FROM officer LOOP
    -- Generate a random salt
    generated_salt := md5(random()::text);
    -- Hash the password with the salt using the crypt function
    hashed_password := crypt(officer_record.o_password || generated_salt, gen_salt('bf'));
    -- Update the officer's password and salt in the database
    UPDATE officer
    SET o_password = hashed_password,
      salt = generated_salt
    WHERE pe_id = officer_record.pe_id;
  END LOOP;
END $$;
```



p_password	salt
$2a$06$f66RmT/JJYhND0SesF7g3.yGJK5GL3Vvgelv	d43ea701c60f616e731358d243bd30bc
$2a$06$f3xRg.c/v.bdnlUC0ggMbelO9M/AU.3ThEf:	b96121254b1da3be292f7f67e4cabeac
$2a$06$GI8EIY.0G/qRWWZE8sAxN.QF6tIDRnSSr6T	1158e9ae3cb10e1f051c0867a2c44719
$2a$06$PC9gAJmWNaA9okxJfdjkk.ipQqEw6g8zZi	16412e2c28ca1e5aecb03db847e9aa06
$2a$06$N5AqFGKBsGVpb1IXn/dDHOqv0plsj.ijluvx	18fa869cdaff2c8f3f80946488ad6fa8
$2a$06$9ffCKml9am17DGRCuh6/Y.5/wlH01OygnI	edae34327092063d0cb235e3cfb8842d
$2a$06$rFhD6RC6fdqUREWfJRd4AuvWGHQPIHfQ	9509c54341e4957cc55a05edc44fe5a4
$2a$06$HAABXsulfGZ2NWvCEdNh9OWICPoCLm(	cd38f184114d38793261744edf701b48
$2a$06$RE7ro9Z8.m5UWbbN1wQ8K.0jACGZVjubC	aa216d5a35ead9fae051098006c1392a

Хеширани лозинки и нивен salt

Кодот за автентикација на апликацијата е соодветно променет за новиот начин на чување на лозинките. Промените се во SessionsController.

Пристап до views соодветни за привилегиите

Најдобриот начин за поставување пермисии е користење на дополнителна табела или колона со пермисии во база. Бидејќи нашата база е веќе поделена на полицајци и началници наместо да чуваме дополнителна колона можеме да го искористиме тоа дали е најавен полицаец или началник и според тоа да им се врати соодветниот view. Тука е важно да се напомене дека доколку корисникот не е автентициран не треба да има пермисии до сите страни освен до login страната.

Во секоја рамка(framework) ова би се извршило на различен начин. Во Laravel постои можност за дефинирање на middleware што ќе се додаде на рутите. Според сценаријата од фаза 3 (дефинирање на UseCase модели ги одредуваме кои страни се за автентициран полицаец, кој страни за автентициран началник и на крај и за двете.

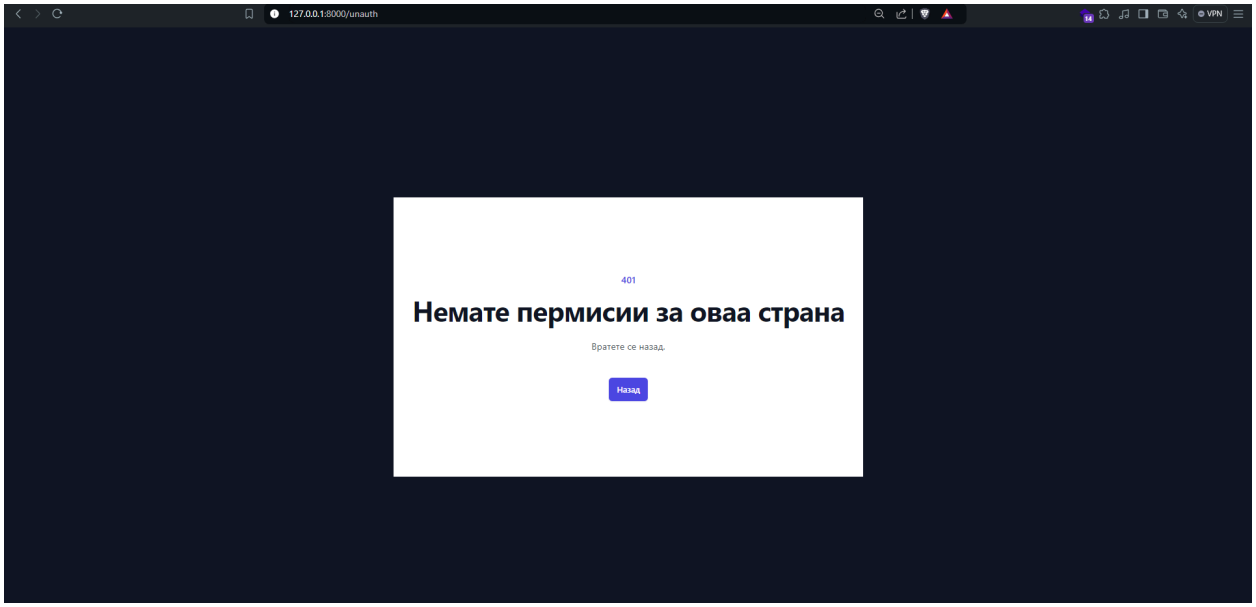
```
protected $routeMiddleware = [
    // other middleware
    'policeman' => \App\Http\Middleware\CheckPoliceman::class,
    'officer' => \App\Http\Middleware\CheckOfficer::class,
    'both' => \App\Http\Middleware\CheckBoth::class,
];
```

Дефинирање на рути за middleware

```php
// UNAUTHORIZED
Route::get( uri: '/login', function () {
    return view( view: 'login');

});
Route::post( uri: '/login', [SessionsController::class, 'store']);

Route::get( uri: '/unauth', function () {
    return view( view: 'unauth');  // Make sure there is a view file named `unauth.blade.php`
})->name( name: 'unauth');  // Name the route 'unauth'

// AUTHORIZED
// POLICEMAN
Route::get( uri: 'register-statement', [CrimeCaseController::class, 'register_statement'])->middleware( middleware: 'policeman');
Route::post( uri: 'register-statement', [CrimeCaseController::class, 'register_statement_post'])->middleware( middleware: 'policeman');

// OFFICER
Route::get( uri: 'register-policeman', [OfficerController::class, 'register'])->middleware( middleware: 'officer');
Route::post( uri: 'register-policeman', [OfficerController::class, 'register_post'])->middleware( middleware: 'officer');

// BOTH
Route::get( uri: '/', function () {
    return view( view: 'welcome');
})->middleware( middleware: 'both');
Route::get( uri: 'logout', [SessionsController::class, 'logout']);

Route::get( uri: 'employees', [OfficerController::class, 'employees'])->middleware( middleware: 'both');
Route::get( uri: '/employees/{id}', [OfficerController::class, 'show'])->middleware( middleware: 'both');

Route::get( uri: 'filter', [PeopleController::class, 'filter'])->middleware( middleware: 'both');
Route::post( uri: 'filter', [PeopleController::class, 'filter_post'])->middleware( middleware: 'both');

Route::get( uri: 'cases', [CrimeCaseController::class, 'cases'])->middleware( middleware: 'both');
Route::get( uri: 'case/{wildcard}', [CrimeCaseController::class, 'case'])->middleware( middleware: 'both');
Route::get( uri: 'finished_cases', [CrimeCaseController::class, 'finished_cases'])->middleware( middleware: 'both');

Route::post( uri: '/get-person', [PeopleController::class, 'getPerson'])->middleware( middleware: 'both');
```

Дефинирање на рути

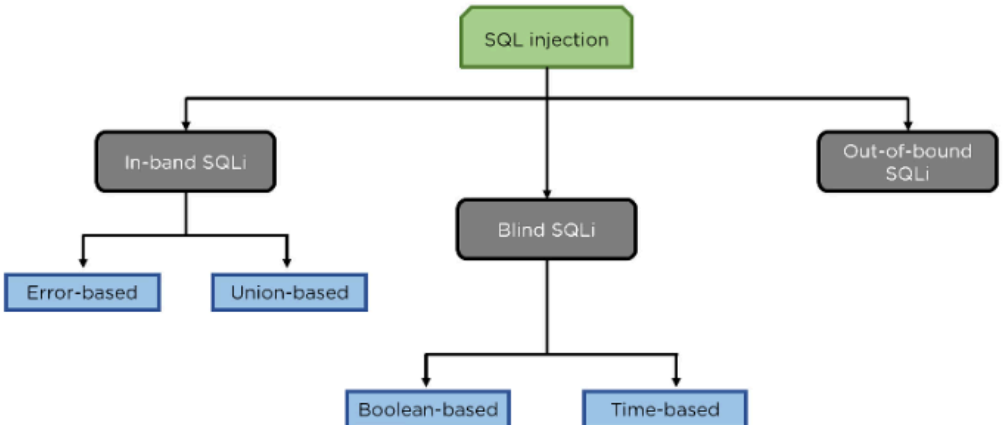
Еден пример би бил доколку се најавиме како полицаец и пробаме да пристапиме register-policeman што е view само за началник ја добиваме следната страна.



Пристап до страна до која немаме пермисии

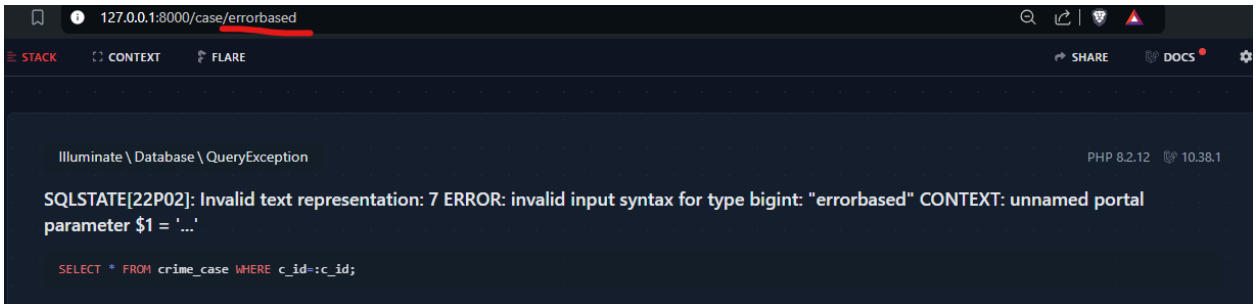
Тестирање на можни SQL инјекции

SQL инекција е еден од топ 10 напади на OWASP top 10 при тестирање на апликации. Ова се должи на тоа како програмерот ги превзема податоците од база, и доколку неговите querries не се санитизирани може да доведе до ‘leak’ на податоци. Постојат повеќе типови на SQL инјектирање, ние ќе ги провериме сите за нашата апликација.



Типови на SQL инјектирање

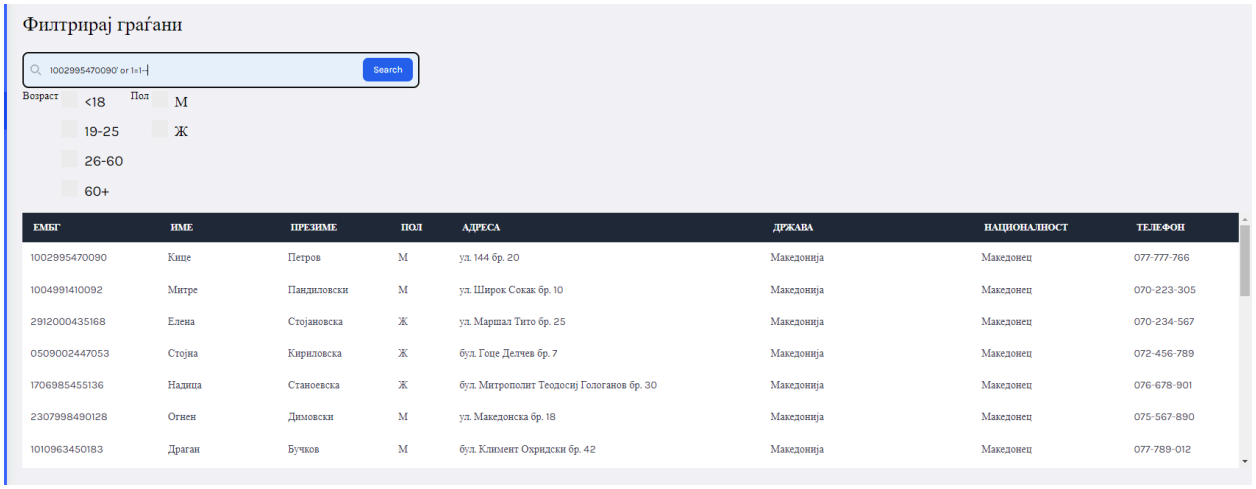
In-band означува кога на вебсајтот земаме податоци од истиот тој вебсајт. Error based е кога правиме грешен request до нешто и дознаваме податоци од error’от. Овој напад во нашата апликација е успешен каде што во error-от го враќа прашалникот и тоа каков параметар е зачуван c_id во базата.



Успешен Error-Based Attack

Union-based се кога пробуваме со манипулација на наводниците да го проширime прашалникот и без разлика што е внесено се да ни врати. Во случајов нигде немам union-based успешен напад освен кај опцијата за филтрирање на граѓани. Тука ни ги враќа сите граѓани иако е внесен еден матичен. Сепак секако овој view овозможува пристап на

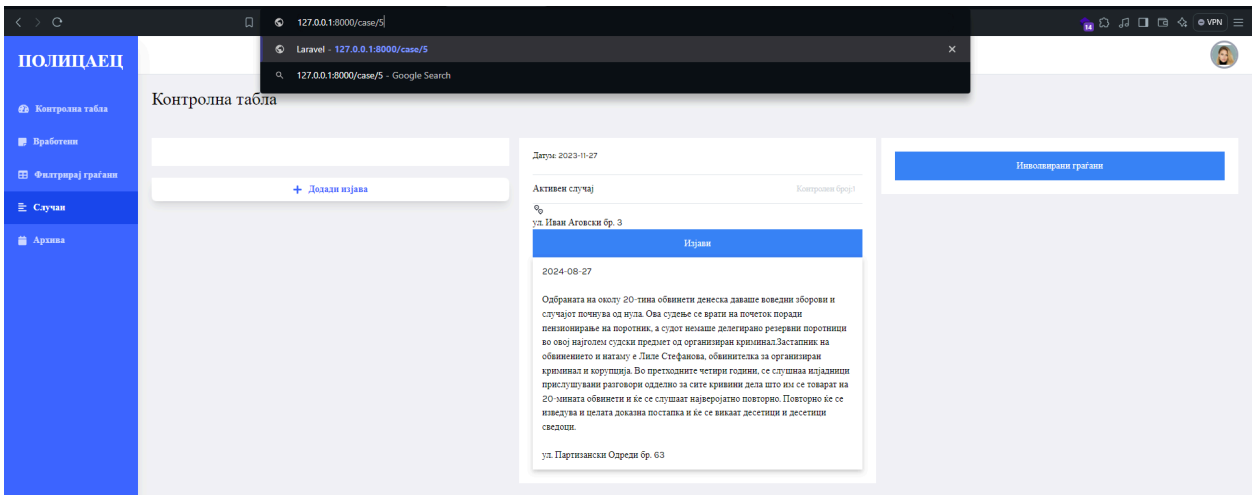
полицаецот до сите граѓани така што не претставува никаква безбедносна закана.



ЕМБГ	ИМЕ	ПРЕЗИМЕ	ПОЛ	АДРЕСА	ДРЖАВА	НАЦИОНАЛНОСТ	ТЕЛЕФОН
1002995470090	Кице	Петров	М	ул. 144 бр. 20	Македонија	Македонец	077-777-766
1004991410092	Митре	Пандиловски	М	ул. Широк Сокак бр. 10	Македонија	Македонец	070-223-305
2912000435168	Елена	Стојановска	Ж	ул. Маршал Тито бр. 25	Македонија	Македонец	070-234-567
0509002447053	Стојна	Кириловска	Ж	бул. Гоце Делчев бр. 7	Македонија	Македонец	072-456-789
1706985455136	Надица	Станоевска	Ж	бул. Митрополит Теодосиј Гологанов бр. 30	Македонија	Македонец	076-678-901
2307998490128	Огнен	Димовски	М	ул. Македонска бр. 18	Македонија	Македонец	075-567-890
1010963450183	Драган	Бучков	М	бул. Климент Охридски бр. 42	Македонија	Македонец	077-789-012

Успешен union based attack

Друг проблем што го овозможува SQL прашалникот е началници да пристапат до случаи што не се од нивната станица со внесување на case/било-кој-број.



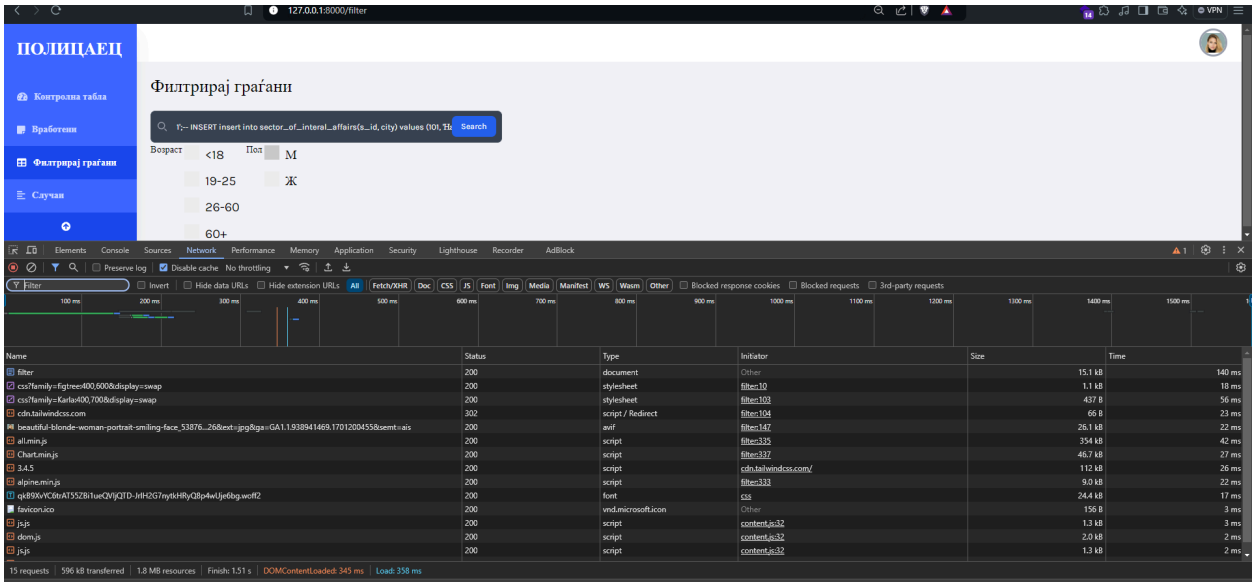
Неавторизиран пристап до случај

Out-of-band е кога ги добиваме податоците од друг комуникациски канал, пример од некој сервер. Во оваа апликација таков напад не е возможен.

Blind sqli претставува кога немаме повратни податоци дали е успешно, тие промени би биле извршени директно во база. Пример инјектирање на нови корисници(полицаец или началник). Најчесто за да се најде дали постои тој корисник во база и ако постои потоа се бара лозинката, тоа би зафатило повеќе време од само не наоѓање на корисничкото име. Со цел да се дознае дали навистина постои тоа корисничко име може да се направат напади базирани врз мерење на времето на повратниот одговор од база.

Пример доколку на местото за филтрирање граѓани внесеме 1';-- INSERT insert into sector_of_interal_affairs(s_id, city) values (101, 'Напад'); барањето е успешно и назад не

добиваме повратен одговор. Од страна на хакери ова би значело дека нападот можно е да е успешен. Сепак ние имаме пристап до база и можеме да видиме дека нападот не е успешен, односно не е додаден нов град во sector_of_interal_affairs. Со тоа заклучуваме дека Blind sql не е возможен за додавање податоци.



Blind sql

Пример за како би се решиле SQL инјекциите е користење на query builder од самата рамка во случајов Laravel. Важно е тука да се напомене дека за сложени прашалници може query builder да ги постави грешно.

Пример:

```
$police_station = DB::select('select * from police_station where p_id=:p_id;',['p_id'=>Session::get('p_id')]);
```

Го заменуваме со

```
$police_station = DB::table('police_station')
   ->where('pe_id', Session::get('pe_id'))
   ->get();
```

Logging на пристапите до датабаза

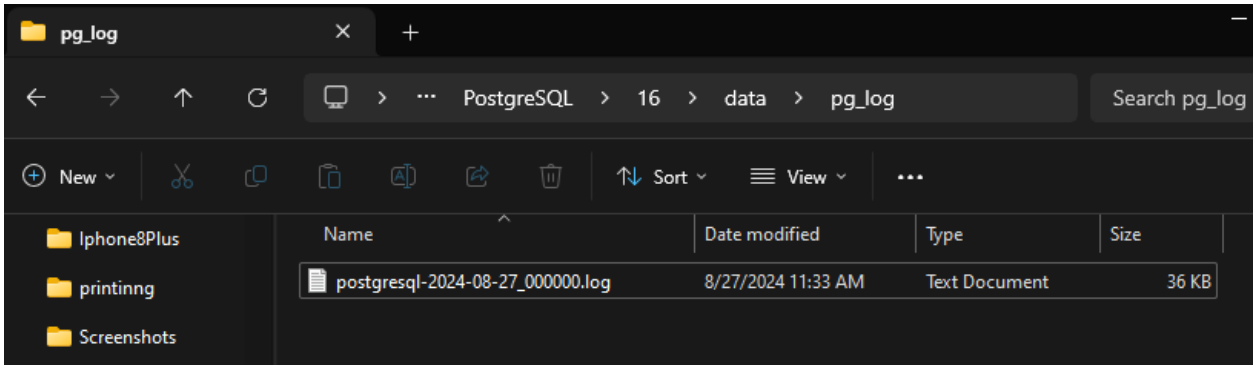
Logging на пристапот до базата на податоци е клучна мерка за безбедносни цели. Неколку причини зошто е важно да се има logging на пристапот до базата на податоци:

Детекција на неовластен пристап: Најчесто логовите имаат timestamp и тоа овозможува во реално време да се детектираат потенцијални закани

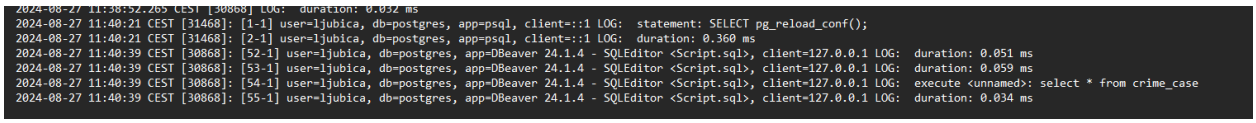
Истражување на инциденти: Кога ќе се случи безбедносен инцидент, логовите на пристапот до базата на податоци можат да бидат искористени како доказ за истражување на инцидентот.

Заштита од интерни закани: Доколку некој сака да ги избрише податоците за кои нема овластување да ги брише или да прочита податоци до кои не би требало да има пристап.

За нашата база можеме да овозможиме да се логира во "C:\Program Files\PostgreSQL\16\data\postgresql.conf" со поставување на лог параметри во дадотеката. За да се зачуваат промените може преку Comand Promt да ја извршиме командата SELECT pg_reload_conf();. Со тоа се создава фајл, во случајот го именував како pg_log и ги зачувува логовите.



Некогаш логовите се потребни во дигитална форензика, еве како би изгледал лог за пристап до нашата табела crime_case пристапена преку DBeaver.



Заштита на backup на податоците во базата

Бекап е потребно за заштита од губење на податоците при технички проблеми или грешки во софтверот. Дополнително доколку има некој вирус, малвер или било каков хакерски напад со цел да не се загубат податоците потребно е да се прави редовен бекап.

Постојат повеќе начини на бекап на подататоци на база, најчесто на екстерни сервери или пак на backup на облак. Сепак чување на податоците на облак е скапо и се плаќа. За демонстрација да замислиме дека правиме бекап на AWS cloud, тоа може да автоматизираме со користење на скрипта и истата таа скрипта да се повикува на одреден временски интервал со креирање на Cron Job.

```
#!/bin/bash
```

```
TIMESTAMP=$(date +%F-%H-%M-%S)
BACKUP_FILE="crimetracker-$TIMESTAMP.backup"
pg_dump -U postgres -F c postgres > $BACKUP_FILE
aws s3 cp $BACKUP_FILE s3://my-bucket/backups/$BACKUP_FILE
rm $BACKUP_FILE
-> Cron Job на секој два часа би изгледал вака c0 2 * * *
```

Сите Cloud providers имаат различна онлајн документација како да се направи бекап, но генерално за сите е ваква замислата.

Заклучок

Анализиравме најпрво зошто треба лозинките во база да се чуваат хеширани и зошто треба истите да се хешираат со користење на salt. Со тоа направивме и рефакторирање на кодот за најавување на полицаец и началник.

Исто така при рефакторирање беше додадено правилен пристап до views во однос дали е најавен полицаец или началник. Тука беше напоменето дека во идеална ситуација ролјите би се чувале како посебна колона или табела во база.

Потоа тестиравме повеќе видови на SQL инјекции врз базата од кои некои успешни, а некои неуспешни. Со цел да се спречат успешните SQL инјекции направивме санитизирање на кодот во секој можен контролер каде што беше можен SQLi напад што е значаен.

Логирањето на пристапите до базата ни беше клучно за детекција на неовластен пристап или пак за истражување инциденти при хакерски напади. Со континуирано следење на логовите може да се открие и грешка при прашалници во базата.

Како за на крај ставивме акцент на значење то на бекапите на податоци во базите. Поради тоа што не е можна демонстрација за нашата апликација, прикажавме како тоа би било доколку во замислена ситуација се користи AWS како бекап cloud провајдер.

Од ова можеме да заклучиме дека не само добро треба да се осмисли како ќе ги чуваме податоците во база туку и како истите би требало да се заштитат да не дојде до нивно крадење, бришење, модифицирање и слично, посебно во некоја апликација со сензитивни податоци како што е Crime Tracker.