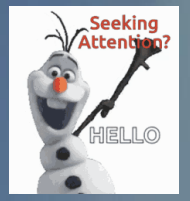
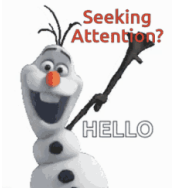


SQL Injection

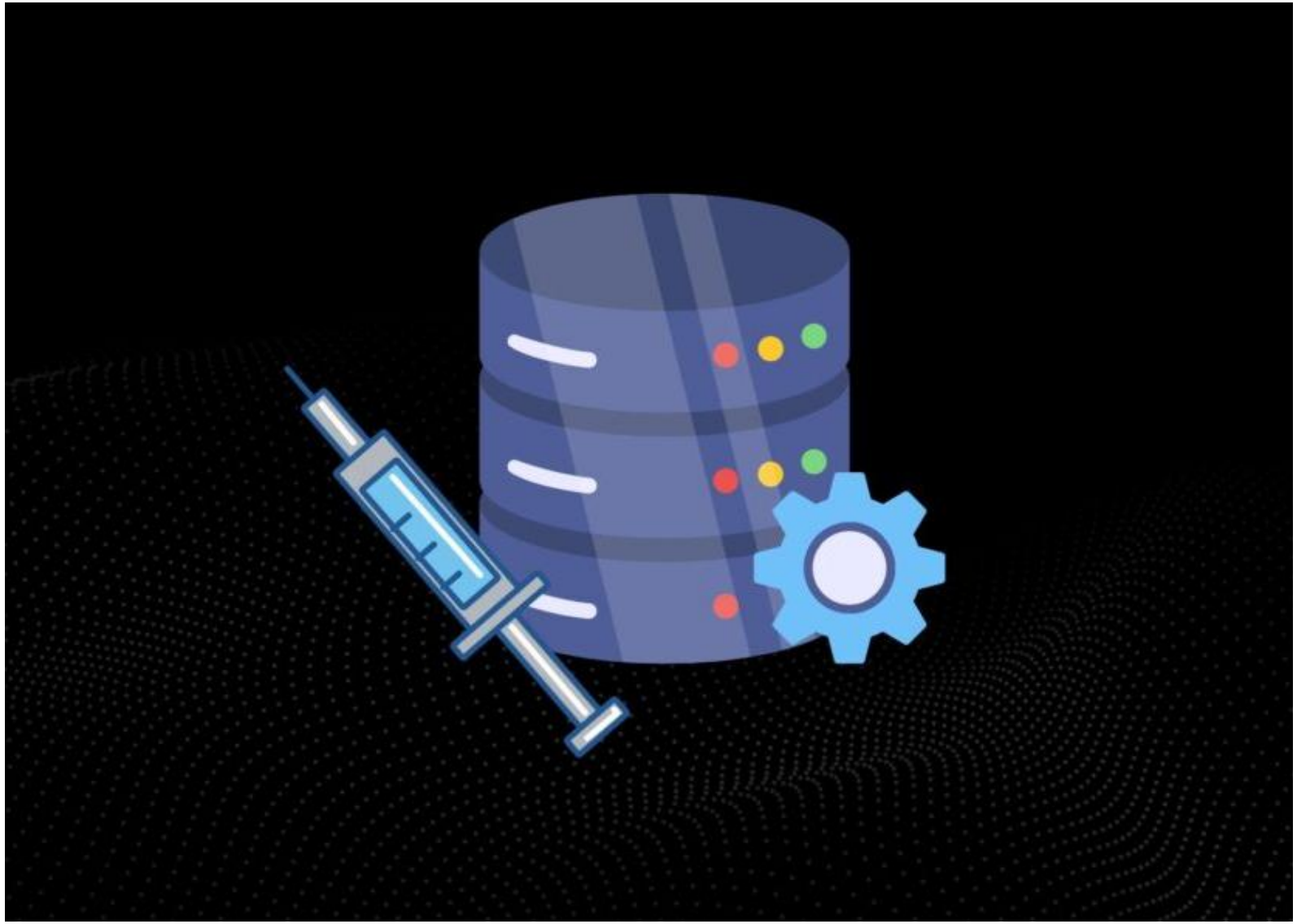
Kacper Ślęzak

Martyna Dominika Peukert

Co to jest SQL Injection?

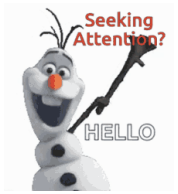


- To podatność aplikacji webowych, dzięki której napastnik może wstrzyknąć własny fragment zapytania SQL. Najczęściej związana jest z błędnym podejściem do budowania zapytań do bazy danych przez programistów aplikacji.



Przyczyny ataków SQL Injection

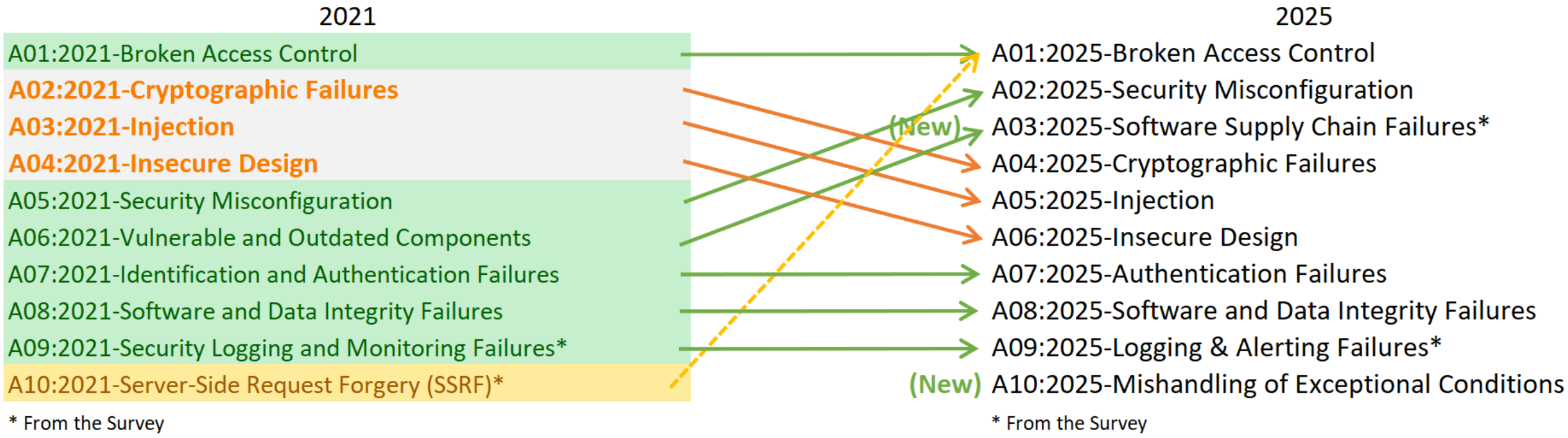
- Źle zaprojektowana witryna internetowa



Stan zagrożenia ->

https://owasp.org/Top10/2025/0x00_2025-Introduction/

Zmiany w strukturze zagrożeń sieciowych



Czym może skutkować ten atak?

W zależności od sytuacji możemy mieć do czynienia z:

- nieautoryzowanym dostępem w trybie odczytu lub zapisu do całej bazy danych,
- możliwością ominięcia mechanizmu uwierzytelnienia,
- możliwością odczytania wybranych plików (system operacyjny, na którym pracuje baza danych),
- możliwością tworzenia plików w systemie operacyjnym, na którym pracuje baza,
- możliwością wykonania kodu w systemie operacyjnym (uprawnienia użytkownika, na którym pracuje baza lub web serwer – w przypadku aplikacji webowych).

CRUD

create

read

update

delete

Mechanika SQL Injection

01

Dynamiczne konstruowanie zapytań

02

Rola komentarzy i operatorów logicznych

03

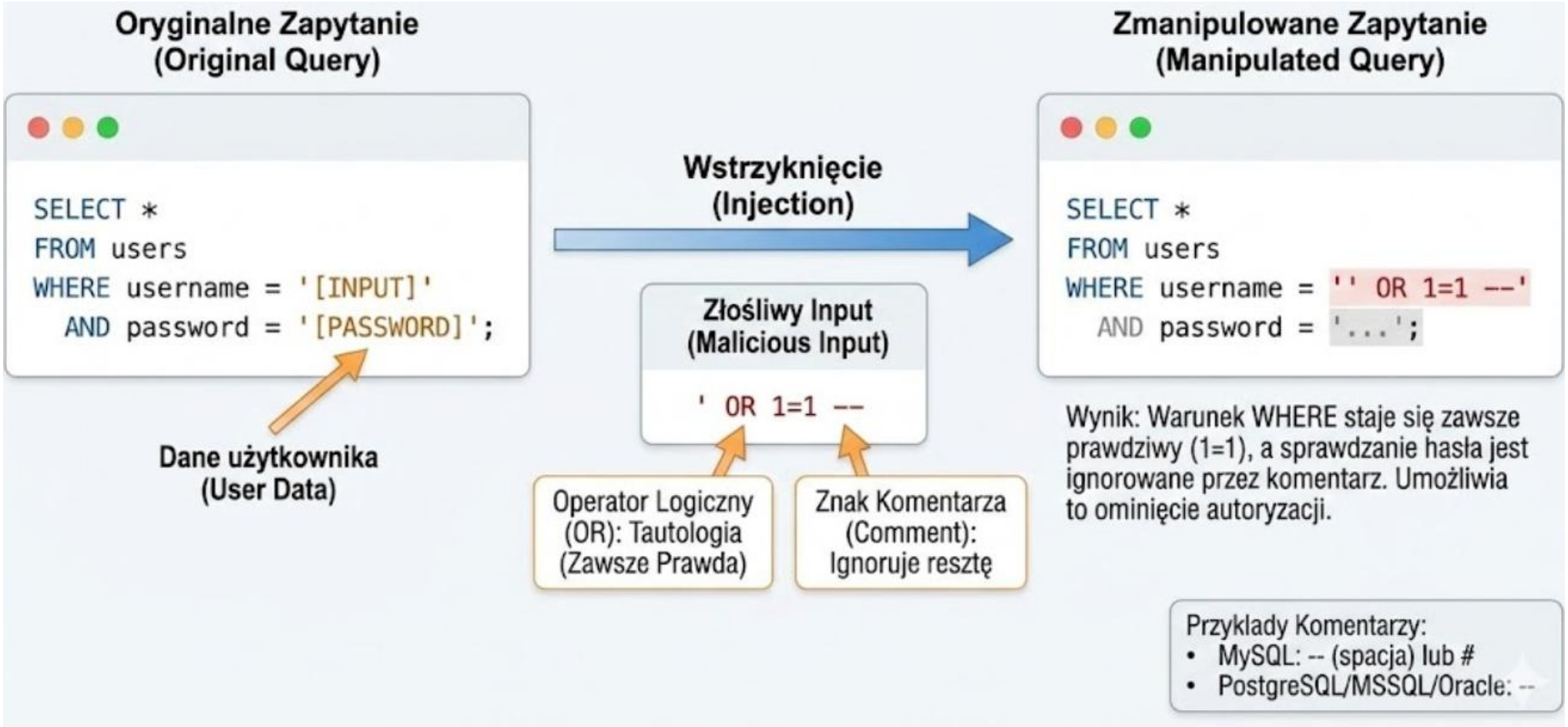
Płaszczyzny ataku: GET, POST, Cookies i Nagłówki

Dynamiczne konstruowanie zapytań

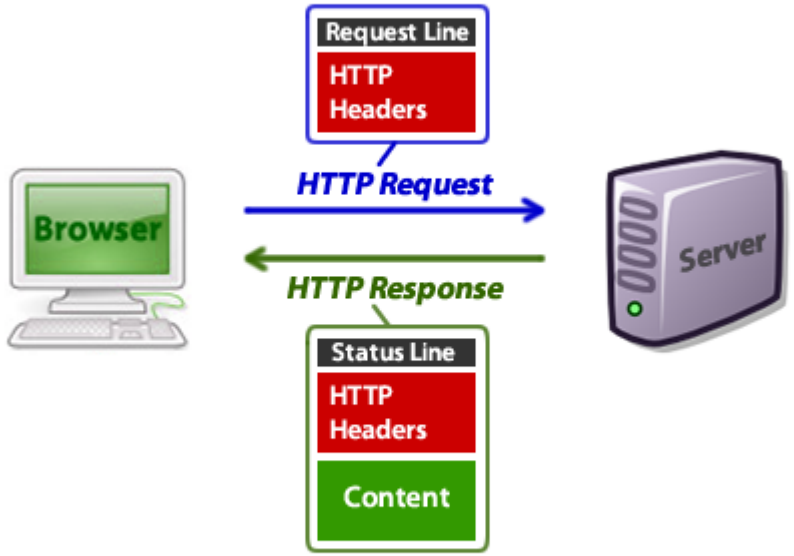
Przykład implementacji naiwnego uwierzytelnienia w języku PHP:

```
$username = $_POST['username'];
$query = "SELECT * FROM users WHERE name = '". $username. "'";
```

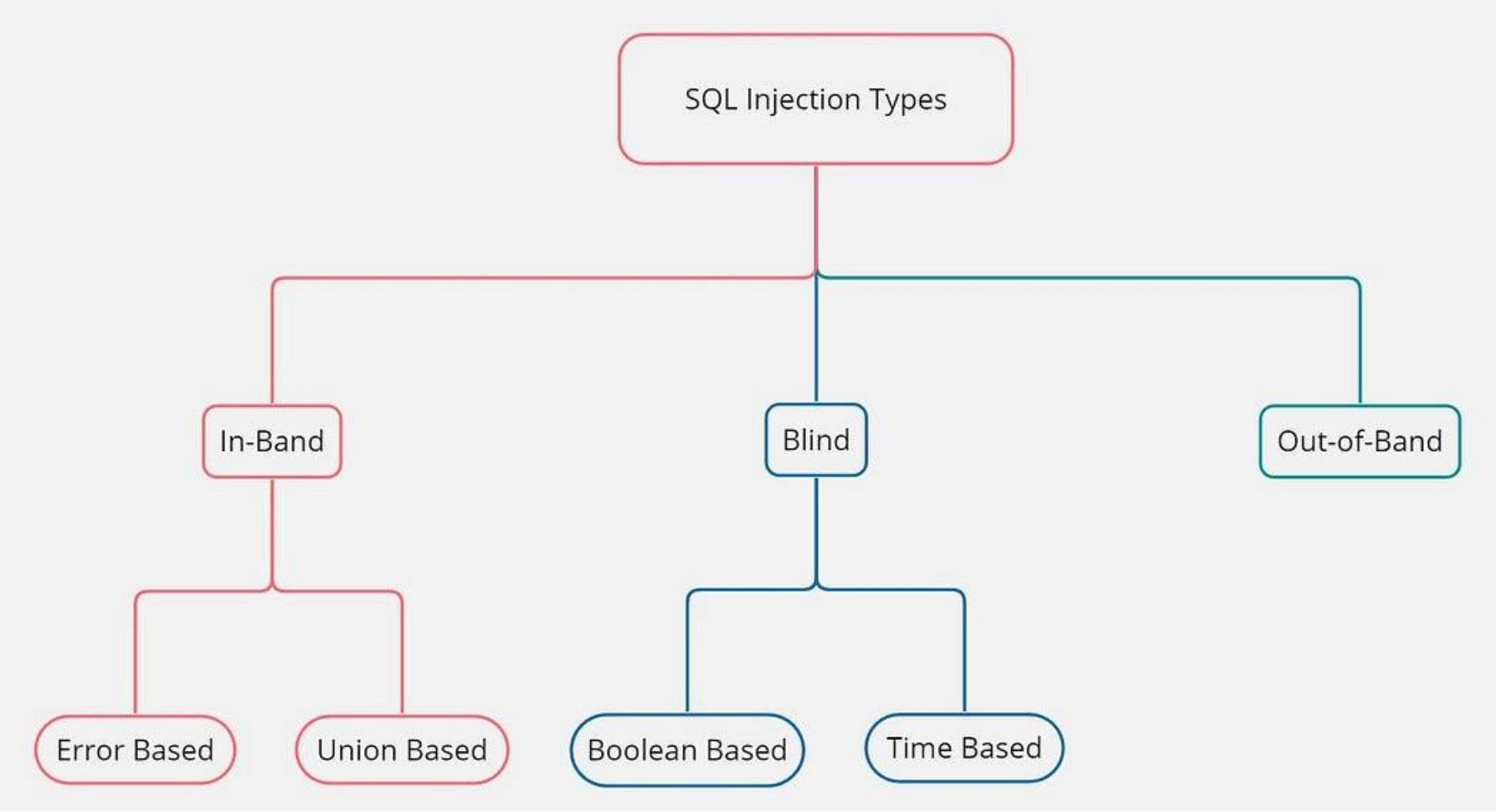
Rola komentarzy i operatorów logicznych



Płaszczyzny ataku: GET, POST, Cookies i Nagłówki



Sposoby wykorzystania SQL Injection



UNION-based

Dzięki UNION jesteśmy w stanie połączyć ze sobą wyniki pobierania danych z dwóch różnych tabel SQL.

```
SELECT kolumna1, kolumna2, kolumna3 FROM tabela1

UNION SELECT col1, col2, col3 FROM tabela2
```

W wyniku wykonywania zapytania zwracana jest treść kolumn kolumna1, kolumna2, kolumna3 z tabeli *tabela1* oraz kolumn col1, col2, col3 z tabeli *tabela2*.

Aby silnik bazodanowy nie odmówił wykonania zapytania, należy pamiętać o dwóch najważniejszych kwestiach:

- Liczba kolumn w obu zapytaniach SELECT **musi być taka sama**
- Typy kolumn w obu zapytaniach SELECT muszą być zgodne*

W praktyce więc wykorzystanie SQL Injection typu UNION sprowadza się do ustalenia:

1. Ile kolumn ma oryginalne zapytanie
2. Treść której z kolumn napastnik jest w stanie przeczytać.

Określanie liczby wymaganych kolumn

1. Testowanie z użyciem „ORDER BY”

- Stopniowo zwiększa się numer kolumny w klauzuli ORDER BY
- Gdy numer przekroczy liczbę dostępnych kolumn, aplikacja zwraca błąd lub inną nietypową odpowiedź. Na tej podstawie można określić, ile kolumn zawiera wynik zapytania.

1.

```
' ORDER BY 1--
' ORDER BY 2--
' ORDER BY 3--
```

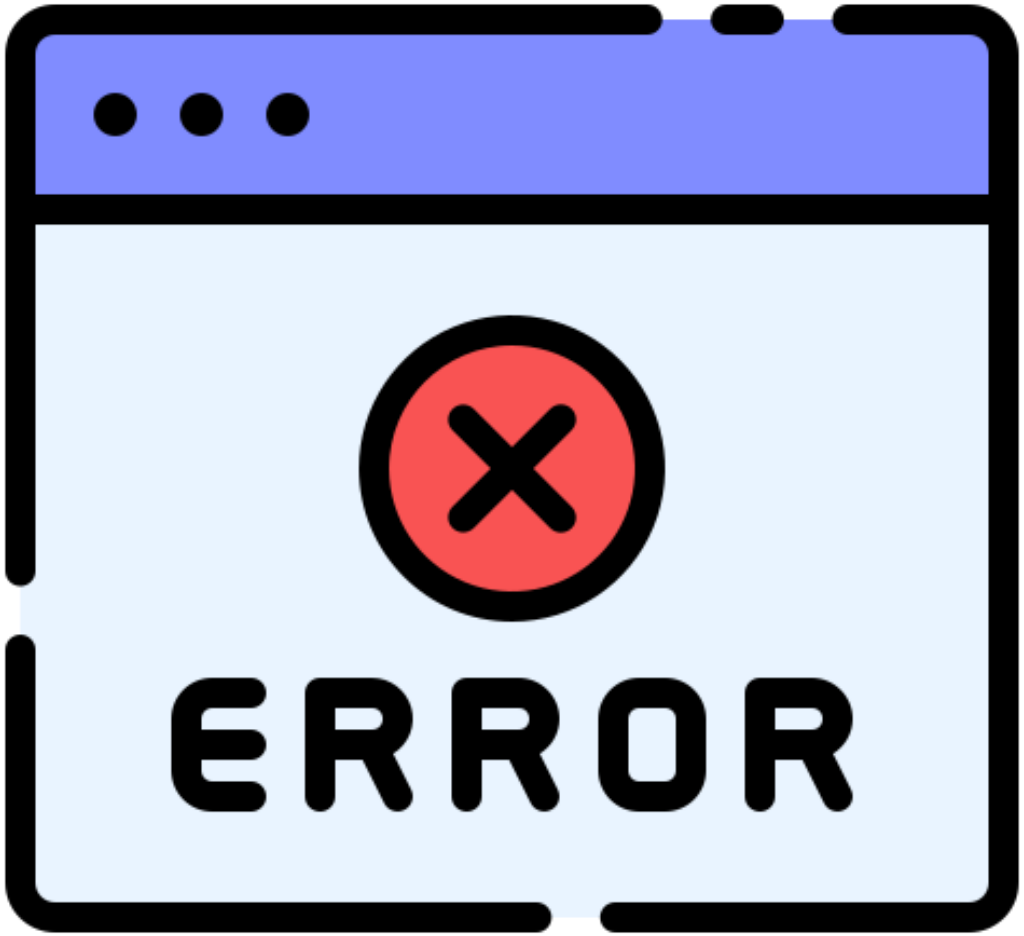
2.

```
' UNION SELECT NULL–
' UNION SELECT NULL,NULL–
' UNION SELECT NULL,NULL,NULL--
```

2. Testowanie z użyciem „UNION SELECT”

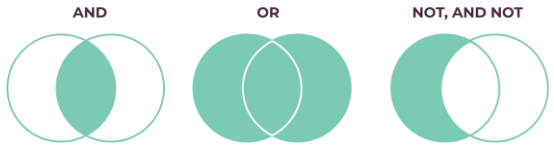
- Przesyła się zapytania UNION SELECT zawierające różną liczbę pól (np. wartości NULL).
- Jeśli liczba pól nie pasuje do liczby kolumn oryginalnego zapytania, baza zgłasza błąd.
- Gdy liczba pól jest prawidłowa, system zwróci dodatkowy wiersz (lub zmienioną odpowiedź), co pozwala potwierdzić poprawną liczbę kolumn.

ERROR-based



BLIND

BLIND Boolean-Based




```
SELECT * FROM products WHERE id = product_id
```

Na początku złośliwy haker używa aplikacji w legalny sposób, aby odkryć co najmniej jeden istniejący identyfikator produktu – w tym przykładzie jest to produkt 42. Następnie może podać następujące dwie wartości dla *product_id* :

```
42 AND 1=1
42 AND 1=0
```

Jeżeli to zapytanie zostanie wykonane w aplikacji przy użyciu prostej konkatenacji ciągów znaków, zapytanie będzie wyglądać następująco:

```
SELECT * FROM products WHERE id = 42 and 1=1

SELECT * FROM products WHERE id = 42 and 1=0
```

Jeżeli aplikacja zachowuje się w każdym przypadku inaczej, jest podatna na ataki typu blind SQL Injection oparte na wartościach logicznych.

BLIND Time-Based

```
SELECT * FROM products WHERE id = product_id
```

Złośliwy haker może podać następującą wartość *product_id* :

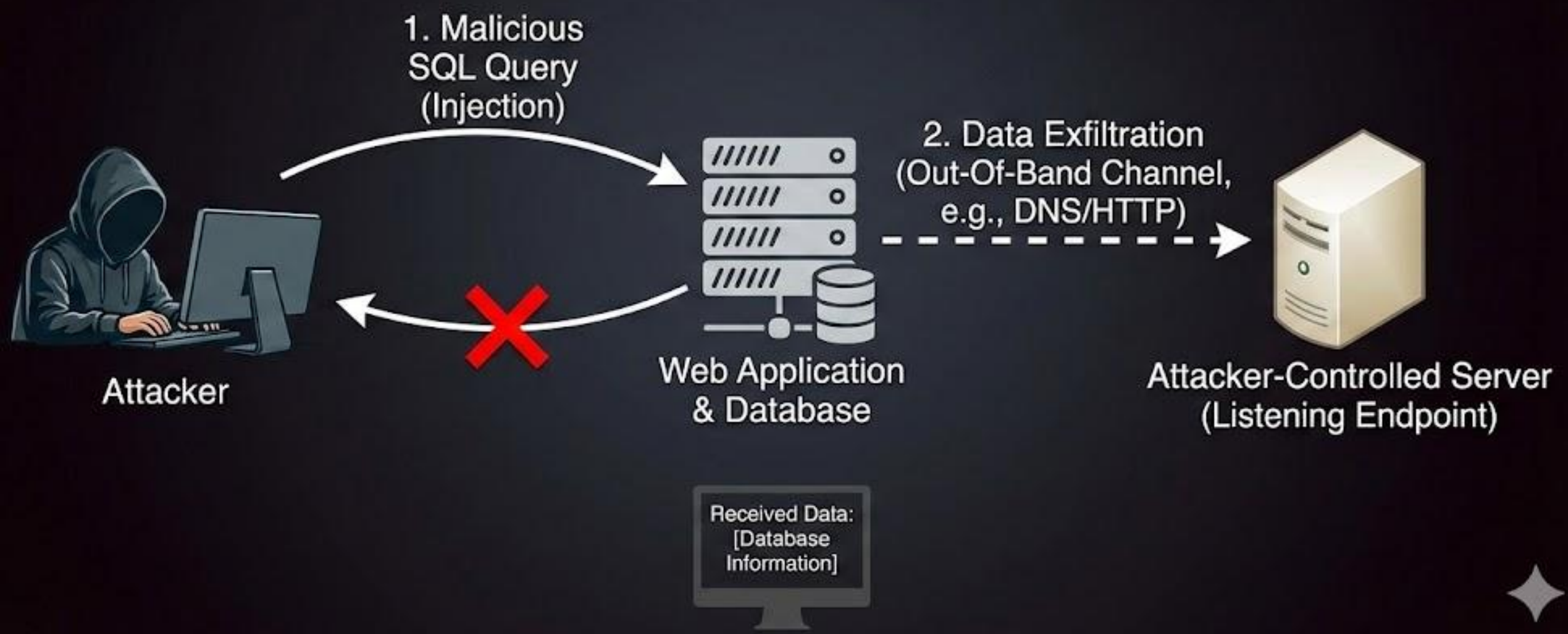
42; WAITFOR DELAY '0:0:10’

W rezultacie zapytanie wygląda następująco:

```
SELECT * FROM products WHERE id = 1; WAITFOR DELAY '0:0:10'
```

Jeśli serwerem bazy danych jest Microsoft SQL Server, a aplikacja jest podatna na ataki typu „ślepe wstrzykiwanie SQL” oparte na czasie, atakujący dostrzeże 10-sekundowe opóźnienie w działaniu aplikacji.

Out-Of-Band SQL Injection



Mechanizmy ochrony przed SQL Injection

Zapytania parametryzowane

Zapytania parametryzowane w SQL odnoszą się do metody wykonywania zapytań do bazy danych, która pozwala na wprowadzanie dynamicznych wartości w czasie wykonywania. Ta technika zwiększa bezpieczeństwo, zapobiegając atakom SQL injection, ponieważ oddziela logikę zapytania od danych wejściowych użytkownika.

```
query = ”SELECT * FROM users WHERE username = ? OR email = ?”
stmt = Database.prepareStatement(query)
stmt.setString(1, username)
Stmt.setString(2, email)
Stmt.execute()
```

O ile najpowszechniej w zapytaniach parametryzowanych stosuje się pytajniki, o tyle w zależności od używanej bazy danych i języka programowania mogą wystąpić odmienne fromaty parametryzacji. Wśród innych spotykanych opcji znajduje się znak dwukropka przed nazwą parametru lub małpki.

```
username = :username
```

```
username = @username
```

Walidacja typów danych

Walidacja typów danych polega na weryfikacji, czy dane, które podał użytkownik, są dokładnie w takim formacie, jak oczekujemy, np.:

- czy pole id jest liczbą,
- czy pole z datą ma format yyy-mm-dd,
- czy kod pocztowy ma format xx-xxx,
- czy PESEL jest liczbą 11-cyfrową z właściwą cyfrą kontrolną.

Stosowanie systemów klasy ORM

- Systemy klasy ORM (Object- relational mapping) zdejmują de facto z programistów aplikacji odpowiedzialność za pisanie zapytań SQL i generują je w pełni automatycznie. Odbywa się to poprzez zdefiniowanie mapowania pomiędzy klasami w aplikacji a tabelami w bazie danych. Popularne systemy ORM to m.in. Hibernate, Nhibernate, Dapper, Doctrine czy Entity Framework.
- W praktycznie każdym systemie ORM istnieje jednak możliwość definiowania treści zapytań w języku specyficznym dla tego ORM-a (np. w Hibernate jest to HQL – Hibernate Query Language). Jeśli korzystamy w aplikacji z możliwości użycia takiego języka, miejmy na uwadze, że wracają bardzo zbliżone problemy jak SQL Injection. *Istnieje np. już opisana podatność HQL Injection!*

Hardening bazy danych

Dobre praktyki: HARDENING BAZY DANYCH

- Aplikacja nie powinna łączyć się z bazą danych z uprawnieniami użytkownika administracyjnego bazy.
- W miarę możliwości należy stosować separację użytkowników na poziomie bazy danych (lub nawet separację samych baz).
- Stacked queries powinny być wyłączone.
- Należy wyłączyć potencjalnie niebezpieczne procedury, takie jak *xp_cmdshell* w SQL Serverze, w wyniku których może dochodzić do poważniejszych podatności.
- Proces bazy danych w systemie opracyjnym nie powinien działać na uprawnieniach *root*.

Ciekawostki – realne ataki SQL Injection

- Polska -> **Pięć polskich organizacji padło ofiarą grupy haktywistów.**
- Świat -> **28-letni haker ukradł 130 mln numerów kart**

QUIZ

Bibliografia

- „Bezpieczeństwo aplikacji webowych” wyd. Securitum
- https://sekurak.pl/czym-jest-sql-injection/
- https://www.ascocyber.pl/sql-injection-na-czym-polega-i-jak-sie-przed-nim-chronic/
- https://owasp.org/Top10/2025/0x00_2025-Introduction/
- https://www.malwarebytes.com/pl/sql-injection
- https://tryhackme.com/room/sqlinjectionlm
- https://portswigger.net/web-security/sql-injection
- https://bezpieczenstwoxd.pl/05/06/2024/sql-injection-nie-takie-banalne-jakby-sie-to-wydawalo/
- https://medium.com/@anhackx/sql-injection-types-f54de766e57e
- https://boringowl.io/blog/co-to-jest-sql-injection-i-jakie-zagrozenia-niesie-ze-soba#rodzaje-sql-injection
- https://www.trendmicro.com/pl_pl/what-is/cyber-attack/types-of-cyber-attacks/sql-injection-attack.html#types-tm-id
- https://nordvpn.com/pl/blog/sql-injection-co-to/?srsltid=AfmBOoqvCN_JKiNnkM_sG90tTRACaYzsC9Y4NMSvkHlLd7lOkMrQJCFO
- https://www.indusface.com/learning/error-based-sql-injection/
- https://www.invicti.com/learn/blind-sql-injection
- https://www.elpassion.com/pl/glossary/what-is-parameterized-queries-in-sql
- https://cyberdefence24.pl/cyberbezpieczenstwo/wyciekly-hasla-z-polskich-portali
- https://www.securitymagazine.pl/pl/a/najglosniejsze-cyberincydenty-w-2024-roku-w-polsce-i-na-swiecie

Dziękujemy za uwagę