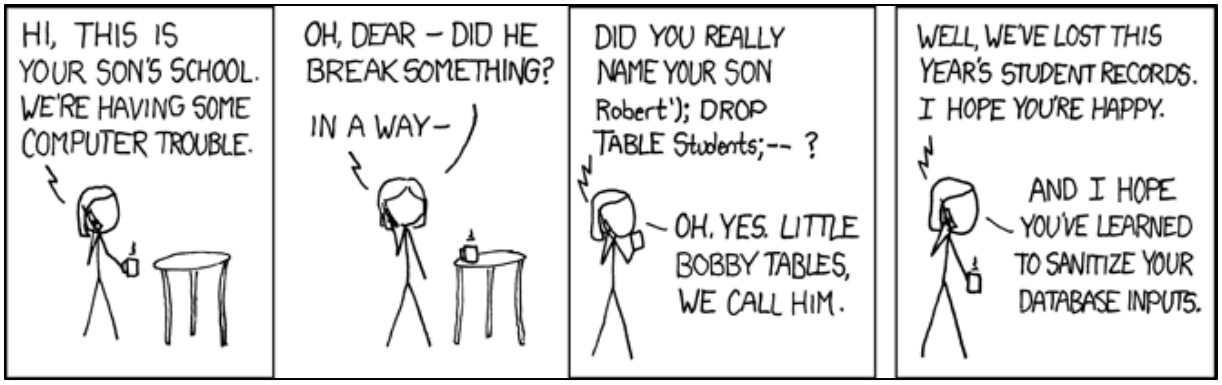
Bazy Danych 2 – Lab 7

SQL Injection



SQL Injection (SQLi) to rodzaj ataku polegającego na wstrzyknięciu złośliwych instrukcji SQL, które umożliwiają wykonanie nieautoryzowanych komend na serwerze baz danych obsługującym aplikację internetową. Atakujący mogą wykorzystać podatność SQL Injection, aby **ominąć mechanizmy bezpieczeństwa aplikacji** – uwierzytelnianie i autoryzację – oraz **uzyskać dostęp do całej zawartości bazy danych SQL**. Ponadto mogą **dodawać, modyfikować i usuwać dane** w bazie.

Podatność na SQL Injection może występować w każdej stronie lub aplikacji internetowej, która używa bazy danych SQL – np. MySQL, Oracle, SQL Server i innych. Przestępcy mogą jej użyć do uzyskania **nieautoryzowanego dostępu do poufnych danych**, takich jak informacje o klientach, dane osobowe, tajemnice handlowe czy własność intelektualna. SQL Injection to **jedna z najstarszych, najczęstszych i najgroźniejszych luk bezpieczeństwa w aplikacjach internetowych**.

Czym są zapytania SQL?

SQL to język zapytań stworzony do zarządzania danymi w relacyjnych bazach danych. Umożliwia dostęp, modyfikację i usuwanie danych. Zapytania SQL to zazwyczaj polecenia żądające określonych danych, np. za pomocą SELECT, choć istnieją też inne instrukcje: UPDATE, DELETE, DROP.

Aplikacje często **łączą dane wejściowe od użytkowników z zapytaniami SQL**, aby pobrać odpowiednie dane z bazy. Najprostszy przykład logowania użytkownika mógłby wyglądać tak:

SELECT id FROM users WHERE username='dane-od-użytkownika' AND password='dane-od-użytkownika'

Jeśli aplikacja **bezpośrednio wstawia dane użytkownika do zapytania SQL**, jest **podatna na SQL Injection**.

Jak i dlaczego wykonuje się atak SQL Injection?

Aby przeprowadzić atak SQLi, atakujący musi **znaleźć pola wejściowe** w aplikacji, które są wrażliwe na SQL Injection. Jeśli dane wejściowe użytkownika są **używane bez walidacji** w zapytaniu SQL, atakujący może przesłać tzw. **ładunek (payload)** zawierający złośliwy kod SQL. Po przesłaniu formularza, ten kod zostaje wykonany przez serwer baz danych.

Skutki udanego ataku SQL Injection

Ponieważ wiele aplikacji i stron internetowych przechowuje wszystkie dane w bazach SQL, skutki ataku mogą być poważne:

- **Kradzież danych logowania** innych użytkowników, w tym administratorów.
- **Pełny dostęp do danych** w bazie (odczyt całej zawartości).
- **Modyfikacja danych**, np. zmiana salda konta w aplikacji finansowej lub anulowanie transakcji.
- **Usuwanie danych**, np. tabel – co może unieruchomić aplikację do momentu przywrócenia kopii zapasowej.
- **Dostęp do systemu operacyjnego**, jeśli serwer bazy danych na to pozwala – co może być początkiem ataku na sieć wewnętrzną.

Typy ataków SQL Injection

- **In-band SQLi** – atakujący otrzymuje odpowiedź w tym samym kanale, co wysyłane zapytanie. Najprostszy i najczęstszy typ ataku.
- **Out-of-band SQLi (OOB)** – dane odpowiedzi są przekazywane innym kanałem (np. DNS), przydatne, gdy aplikacja nie pokazuje wyników zapytania.
- **Blind SQLi (niewidoczne)** – atak oparty na analizie zachowania aplikacji (czas odpowiedzi, reakcje na wartości logiczne true/false), używany, gdy nie ma dostępu do wyników.
- **Second-order SQLi** – ładunek SQL jest zapisany w systemie i wykonany później, np. gdy dane są przetwarzane przez innego użytkownika.

Przykład prostego ataku SQL Injection

Poniższy pseudokod reprezentuje prostą aplikację logowania:

```
uname = request.POST['username']

passwd = request.POST['password']

sql = "SELECT id FROM users WHERE username='" + uname + "' AND password='" + passwd + "'"

database.execute(sql)
```

Jeśli użytkownik poda w polu password:

```
password' OR 1=1
```

To zapytanie SQL stanie się:

```
SELECT id FROM users WHERE username='username' AND password='password' OR 1=1
```

OR 1=1 zawsze zwraca true, więc aplikacja **uwierzytelnia atakującego jako pierwszego użytkownika w tabeli** – często jest to administrator.

Dodatkowo, atakujący może **"zakomentować"** dalszą część zapytania:

```
-- MySQL, MSSQL, Oracle, PostgreSQL, SQLite

' OR '1'='1' --

' OR '1'='1' /*

-- MySQL

' OR '1'='1' #

-- MS Access

' OR '1'='1' %00

' OR '1'='1' %16
```

Przykład SQL Injection typu UNION

UNION umożliwia **połączenie wyników kilku zapytań SELECT**. Jest to często wykorzystywane do **wydobycia danych z innych tabel**.

Przykład:

Legalne zapytanie:

```
GET http://testphp.vulnweb.com/artists.php?artist=1
```

Złośliwe zapytanie:

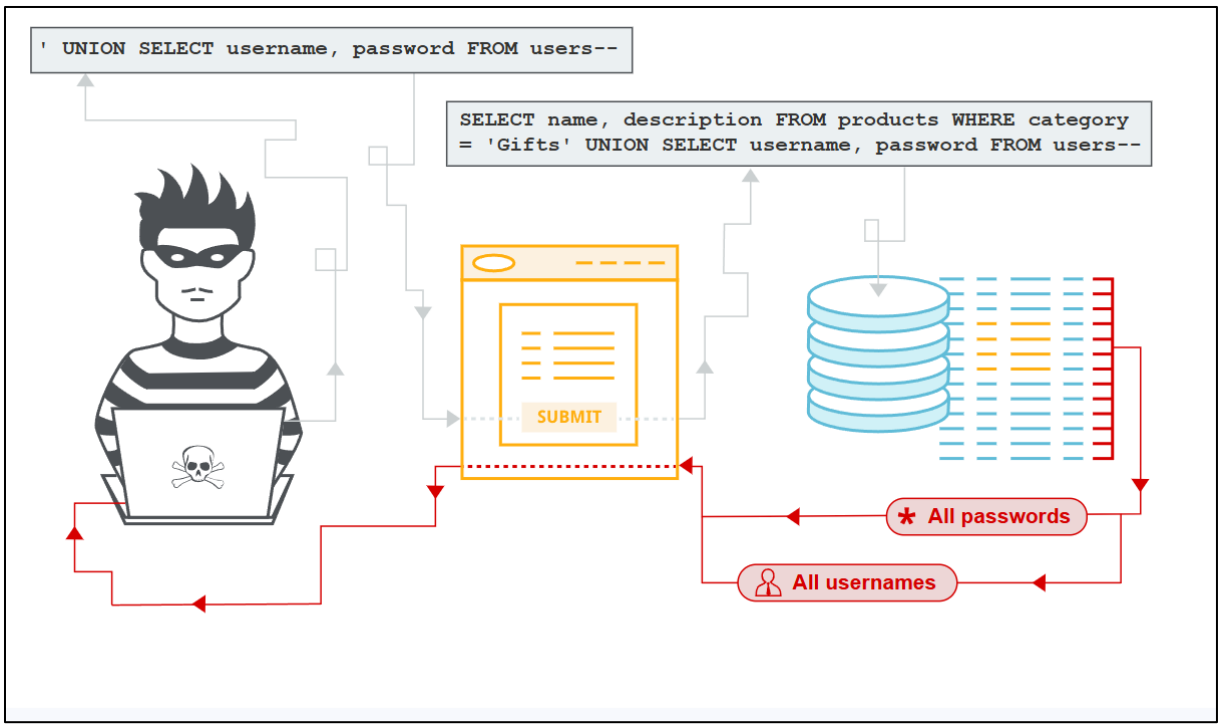
```
GET http://testphp.vulnweb.com/artists.php?artist=-1 UNION SELECT 1,2,3
```

To zapytanie łączy pierwotne zapytanie z nowym, którego wyniki są również wyświetlane.

Przykład z kradzieżą danych:

```
GET http://testphp.vulnweb.com/artists.php?artist=-1 UNION SELECT 1,pass,cc FROM users WHERE uname='test'
```

Tutaj atakujący łączy wynik z tabeli users, wyciągając **hasła (pass)** i **numery kart kredytowych (cc)** użytkownika test.



Jak zapobiegać SQL Injection?

Anatomia typowej luki SQL Injection

Typowa luka w języku Java wygląda następująco. Ponieważ parametr „customerName” nie jest walidowany, może zostać bezpośrednio dołączony do zapytania, umożliwiając atakującemu wstrzyknięcie własnego kodu SQL:

```
String query = "SELECT account_balance FROM user_data WHERE user_name = "
        + request.getParameter("customerName");
try {
   Statement statement = connection.createStatement();
   ResultSet results = statement.executeQuery(query);
}
```

Podstawowe formy ochrony

Opcja 1: Użycie przygotowanych zapytań (ang. *Prepared Statements*)
Opcja 2: Użycie bezpiecznie zdefiniowanych procedur składowanych (*Stored Procedures*)
Opcja 3: Walidacja wejścia oparta na białej liście (*Allow-list Input Validation*)
Opcja 4: STANOWCZO ODRADZANA: Ucieczka znaków wejściowych

Obrona 1: Przygotowane zapytania

Programiści powinni być uczeni używania zapytań z parametrami (przygotowanych zapytań), które oddzielają kod SQL od danych. Dzięki temu nawet złośliwe dane wejściowe nie zmienią logiki zapytania.

Przykład bezpiecznego zapytania w języku Java:

```
String custname = request.getParameter("customerName");
String query = "SELECT account_balance FROM user_data WHERE user_name = ?";
PreparedStatement pstmt = connection.prepareStatement(query);
pstmt.setString(1, custname);
ResultSet results = pstmt.executeQuery();
```

Przykład bezpiecznego zapytania w C# .NET:

```
String query = "SELECT account_balance FROM user_data WHERE user_name = ?";
OleDbCommand command = new OleDbCommand(query, connection);
command.Parameters.Add(new OleDbParameter("customerName", CustomerName.Text));
OleDbDataReader reader = command.ExecuteReader();
```

Obrona 2: Procedury składowane

Choć procedury składowane nie są z definicji bezpieczne, można je pisać w sposób zabezpieczający przed SQL Injection – tak samo jak zapytania z parametrami. Kluczowe jest, by **nie generować dynamicznego SQL w procedurach**.

Bezpieczny przykład w języku Java:

```
String custname = request.getParameter("customerName");
CallableStatement cs = connection.prepareCall("{call sp_getAccountBalance(?)}");
cs.setString(1, custname);
ResultSet results = cs.executeQuery();
```

Bezpieczny przykład w VB .NET:

```
Dim command As SqlCommand = new SqlCommand("sp_getAccountBalance", connection)
command.CommandType = CommandType.StoredProcedure
command.Parameters.Add(new SqlParameter("@CustomerName", CustomerName.Text))
Dim reader As SqlDataReader = command.ExecuteReader()
```

Obrona 3: Walidacja wejścia oparta na białej liście

W przypadkach, gdy nie można użyć zmiennych wiązanych (np. przy dynamicznej nazwie kolumny czy sortowaniu), należy stosować walidację wejścia – najlepiej przez mapowanie do zdefiniowanych wcześniej wartości.

Przykład:

```
String tableName;
switch(PARAM) {
  case "Value1": tableName = "fooTable"; break;
  case "Value2": tableName = "barTable"; break;
  default: throw new InputValidationException("Nieoczekiwana wartość dla nazwy tabeli");
}
```

Obrona 4 (STANOWCZO ODRADZANA): Escaping wszystkich danych wejściowych

Ta metoda polega na „ucieczce” znaków wejściowych, ale jest krucha i bardzo zależna od konkretnego silnika bazy danych. Nie gwarantuje pełnego bezpieczeństwa i **nie powinna być stosowana jako główna forma ochrony**.

Dodatkowe środki ochrony (obrona warstwowa)

Zasada najmniejszych uprawnień (Least Privilege)

Każde konto w bazie danych powinno mieć tylko takie uprawnienia, jakie są niezbędne. **Nigdy nie przypisuj uprawnień administratora** kontom aplikacyjnym – nawet jeśli „wszystko działa".

Przykład: różne konta dla różnych aplikacji

Strona logowania potrzebuje tylko dostępu do odczytu (SELECT), natomiast formularz rejestracyjny potrzebuje INSERT – te dwa moduły powinny używać **różnych kont bazy danych**.

Wzmocnienie ograniczeń przez widoki SQL

Widoki mogą ograniczyć dostęp tylko do określonych kolumn. Przykładowo, zamiast dawać aplikacji dostęp do tabeli z hasłami, twórca może stworzyć widok, który zwraca tylko ich zaszyfrowane wersje.

Walidacja wejścia

Walidacja danych wejściowych powinna być stosowana **zawsze**, nawet jeśli korzystasz z zapytań z parametrami. Więcej informacji można znaleźć w dokumencie **Input Validation Cheat Sheet**.

Materiały dodatkowe

- https://book.hacktricks.wiki/pl/pentesting-web/sql-injection/index.html
- https://github.com/swisskyrepo/PayloadsAllTheThings/tree/master/SQL%20Injection
- https://bobby-tables.com/
- https://owasp.org/www-project-web-security-testing-guide/stable/4-Web_Application_Security_Testing/07-Input_Validation_Testing/05-Testing_for_SQL_Injection.html
- https://wiki.owasp.org/index.php/Reviewing_Code_for_SQL_Injection
- https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html
- https://cheatsheetseries.owasp.org/cheatsheets/Query_Parameterization_Cheat_Sheet.html
- https://owasp.org/www-community/attacks/SQL_Injection
- https://owasp.org/www-community/attacks/Blind_SQL_Injection
- https://www.w3schools.com/sql/sql_injection.asp
- https://github.com/sqlmapproject/sqlmap
- https://www.esecurityplanet.com/threats/how-to-prevent-sql-injection-attacks/

Zadania

W trakcie zajęć będziemy przerabiać zadania z następujących stron:

- https://portswigger.net/web-security/sql-injection
- https://tryhackme.com/room/sqlilab
- https://tryhackme.com/room/sqlinjectionlm
- https://tryhackme.com/room/advancedsqlinjection
- https://tryhackme.com/room/sqlmap