|  |  |
| --- | --- |
| Bezpieczeństwo aplikacji Internetowych  Dr inż. Maciej Brzozowski | Studenci:  Bogumił Bierć  Łukasz Mosiej |
| Temat: METODY ZABEZPIECZEN PRZED SQL INJECTION |  |

# Treść zadań

1. W jaki sposób sprawdzic podatność na SQL Injection?
2. W jaki sposób przy wykorzystaniu SQL Injection sprawdzic możliwość wykonania zapytań zagniezdżonych?
3. W jaki sposób przy wykorzystaniu SQL Injection okreslić liczbę kolumn w zapytaniu?
4. Czym charakteryzuje się atak SQL Injection typu blind?
5. Jak zabezpieczyc się przed SQL Injection - w czym może by ć pomocne rzutowanie i escap’owanie? Czy wyrazenia regularne są dobrym rozwiązaniem?
6. W jaki sposób osoba atakująca moze wykorzystać UNION oraz opóźnienie zapytania?
7. Czym jest atak wielofazowy SQL Injection?
8. Jak oceniasz wykorzystanie przy parametryzowaniu zapytan białych oraz czarnych list? Co jest lepszym rozwiązaniem?
9. Jak oceniasz wykorzystanie procedur składowanych?
10. Jaki wpływ na bezpieczenstwo ma projekt bazy oraz poziom uprawnień zytkowników.

# Rozwiązania

1. W celu sprawdzenia podatności na SQL injection należy w miejsce jednego z parametrów, który przekazujemy w formularzu podstawić kawałek kodu SQL.

Przykładowo biorąc zapytanie z poprzednich zajęć:

<http://localhost:8085/login?login=bolek&password=Bolek123!>

Możemy zmodyfikować parametr password, dodając tam kawałek SQL:

[http://localhost:8085/login?login=bolek&password=Bolek123!'+OR+1=1+OR+'’='](http://localhost:8085/login?login=bolek&password=Bolek123!'+OR+1=1+OR+''=')

Jak widać w linku: parametr password dostał na zakończeniu apostrof, dołożyliśmy warunek który jest zawsze prawdziwy oraz warunek, który umozliwił nam zakończenie pierwotnego apostrofu dla hasła.

Taki atak powiódłby się, gdyby aplikacja wysyłała do bazy zapytanie w postaci przybliżonej do poniższej:

**SELECT \* FROM USERS WHERE LOGIN=:login AND PASSWORD=:password**

Zakładając, że wartości byłyby podstawiane bez escapowania.

Jeśli atak się powiedzie, to zostaniemy zalogowani do aplikacji.

1. W celu sprawdzenia możliwości wykonywania zapytań zagnieżdzonych ponownie będziemy bazować na linku z poprzednich zajeć:

<http://localhost:8085/login?login=bolek&password=Bolek123!>

Tym razem w miejsce hasła wstawimy podzapytanie:

[http://localhost:8085/login?login=bolek&password='+OR+PASSWORD=(SELECT+PASSWORD+FROM+USERS+WHERE+LOGIN='BOLEK')+OR+''+='](http://localhost:8085/login?login=bolek&password='+OR+PASSWORD=(SELECT+PASSWORD+FROM+USERS+WHERE+LOGIN='BOLEK')+OR+''+=' )

Jeśli atak się powiedzie, to zostaniemy zalogowani do aplikacji.

1. Najprostszym sposobem na określenie liczby kolumn w zapytaniu jest skorzystanie z klauzuli ORDER BY

<http://localhost:8085/login?login=bolek&password=%27+OR+1%3D1+ORDER+BY+2+OR+%27%27%3D%27>

W aplikacji podatnej na SQL Injection zapytanie powiedzie się, jeśli podaliśmy przy ORDER BY wartość mniejszą lub równą ilości kolumn w zapytaniu. Dla większej ilość aplikacja powinna zwrócić błąd.

1. Atak typu blind ma pozwolić określić, czy aplikacja jest ogólnie podatna na SQL Injection. Nie zależy nam na wyniku zapytania ani zniszczeniu struktury bazy danych. Chodzi nam wyłącznie o sprawdzenie podatnośći celem późniejszych bardziej wyrafinowych ataków.

Ciężko jest wyznaczyć jednoznaczną granicę pomiędzy atakiem typu blind a zwykłym. Biorąć pod uwagę atak z zadania pierwszego moglibyśmy go uznać za blind, ponieważ jest to najprostszy atak, z drugiej strony zalogowaliśmy się do aplikacji, więc poniekąd otrzymaliśmy też jakiś wynik zapytania (lub błąd).

1. Rzutowanie jest średnim rozwiązaniem – w logach bazy danych możemy otrzymywać niepotrzebne wyjątki rzutowania (przykładowo, gdy ktoś będzie próbwał przeprowadzać atak na polu typu liczbowego). Wyrażenia regularne są pomocne, musielibyśmy poświęcić jednak dużo czasu na dostosowanie wyrażen dla każdego typu kolumn. Escapowanie jest najlepszym i najszybszym rozwiązaniem. Warto wykorzystać też frameworki, które robią to za nas. W Javie jest to np. Hibernate, który posiada gotowe metody do ustawiania wartości parametrów w zapytaniu (zapytanie definiujemy przez pseudo-wartośći w stylu password=:password).
2. Przykładowy atak wykorzystujący UNION:

[http://localhost:8085/login?login=bolek&password='+OR+''='+UNION+SELECT+\*+FROM+MESSAGES](http://localhost:8085/login?login=bolek&password='+OR+''='+UNION+SELECT+*+FROM+MESSAGES)

Atak z wykorzystaniem UNION jest jednak dość problematyczny. Oba zapytania muszą zwracać taką samą ilość kolumn i to w dodatku tego samego typu. Opis jak sobie z tym poradzić jest zawarty w kolejnym zadaniu.

1. W ataku wielofazowym wykorzystujemy połączenie wielu ataków typu SQL Injection. Załóżmy, że chcielibyśmy wykorzstać UNION, ale nie chcielibyśmy tracić zbyt wiele czasu – baza może nie być podatna na SQL Injection. Pierwszą fazą naszego ataku jest więc sprawdzenie podatnośći na SQL Injection (jak opisane w zad. 1). Kolejną fazą jest określenie ilości kolumn w zapytaniu (jak opisane w zad.3). Następnie przeprowadzamy atak typu UNION jak opisane w poprzednim zadaniu. Sam atak typu UNION może zająć wiele zapytań. Jeśli znamy ilość kolumn, to początkowo możemy zamiast \* w zapytaniu (SELECT \*) podstawiać null, który jest zgodny z każdym typem (SELECT null, null, null) itp.
2. Oba rozwiązania są problematyczne i czasochłonne. Przy zapytaniach, które mogą mieć ograniczoną liczbę wartości (dane statystyczne, np. Miesiąc sprawozdania) lepiej sprawdzą się białe listy. Przy zapytaniach, które mają dużo wartości (np. Tabela z danymi logowania) białe listy się nie sprawdzą. Jedynym wyjściem są czarne listy, który muszą być odpowiednio przygotowane (ktoś może mieć login „selectic”, nie możemy go wykluczyć doszukując sie SQL Injection).
3. Procedury składowe mogą ułatwić escapowanie zapytań. Nie są jednak wspaniałym rozwiązaniem, zwłaszcza jeśli korzystamy z frameworków do ORM. Posiadają one zazwyczaj wbudowane mechanizmy zabezpieczeń przed SQL Injection a dodatkowo są zoptymalizowane do pracy na encjach i klasach encji, więc wykorzystanie procedur kłóci się z ich ideą.
4. Poziom uprawnień i odpowiedni schemat może mieć bardzo duży wpływ na bezpieczeństwo danych. Kluczowe informacje zawarte w bazie danych powinne być dostępne dla wybranej ilości użytkowników, nie dla wszystkich. Odpowiednio zaprojektowana baza danych może też utrudnić ataki typu SQL Injection. Tabele powinne być nazwane w sposób opisowy (np. Users, a nie tabela1), ale dodanie podkreślnika w nazwie tabeli (\_Users) może już skutecznie utrudnić atak, a nazwa tabeli jest dalej czytelna. Nie można tego jednak stosować jako jedyne zabezpieczenie bazy danych.