



# Bezpieczeństwo Aplikacji Web

## 04. SQL Injection





# Bezpieczeństwo Aplikacji Web

## 04. SQL Injection

### Spis treści

Baza danych	4
SQL Injection	5
Prepared Statement	7
Stored Procedure	8
Pobieranie danych z innej tabeli	8
Atak SQL Injection	11
„Ślepe” ataki SQL Injection	12

## Baza danych

Zazwyczaj w aplikacjach większość danych przechowywane jest w bazie danych. Używa się ich ze względu na to, że są one przystosowane do przechowywania gigabajtów czy nawet terabajtów danych, a pobieranie danych z nich jest bardzo szybkie.

Najpopularniejszym typem baz danych są bazy relacyjne.

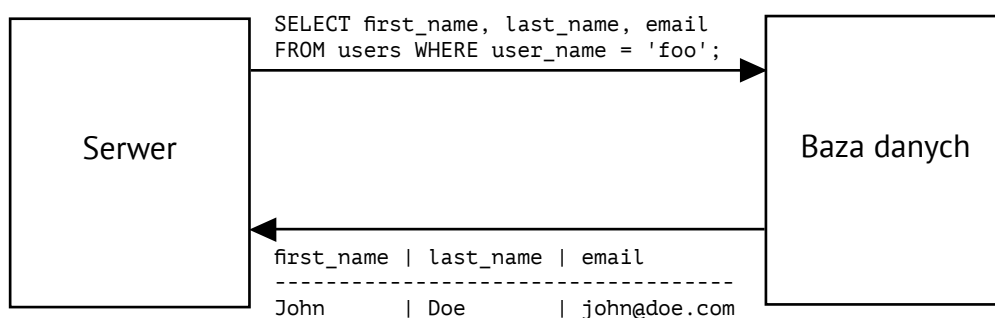
Dane w relacyjnych bazach danych przechowywane są w tabelach o ustalonych kolumnach. Pojedynczy wpis do bazy danych nazywa się encją. Encje mogą mieć między sobą relacje, czyli powiązania. Dodatkowo na niektóre kolumny można wprowadzać ograniczenia - na przykład uniemożliwiając wystąpienie duplikatów w kolumnie „user\_name”.

user_name	first_name	last_name	email	password_hash
foo	John	Doe	john@doe.com	\$2y\$12\$RoSAWDAg7W7zvlVwy-BrVeq5895w591yTAK/uSjA964Po7l/yH0la
demo	Jane	Smith	demo@user.com	\$2y\$12\$lof8Qyql6XKU0ixtv5aUMO/nYFRq82VTOX6SzoCTJRRuEjluo8pK
admin	System	Administrator	admin@localhost	\$2y\$12\$uN98MnxVmlaQWBskQO3qGe-d9l/qbkWfBtbe6fxiyuDcoGecw/YVNu

Ryc. 1. Przykładowa reprezentacja tabeli „users” w bazie danych.

Zbiór informacji o tym, jak wygląda baza danych, jakie ma tabele, jakie kolumny w tabelach i jakie ograniczenia nazywamy **schematem bazy danych**.

Aby móc wstawiać i pobierać dane do bazy danych używa się SQL. SQL to skrót od *Structured Query Language*, czyli *Strukturalny Język Zapytań*.



Ryc. 2. Przykładowe zapytanie o imię, nazwisko oraz email użytkownika.

Przykładowe zapytanie w tym języku pozwalające pobrać wszystkie dane z tabeli „users” wygląda następująco:

```
SELECT * FROM users;
```

Możliwe jest wybranie kolumn, które mają być pobrane. Pobranie wszystkich nazw użytkowników i adresów email z tabeli przedstawionej powyżej wygląda następująco:

```
SELECT user_name, email FROM users;
```

W zapytaniach możliwe jest też ograniczenie wyników poprzez dodanie warunków. Pobranie wszystkich danych o użytkownikach „foo” oraz „demo” wygląda następująco:

```
SELECT * FROM users WHERE user_name = 'foo' OR user_name = 'demo';
```

Dane mogą być też pobierane z wielu tabel poprzez wykonywanie złączeń. Na przykład jeżeli w tabeli orders przechowywane są zamówienia, możemy pobrać dane zamówień i użytkowników składających zamówienia następującym zapytaniem:

```
SELECT * FROM orders LEFT JOIN users ON orders.user_name = users.user_name;
```

Aby powyższe zapytanie było możliwe do wykonania, w obydwu tabelach muszą istnieć jakieś wspólne dane, które można powiązać. W tym przykładzie jest to kolumna user\_name reprezentująca nazwę użytkownika. Do danych z tabeli orders dodawane są pasujące dane z tabeli users, o ile w users istnieje wpis o pasującej wartości kolumny user\_name.

Do zapytań można dołączać też komentarze za pomocą -- :

```
SELECT * FROM users; -- wszyscy użytkownicy
```

## SQL Injection

Podatność SQL Injection należy do grupy podatności „wstrzykiwania” kodu (ang. code injection).

„Wstrzykiwanie” polega na tym, że wykorzystuje się istniejącą lukę w systemie, aby wykonać kod, którego nie było oryginalnie w aplikacji.

SQL Injection polega na tym, że „wstrzykuje” się dodatkowy kod do zapytania do bazy danych.

user_name	first_name	last_name	email	password
foo	John	Doe	john@doe.com	weakpassword
demo	Jane	Smith	demo@user.com	demo
admin	System	Administrator	admin@localhost	admin

Ryc. 3. Przykładowa tabela „users”.

Analiza tego ataku zostanie przeprowadzona na przykładowej tabeli „users”. Serwer podczas logowania może pobierać dane użytkownika używając wzoru poniższego zapytania:

```
SELECT * FROM users WHERE user_name = '$login' AND password = '$password';
```



W miejsce \$login aplikacja wstawia to, co użytkownik wpisał w pole „login”, a w \$password wstawia to, co wpisał w pole „hasło” na formularzu logowania. Logując się jako użytkownik „foo” z hasłem „weakpassword” wykonane zostanie następujące zapytanie:

```
SELECT * FROM users WHERE user_name = 'foo' AND password = 'weakpassword' ;
```

To, czy użytkownik zostanie pomyślnie uwierzytelniony, zależy od tego, czy zostanie zwrócony jakikolwiek wynik. Jeżeli w bazie danych nie zostanie znaleziony użytkownik pasujący do warunku `user_name = 'foo' AND password = 'weakpassword'`, to znaczy że dane logowania są błędne.

Co się stanie jeśli w pole „login” zostanie wpisane **admin'**; -- **pomiń resztę**?

```
SELECT * FROM users WHERE user_name = 'admin'; -- pomiń resztę'
AND password = 'weakpassword' ;
```

W tym momencie został wstrzyknięty kod do zapytania (w tym przypadku komentarz). System zarządzania bazą danych odrzuci treść komentarza i wykona tylko następujące zapytanie:

```
SELECT * FROM users WHERE user_name = 'admin' ;
```

Jeżeli jest to jedyny mechanizm uwierzytelniania w aplikacji, to w tym momencie atakujący załogował się jako użytkownik admin nie znając jego hasła.

Nastąpiło wstrzyknięcie kodu - zapytanie do bazy danych zostało zmodyfikowane. Tę podatność można wykorzystać nie tylko do załogowania się jako inny użytkownik. Możliwość modyfikacji zapytania pozwala potencjalnie na **wykonanie dowolnego kodu SQL**, który jest poprawny.

Wykorzystując SQL Injection możliwe jest więc:

- zmodyfikowanie wyniku zapytania (np. poprzez manipulację warunkami),
- odczyt dodatkowych danych (np. poprzez wykorzystanie złączeń),
- wstawianie danych bądź modyfikację schematu,
- wykonanie innych funkcji (np. `DATABASE()` w celu odczytu nazwy bazy danych albo `SHOW PROCESSLIST` w celu odczytu listy procesów bazy danych MySQL),
- wykonanie ataku DoS (np. używając `BENCHMARK()` w celu obciążenia systemu zarządzania bazą danych).

W celu wyeliminowania niektórych skutków wykorzystania podatności SQL Injection należy:

- używać użytkownika bazy danych z ograniczonymi uprawnieniami (tylko do konkretnych tabel i funkcji, z ograniczonymi możliwościami pobierania i wstawiania danych),
- wyłączyć możliwość bądź uniemożliwić użycie wielu zapytań jednocześnie (czyli np. `SELECT * FROM users; SELECT * FROM orders;`),
- skonfigurować maksymalny czas wykonania zapytania (tzw. timeout albo `MAX_EXECUTION_TIME`).

Warto zauważyć, że wykradnięcie loginów i haseł pozwala **przejść konta**. Obrona przed przejęciem konta nie powinna polegać tylko na zabezpieczeniu aplikacji przed atakiem SQL Injection.

Hasła w bazie danych powinny być hashowane, aby nie dało się ich odtworzyć w formie jawnej. Proces hashowania jest nieodwracalny w porównaniu do szyfrowania.

Jednym z polecanych algorytmów hashujących do haseł jest **bcrypt**.

Pełna ochrona przed wstrzyknięciami SQL polega na wdrożeniu poniższych mechanizmów:

- Prepared Statement,
- Stored Procedure.

## Prepared Statement

Mechanizm *Prepared Statement* polega na „przygotowywaniu zapytania SQL”. Zapytanie do bazy danych budowane jest w kilku krokach:

1. Przygotowanie zapytania bez parametrów.
2. Przypisanie parametrów do przygotowanego zapytania.
3. Wysłanie do systemu zarządzania bazą danych zapytania i (oddzielnie) parametrów.

Oryginalnie ten mechanizm służył do zwiększenia wydajności wykonywania wielokrotnie zapytań. Zapytanie zapisane jako Prepared Statement może być używane wielokrotnie, dla różnych parametrów.

Dużą zaletą tego mechanizmu jest to, że parametry są wysyłane oddzielnie. Nie mogą wpłynąć w żaden sposób na treść zapytania, bo nie będą nigdy wstawiane bezpośrednio do treści zapytania jako fragment kodu.

Zapytanie *Prepared Statement* może wyglądać następująco:

```
SELECT * FROM users WHERE user_name = ? AND password = ?;
```

System zarządzania bazą danych skompiluje to wyrażenie bez wykonywania go. Możliwe będzie wykonanie tego zapytania tak jak procedury - podając parametry jako argumenty. Skompilowany kod nie może już zostać zmodyfikowany, a więc jest pewność, że parametry nie wpłyną na oryginalny kod zapytania.

Następnie możliwe jest podpięcie dwóch parametrów - w tym przypadku tego, co użytkownik wpisał w polu „login” i „hasło”.

W przedstawionym przykładzie obydwa parametry są anonimowe - to, jaka wartość zostanie użyta w miejscu znaków zapytania, zależy od kolejności, w jakiej przypisywane są parametry. W „Prepared Statement” istnieje możliwość użycia **nazwanych parametrów**, które mogą poprawić czytelność.

Przykładowe zapytanie z użyciem nazwanych parametrów wygląda następująco:

```
SELECT * FROM users WHERE user_name = :username AND password = :password;
```

Dla takiego wyrażenia nie można przypisać parametrów anonimowo - parametry przypisuje się pod daną nazwą, ale za to w dowolnej kolejności. Pod „username” podpiną się to, co użytkownik wpisał w polu „login”, a pod „password” to, co użytkownik wpisał w polu hasło.

```
username -> $login  
password -> $password
```

W zapytaniu używa się nazw poprzedzonych dwukropkiem, ale podczas przypisywania nie używa się dwukropka.

## Stored Procedure

Możliwe jest też wykorzystanie procedur, które wcześniej zdefiniowało się w bazie danych. Zamiast wywoływać jawnie zapytanie SELECT, można je zawrzeć w procedurze w bazie danych.

```
DELIMITER //  
CREATE PROCEDURE SelectUser (username varchar(255), pass varchar(255))  
BEGIN  
    SELECT * FROM users WHERE user_name = username AND password  
= pass;  
END //
```

Procedury, podobnie jak *Prepared Statements*, są wcześniej kompilowane przez system zarządzania bazą danych, więc niemożliwe jest wstrzyknięcie kodu. Dodatkową zaletą procedur jest też to, że pozwalają na łatwą migrację tabel - aplikacja nie musi wiedzieć, z jakiej tabeli są pobierane dane, ponieważ ta informacja jest przechowywana wewnątrz procedury.

## Pobieranie danych z innej tabeli

Wykorzystując podatność SQL Injection, można nie tylko zmodyfikować warunki zapytania, ale także pobrać dane z innych tabel za pomocą złączeń.

Podstawowym złączeniem jest złączenie typu JOIN. Polega na powiązaniu danych z jednej tabeli z danymi z drugiej - tak, żeby zestaw wyników był uzupełniony o dodatkowe pola.



id	movie_name	order_date	user
1	The Godfather	2020-02-28	john
2	Titanic	2020-03-02	john

Ryc. 4. Tabela „movie\_orders”.

user_name	first_name	last_name	email	password
john	John	Doe	john@doe.com	weakpassword
jane	Jane	Smith	demo@user.com	demo

Ryc. 5. Tabela „users”.

W przykładzie zostaną wykorzystane przedstawione tabele „movie\_orders” i „users”. W pierwszej z nich przechowywane są dane o zamówieniach filmów, a w drugiej dane użytkowników.

Aby dowiedzieć się, jakie są adresy email osób, które zamówiły poszczególne filmy, należy wykonać **złączenie**. W tym przypadku trzeba wykonać poniższe zapytanie:

```
SELECT * FROM movie_orders LEFT JOIN users ON movie_orders.user = users.user_name;
```

id	movie_name	order_date	user	user_name	first_name	last_name	email	password
1	The Godfather	2020-02-28	john	john	John	Doe	john@doe.com	weakpassword
2	Titanic	2020-03-02	john	jane	Jane	Smith	demo@user.com	demo

Ryc. 6. Efekt złączenia tabel „movie\_orders” oraz „users”.

W efekcie zwrócone zostaną dane z „movie\_orders” uzupełnione o odpowiednie pasujące dane z „users”.

Niestety w większości przypadków nie uda się wykonać złączenia JOIN podczas ataku SQL Injection - wynika to z faktu, że klauzula WHERE musi występować po klauzuli JOIN, a nie na odwrót.

Poprawnym zapytaniem byłoby:

```
SELECT * FROM movie_orders LEFT JOIN users ON movie_orders.user = users.user_name WHERE user_name = 'john';
```

Ze względu na powyższe ograniczenie, podczas ataku zwykle korzysta się z innego złączenia – **UNION**. Unia rozszerza zestaw wyników o dane z drugiego zapytania - w rezultacie możliwe jest, aby wykonać kilka różnych zapytań SELECT i połączyć je w jeden zestaw

Jeżeli administrator bazy danych chciałby pobrać listę użytkowników, którzy zamówili film, wykonalibyśmy:

```
SELECT user FROM movie_orders;
```

Natomiast pobranie nazw zarejestrowanych użytkowników wyglądałoby następująco:

```
SELECT user_name FROM users;
```

Aby połączyć te dwa zapytania można użyć UNION ALL:

```
SELECT user FROM movie_orders  
UNION ALL  
SELECT user_name FROM users;
```

Efektem tego zapytania będzie wynik przedstawiony w formie tabeli przedstawionej na rysunku 7.

user	Legenda
john	Dane z tabeli movie_orders
john	
john	Dane z tabeli users
jane	

Ryc. 7. Złączenie kolumn user i user\_name z tabel movie\_orders oraz users.

Co ważne, łączenie danych za pomocą UNION ALL pozwala na używanie WHERE w obydwu częściach zapytania:

```
SELECT user FROM movie_orders  
WHERE id = 1  
UNION ALL  
SELECT user_name FROM users  
WHERE email = 'demo@user.com';
```

Dzięki temu możliwe jest użycie UNION ALL w ataku SQL Injection.

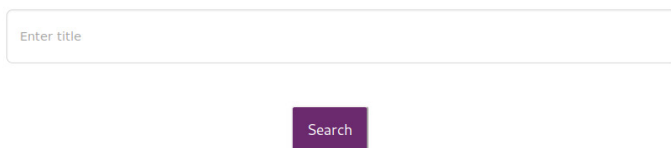
user
john
jane

Ryc. 8. Złączenie user i user\_name z tabel movie\_orders oraz users z użyciem warunków WHERE.

## Atak SQL Injection

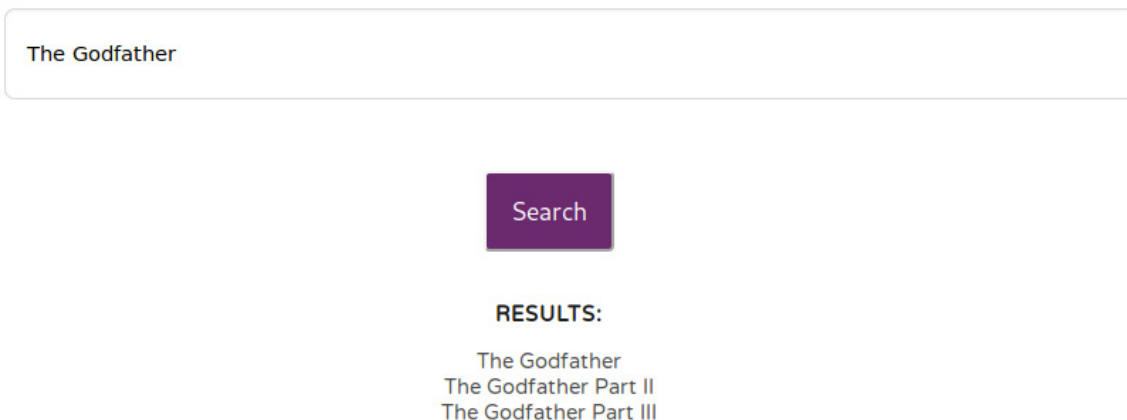
Atakowaną aplikacją, która posłuży do zaprezentowania sposobu działania SQL Injection, będzie prosta wyszukiwarka filmów.

### Movie Search



Ryc. 9. Wyszukiwarka filmów z podatnością SQL Injection.

Po wpisaniu wyszukiwanego tytułu wyświetla się lista z wynikami filmów obecnymi w bazie danych i zawierającymi wyszukiwaną frazę.



Ryc. 10. Wyniki wyszukiwania „The Godfather”.

Na bazie zachowania aplikacji, można domyślić się jak wygląda zapytanie do bazy danych. Możliwe jest, że do wyszukania filmów aplikacja używa poniższego zapytania:

```
SELECT title FROM movies WHERE title LIKE '%$title%';
```

W tym zapytaniu zamiast \$title wstawiane jest to, co zostało wysłane w formularzu.

Celem tego ataku będzie próba wykonania poniższego zapytania, które zwróci wersję bazy danych:

```
SELECT @@version
```

Aby poprawnie opracować atak SQL Injection, warto wyobrazić sobie, jakie zapytanie wysła aplikacja w momencie, gdy atak się powiedzie. Wstrzyknięcie kodu musi się odbyć w miejscu, w którym obecne jest wyrażenie \$title i jednocześnie efektem tego wstrzyknięcia powinno być wykonanie zapytania `SELECT @@version`. Kod SQL, który spełnia te warunki, może wyglądać następująco:

```
SELECT title FROM movies WHERE title LIKE '%The Godfather'  
UNION ALL SELECT @@version; -- comment%';
```

W pole wyszukiwarki należy wpisać to, co w zapytaniu zaznaczone jest kolorem niebieskim, czyli `The Godfather' UNION ALL SELECT @@version; -- comment`. Jeżeli aplikacja wykonuje zapytanie podobne do przedstawionego powyżej, wśród wyników wyszukiwania powinna znaleźć się wersja używanej bazy danych.

```
The Godfather" UNION ALL SELECT @@version; -- comment
```

Search

#### RESULTS:

```
The Godfather  
5.7.27-0ubuntu0.18.04.1
```

Ryc. 11. Udany atak SQL Injection.

Wynikiem wyszukiwania tej frazy są dwie pozycje: „The Godfather” oraz „5.7.27-0ubuntu0.18.04.1”. Wyszukanie w internecie drugiego elementu listy pozwala stwierdzić, że na aplikacja używa bazy danych MySQL 5.7.27 działającej w systemie Ubuntu 18.04.

Ta informacja może być kluczowa przy przeprowadzaniu kolejnych ataków, ponieważ język SQL nieznacznie różni się w zależności od używanego systemu zarządzania bazą danych. Dodatkowo niektóre starsze wersje systemów zarządzania bazą danych posiadają znane luki, które można odnaleźć w internecie i wykorzystać.

## „Ślepe” ataki SQL Injection

Niektóre aplikacje nie wyświetlają wyniku danych pobranych za pomocą wstrzyknięcia SQL. W takiej sytuacji należy przyjąć strategię „zgadywania” danych aż do momentu osiągnięcia pozytywnego wyniku.

Taki atak może zostać przeprowadzony na przykładowej aplikacji zawierającej formularz dodawania przepisów do bazy danych. Na początek warto jednak zbadać, jak aplikacja zachowa się w przypadku dodania przepisu o nazwie „Demo Recipe”.

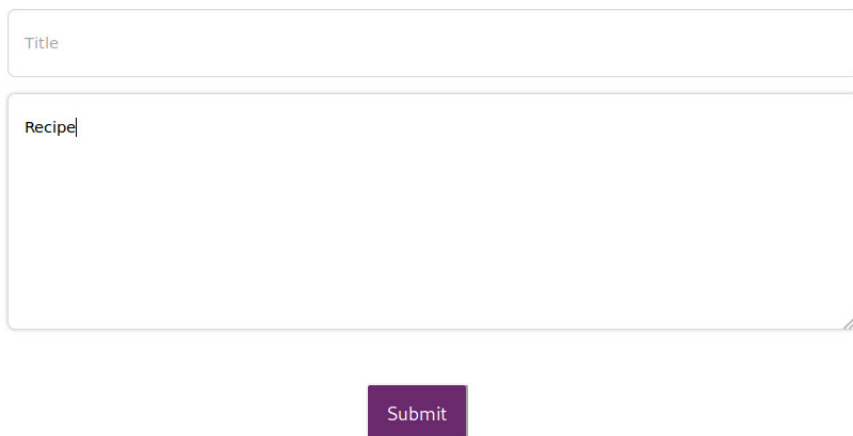
Po wysłaniu przepisu po raz drugi pojawia się napis „This title is already taken”. Po tym można wywnioskować, że system po wysłaniu formularza w pierwszej kolejności sprawdza, czy przepis istnieje. Zapytanie, które sprawdza nazwę, może wyglądać podobnie do poniższego:

```
SELECT title FROM recipes WHERE title = 'Demo Recipe';
```

Warto zastanowić się, czy nie można wykorzystać tego zapytania do przeprowadzenia ataku SQL Injection. Nie będzie możliwe zaprezentowanie wyników na stronie, ale sama podatność może zostać wykorzystana do odkrycia innych informacji. Taką informacją może być to, jacy użytkownicy istnieją w bazie danych.

## Recipe Submit Form

SUBMIT A RECIPE TO PUBLISH



Ryc. 12. Aplikacja do dodawania przepisów.

## Recipe Submit Form

SUBMIT A RECIPE TO PUBLISH

THIS TITLE IS ALREADY TAKEN



Ryc. 13. Efekt wysłania formularza z tytułem, który jest zajęty.

Aplikacja wyświetla komunikat „This title is already taken” bazując na tym, czy zapytanie SELECT zwraca jakikolwiek wynik. Jeżeli zbiór danych pasujących do WHERE title = 'Demo Recipe' nie jest pusty, zostanie wyświetlony komunikat o błędzie. W przeciwnym razie system nie wyświetli komunikatu.

Jeżeli w aplikacji występuje podatność SQL Injection w tym zapytaniu SELECT, możliwe będzie wykorzystanie jej do wykonania kodu. Aby opracować atak sprawdzający, czy istnieje użytkownik „admin”, można wykorzystać poniższe zapytanie:

```
SELECT title FROM recipes WHERE title = '' UNION ALL SELECT  
user_name FROM users WHERE user_name = 'admin'; -- comment'
```

Pierwsza część (pierwszy SELECT) pobierze dane z kolumny title z tabeli recipes dla wszystkich encji, których atrybut title jest pusty. Wynik tego zapytania powinien być zawsze pusty, o ile system nie dopuszcza pustych tytułów.

Druga część zapytania pobierze dane z kolumny user\_name z tabeli users dla wszystkich encji, których atrybut user\_name jest równy „admin”.

W przypadku, gdy użytkownik „admin” istnieje, to zapytanie zwróci niepusty zbiór danych. Aplikacja zinterpretuje to jako fakt, że przepis już istnieje i wyświetli komunikat „This title is already taken”. W przypadku, gdy użytkownik nie istnieje, ten komunikat nie powinien się pojawić.

Na podstawie powyższego zapytania można stwierdzić, że aby przeprowadzić atak należy wprowadzić poniższy tytuł:

```
' UNION ALL SELECT user_name FROM users WHERE user_name =  
'admin'; -- comment
```

**THIS TITLE IS ALREADY TAKEN**

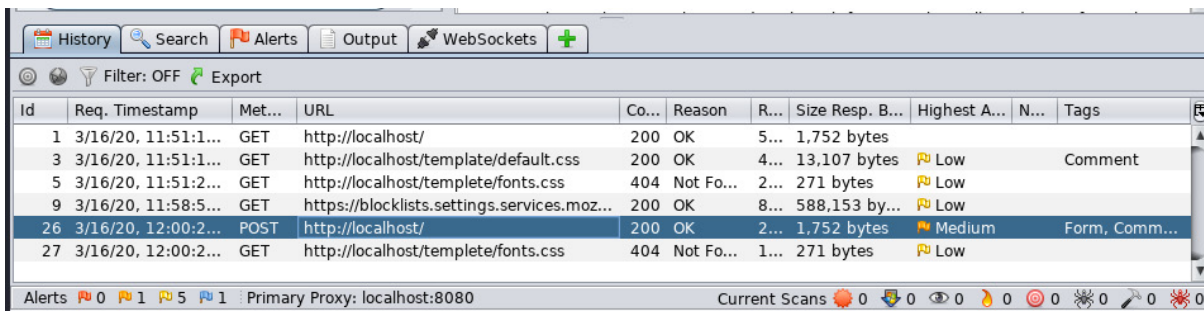
```
' UNION ALL SELECT user_name FROM users WHERE user_name = 'admin'; -- comment
```

Ryc. 14. Udały atak SQL Injection na formularz dodawania przepisów.

Ten atak się powiódł i udało się stwierdzić, że w bazie danych występuje użytkownik „admin”. Na bazie opracowanego tytułu można zautomatyzować szukanie użytkowników poprzez użycie narzędzia OWASP ZAP i słownika domyślnych użytkowników SecLists (<https://github.com/danielmiessler/SecLists/blob/master/Usernames/cirt-default-usernames.txt>).

Na początek należy przygotować wzór żądania w OWASP ZAP. W tym celu należy raz dodać przepis o nazwie „' UNION ALL SELECT user\_name FROM users WHERE user\_name = 'admin' ; -- comment”, ale w trybie „Manual Explore”.





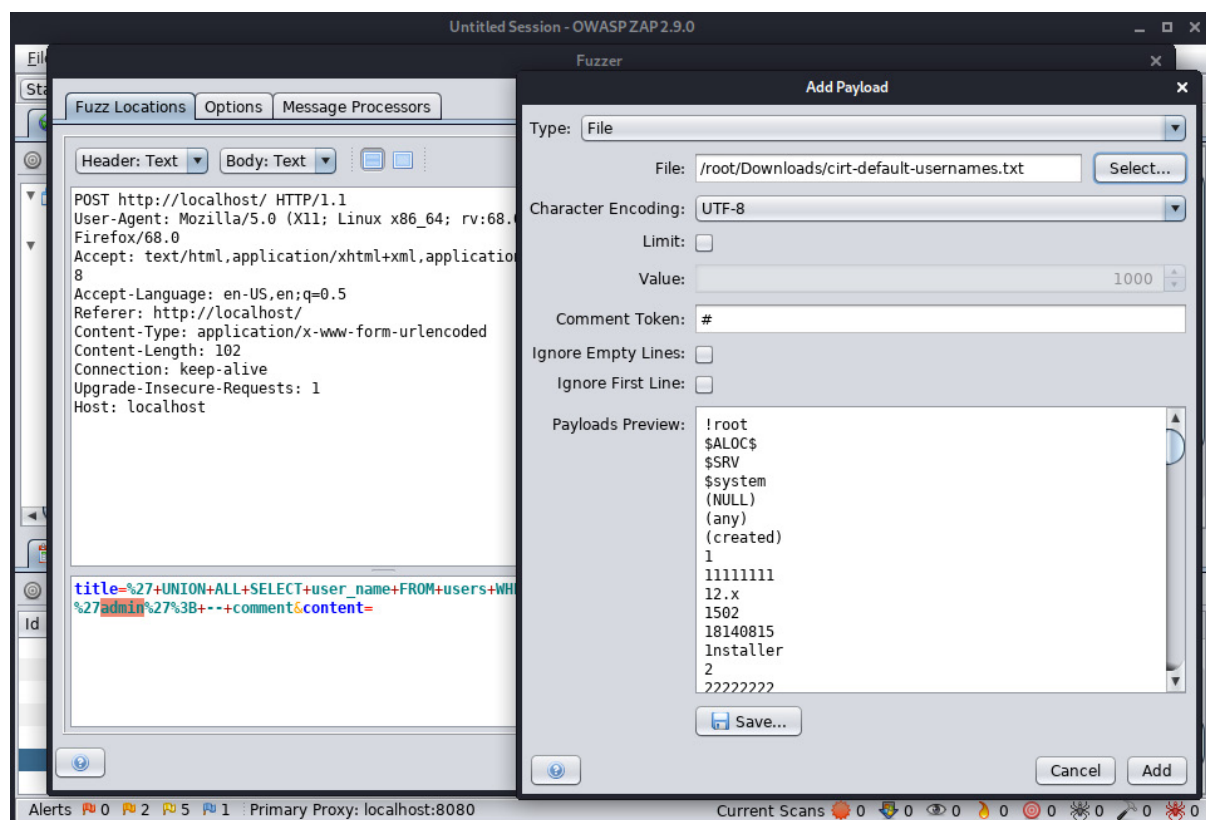
Id	Req. Timestamp	Met...	URL	Co...	Reason	R...	Size Resp. B...	Highest A...	N...	Tags
1	3/16/20, 11:51:1...	GET	http://localhost/	200	OK	5...	1,752 bytes			
3	3/16/20, 11:51:1...	GET	http://localhost/template/default.css	200	OK	4...	13,107 bytes	Low		Comment
5	3/16/20, 11:51:2...	GET	http://localhost/template/fonts.css	404	Not Fo...	2...	271 bytes	Low		
9	3/16/20, 11:58:5...	GET	https://blocklists.settings.services.moz...	200	OK	8...	588,153 by...	Low		
26	3/16/20, 12:00:2...	POST	http://localhost/	200	OK	2...	1,752 bytes	Medium		Form, Comm...
27	3/16/20, 12:00:2...	GET	http://localhost/template/fonts.css	404	Not Fo...	1...	271 bytes	Low		

Ryc. 15. Historia eksploracji aplikacji.

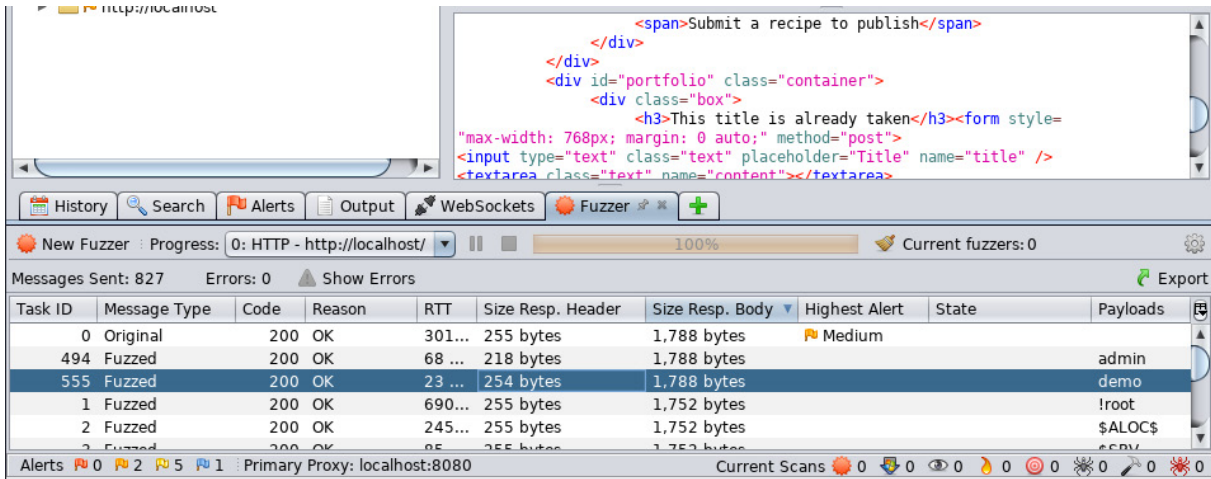
W historii pojawi się jedno żądanie POST. Można je wykorzystać do przygotowania ataku typu „Fuzz...”. W żądaniu lokalizacja ataku Fuzz zostanie ustawiona na słowo „admin”, aby testować różnych użytkowników pobranych z pliku *cirt-default-usernames.txt*.

Po wykonaniu ataku wyniki mogą zostać posortowane po rozmiarze odpowiedzi. W przypadku znalezienia użytkownika, na stronie pojawia się dodatkowe zdanie, które wpływa na rozmiar odpowiedzi.

Wyniki ataku pozwalają stwierdzić, że w systemie istnieje przynajmniej dwóch użytkowników - „admin” oraz „demo”. Używając podatności Blind SQL Injection udało się odnaleźć część użytkowników systemu. W tym przypadku brak bezpośredniej prezentacji wyników zapytania utrudnił, ale nie uniemożliwił ataku.



Ryc. 16. Przygotowywanie zautomatyzowanego ataku SQL Injection.



The screenshot shows the Burp Suite interface. The top pane displays the HTML source code of a web page, which includes a form for submitting a recipe. The bottom pane shows a table of messages sent during the fuzzing process.

Task ID	Message Type	Code	Reason	RTT	Size Resp. Header	Size Resp. Body	Highest Alert	State	Payloads
0	Original	200	OK	301...	255 bytes	1,788 bytes	Medium		
494	Fuzzed	200	OK	68 ...	218 bytes	1,788 bytes			admin
555	Fuzzed	200	OK	23 ...	254 bytes	1,788 bytes			demo
1	Fuzzed	200	OK	690...	255 bytes	1,752 bytes			!root
2	Fuzzed	200	OK	245...	255 bytes	1,752 bytes			\$ALOC\$
3	Fuzzed	200	OK	85 ...	255 bytes	1,752 bytes			\$CPV

Ryc. 17. Efekt zautomatyzowanego ataku SQL Injection.

