JOURNAL of COMPUTER SCIENCES INSTITUTE

JCSI 7 (2018) 150-157
Wysłane: 2017-11-28
Przyjęte: 2017-12-03

Analiza możliwości obrony przed atakami SQL Injection

Bogdan Krawczyński*, Jarosław Marucha, Grzegorz Kozieł
Politechnika Lubelska, Instytut Informatyki, Nadbystrzycka 36B, 20-618 Lublin, Polska

Streszczenie: Publikacja dotyczy ataków SQL Injection, które stanowią jedno z głównych zagrożeń w cyberprzestrzeni. W oparciu o badania literaturowe dokonano klasyfikacji ataków SQL Injection. Celem pracy było przeprowadzenie analizy możliwości obrony przed atakami SQL Injection. Metodykę badawczą oparto na autorskiej aplikacji, zaimplementowanej w technologii JSP (JavaServer Pages) z wykorzystaniem serwera baz danych MySQL.

Słowa kluczowe: Wstrzykiwanie kodu SQL; bezpieczeństwo danych; podatność aplikacji

*Autor do korespondencji.

Adres e-mail: bogdan.krawczynski@pollub.edu.pl

Analysis of protection capabilities against SQL Injection attacks

Bogdan Krawczyński*, Jarosław Marucha, Grzegorz Kozieł
Institute of Computer Science, Lublin University of Technology, Nadbystrzycka 36B, 20-618 Lublin, Poland

Abstract. Publication refers to SQL Injection attacks whose are one of the most dangerous in a cyberspace. Based on a literature studies, classification of the SQL Injection attacks was prepared. The purpose of the work was to analyse of protections effectiveness against SQL Injection attacks. Research method has been based on author application, which was implemented in JSP (JavaServer Pages) technology using MySQL database server.

Keywords: SQL Injection; data security; application vulnerability

*Corresponding author.

E-mail address: bogdan.krawczynski@pollub.edu.pl

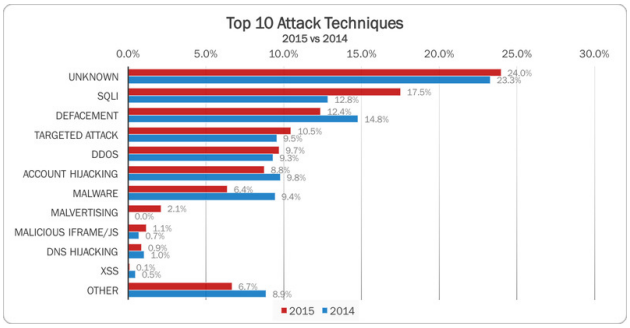
1. Wprowadzenie

Nasilenie globalizacji oraz konsekwentnie postępujący rozwój technologiczny spowodowały, że systemy teleinformatyczne towarzyszą nam niemal w każdej dziedzinie życia. Praktycznie wszystkie z nich, niezależnie od realizowanych funkcjonalności, opierają swoje działanie na gromadzeniu i/lub przetwarzaniu danych. W konsekwencji, doprowadziło to do sytuacji, w której dane stały się kluczowym zasobem.

Równolegle z procesem informatyzacji obserwujemy zjawisko zwane cyberprzestępczością. Pod tym pojęciem kryje się szereg nielegalnych działań dokonywanych przy pomocy Internetu. Łupem hackerów bardzo często stają się dane np. osobowe, przy pomocy których możliwe jest dokonywanie nieautoryzowanych transakcji (np. bankowych). Pomimo istnienia szeregu regulacji administracyjno-prawnych, w postaci dokumentów takich jak ustawa o ochronie danych osobowych, stale dochodzi do eskalacji liczby poszkodowanych oraz strat, powstałych w wyniku ataków hackerskich.

W związku z powyższym odpowiedzialność za bezpieczeństwo danych spoczywa na twórcach oprogramowania. Wynik pracy nie tylko programistów, ale również projektantów i testerów decyduje o końcowej jakości. Produkt dobry i konkurencyjny, to taki który jest użytkowy, ale przede wszystkim bezpieczny. Zapewnienie poufności, dostępności oraz integralności danych jest kluczowe dla każdego klienta docelowego. Niezależnie od tego czy jest nim osoba prywatna, urząd państwowy czy międzynarodowe przedsiębiorstwo.

Pierwsze doniesienia dotyczące ataków SQL Injection datowane są na rok 1998 [1]. Jednak pojęcie to zaczęło funkcjonować dopiero w roku 2000, po wprowadzeniu go przez Chipa Andrewsa po napisaniu i opublikowaniu artykułu „SQL Injection FAQ". Pomimo upływu ponad 15 lat, ataki SQL Injection nie tracą na „popularności" [2]. Wręcz przeciwnie – utrzymują się w czołówce ataków dokonywanych w cyberprzestrzeni, co przedstawiono na rysunku 1.



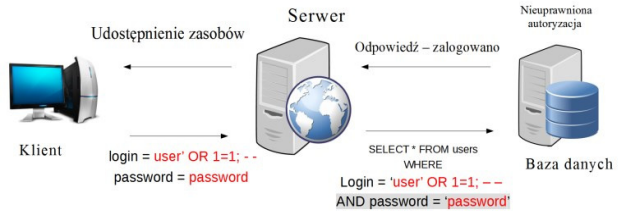
Rys. 1. Ranking 10 najczęstszych ataków w latach 2014-2015 [2]

OWASP (ang. *Open Web Application Security Project*) to światowa, otwarta i profesjonalna organizacja non-profit. Główny celem stowarzyszenia jest „poprawa bezpieczeństwa oprogramowania". Mając na celu obiektywizm w zakresie prowadzonych działań (badań, publikacji), społeczność unika wszelkich powiązań z firmami technologicznymi [3]. Raport organizacji OWASP z bieżącego roku, jako główne zagrożenie dla bezpieczeństwa aplikacji internetowych, wskazuje ich podatności na iniekcję kodu (w tym kodu SQL, tj. SQL Injection) [4].

2. SQL Injection – mechanika ataku

Na rysunku 2. przedstawiono logiczny schemat przeprowadzenia ataku SQL Injection. Aby skutecznie zabezpieczyć się przed atakami SQL Injection należy w pierwszej kolejności zrozumieć ich istotę. SQL Injection w najprostszy sposób można scharakteryzować jako atak na system bazy danych, polegający na wstrzyknięciu (ang. *injection*) niepożądanego kodu SQL za pośrednictwem aplikacji. Zainfekowany kod zostaje przekazany, a następnie wykonany przez serwer bazy danych. W konsekwencji napastnik jest w stanie w sposób nieautoryzowany wstawić, odczytać, zmodyfikować oraz usunąć dane (ang. *CRUD – Create Read Update Delete*). Oznacza to, że ataki SQL Injection naruszają trzy podstawowe własności bezpieczeństwa informacji [5]:

- dostępność – udostępnianie i umożliwienie modyfikacji danych, w odpowiedzi na żądanie uprawnionego użytkownika systemu, realizowane w określonym czasie,
- poufność – gwarancja ochrony danych przed ich nieautoryzowanym dostępem przez osoby, procesy lub inne podmioty,
- integralność – zabezpieczenie danych przed ich nieautoryzowaną modyfikacją lub usunięciem.



Rys. 2. Schemat logiczny ataku SQL Injection

Ponadto, w niektórych przypadkach, ataki SQL Injection mogą posłużyć do przeglądania, zapisu i odczytu plików lokalnych [6].

Aplikacje internetowe, niezależnie od technologii w której powstały, działają w architekturze klient-serwer, którą najlepiej opisuje reguła żądanie-odpowiedź (ang. *request-response*). Klient wysyłając żądanie może przesyłać do serwera wartości (np. formularz logowania), które następnie są wstawiane podczas generowania dynamicznych zapytań do bazy danych. To rodzi zagrożenie stwarzając sytuację, w której możliwe jest przesłanie łańcucha znaków, który zmodyfikuje oryginalną postać zapytania.

3. Klasyfikacja ataków SQL Injection

SQL Injection to obszerny termin, pod którym kryje się szereg różnych technik dokonywania ataku. Umiejętności, specjalistyczna wiedza, ale i kreatywność hackerów przekładają się na liczebność wariantów ataków. W ramach analizy możliwości obrony przed nimi, postanowiono skupić się najpopularniejszych spośród nich. Język SQL w znaczący sposób przekłada się na dostępne techniki ataków. Co więcej, samo wstrzyknięcie kodu SQL – przekazanie parametrów, za pośrednictwem aplikacji internetowej, może odbywać się na kilka sposobów.

Protokół HTTP (ang. *HyperText Transfer Protocol*) umożliwia dwukierunkową i znormalizowaną komunikację pomiędzy klientem a serwerem. Każdy nagłówek (ang. *HTTP header*) żądania klienta oprócz adresu URL żądanego zasobu może zawierać również parametry.

Parametry te mogą być przekazywane przy użyciu [7]:

- metody HTTP GET – za pośrednictwem adresu URL (ang. *Uniform Resource Locator*), czyli ciągu znakowego, w którym oprócz ścieżki żądanego zasobu, przekazywać można wartości parametrów.
- metody HTTP POST – która w porównaniu do poprzedniej, cechuje się większym poziomem bezpieczeństwa, ponieważ przesyła wartości parametrów w sekcji body nagłówka HTTP. Najczęściej są to wartości wprowadzane przez użytkownika w formularzu.
- nagłówka HTTP – w którym, oprócz powyższych metod oraz metadanych, przesyłane są również inne dane na podstawie, których dokonywana jest optymalizacja wyświetlanej strony.
- plików cookie – (ang. *HTTP cookies*), które bardzo często wykorzystywane są np. do utrzymania sesji pomiędzy klientem a serwerem. Bezstanowość protokołu HTTP stanowi pewną przeszkodę dla twórców aplikacji internetowych. Przy pomocy ciasteczek możliwa jest personalizacja serwisu WWW [6].

Strukturalny język zapytań (ang. *Structured Query Language*) umożliwia tworzenie i modyfikowanie baz danych. Ponadto pozwala wstawiać, manipulować oraz usuwać dane. Każda z powyższych operacji odbywa się w analogiczny sposób i jest realizowana przy pomocy zapytania wykonywanego przez serwer bazy danych. Struktura zapytań SQL, podobnie jak sekwencja jego wykonania, jest precyzyjnie określona. Podczas próby wykonania zapytania z błędem składniowym lub niekompatybilnym typem zmiennych, wyświetlony zostaje stosowny błąd.

Ataki SQL Injection można podzielić na trzy główne typy [8]:

1) In-band (wewnątrzpasmowy) – najprostszy i najpopularniejszy typ ataku. Napastnik, przeprowadzając atak, jest w stanie zaobserwować jego rezultat za pośrednictwem tego samego kanału komunikacyjnego. Przykładem takiego ataku może być wyświetlenie dodatkowych informacji przy pomocy operatora UNION, bezpośrednio w aplikacji.
2) Out-of-band (pozapasmowy) – znacznie mniej popularny typ ataku, podczas którego rezultat zwrócony zostaje innym kanałem komunikacyjnym. Alternatywnymi sposobami przekazania wykradzionych informacji może być żądanie HTTP lub DNS, czy nawet wiadomość mail [6].
3) Inference (inferencyjny) – w odróżnieniu od pozostałych, atak nie zwraca danych z bazy danych w sposób bezpośredni. Niemniej jednak, napastnik wstrzykując kod SQL analizuje czas odpowiedzi oraz reakcje aplikacji. W ten sposób jest on w stanie zgrodzmadzić szereg istotnych informacji np. dotyczących struktury bazy danych.

4. Możliwości obrony

Szerokie spektrum różnych technik ataków SQL Injection przekłada się na sytuację, w której nie istnieje jedno, w pełni skuteczne zabezpieczenie przed nimi [9]. Poszczególne sposoby obrony cechuje różny poziom skuteczności. W celu zapewnienia możliwie jak najwyższego poziomu ochrony aplikacji, zalecane jest zastosowanie kompleksowych rozwiązań obronnych. W kontekście bazodanowych aplikacji internetowych jest to jednoznaczne z koniecznością obrony na poziomie: klienta, serwera aplikacji oraz serwera bazy danych.

Zapytania sparametryzowane

Wiele publikacji [6, 8, 9, 10] dotyczących możliwości obrony przed atakami SQL Injection, zaleca stosowanie zapytań sparametryzowanych (ang. *prepared statement*). Predefiniowanie zapytań SQL gwarantuje większe bezpieczeństwo niż dynamicznie generowane zapytania, zawierające dane od użytkownika [6]. Programista, tworząc szkielet zapytania SQL, uniemożliwia napastnikowi modyfikację jego struktury. Znaki zapytania umieszczone wewnątrz tego szkieletu, pozwalają dołączyć wprowadzone przez użytkownika parametry. Zapytania sparametryzowane nie umożliwiają wstawiania nazw tabeli, kolumn czy też wskaźników sortowania (ASC – rosnąco; DESC – malejąco) [9].

Filtrowanie danych

Najbardziej naturalnym i oczywistym sposobem zapobiegania atakom SQL Injection wydaje się być filtrowanie danych wprowadzanych przez użytkownika. Ograniczając się wyłącznie do sprawdzenia poprawności typu wprowadzonych danych, możliwe jest uchronienie się przed wieloma atakami SQL Injection [11]. Jak wspomniano wcześniej – użycie zapytań sparametryzowanych nie zawsze jest możliwe. Wtedy zalecane jest użycie tzw. białej listy (ang. *white list*), która umożliwia użytkownikowi wprowadzanie wyłącznie dozwolonych znaków.

Istnieje również inny wariant obrony w ramach walidacji danych, polegający na wykorzystaniu tzw. czarnej listy. Za jej pośrednictwem możliwe jest przeszukanie wartości parametrów, pod kątem niedozwolonych znaków i wyrażeń. Wśród nich znaleźć się mogą znaki niedrukowalne (ang. *whitespaces*), znaki specjalne (apostrofy, myślniki, średniki itd.) oraz słowa kluczowe języka SQL. Takie rozwiązanie nie jest optymalne i może generować sytuacje problematyczne. Np. podczas próby rejestracji w systemie użytkownika, którego nazwisko zawiera apostrof. Kluczowe, z punktu widzenia bezpieczeństwa, jest jednoczesne stosowanie walidacji danych po stronie klienta oraz serwera. Ograniczanie się do walidacji po stronie klienta jest złą praktyką, która przekłada się na podatność aplikacji na przeprowadzenie ataku SQL [12].

Procedury składowane

Procedury składowane to struktury, które w ramach jednego wywołania realizują określoną sekwencję instrukcji. Można je porównać do funkcji występujących w językach programowania wysokiego poziomu. Struktury te są przechowywane po stronie serwera bazy danych [6]. Dzięki temu gwarantują zwiększoną wydajność oraz większy poziom bezpieczeństwa [9].

Deklarując własną procedurę składowaną, programista może zdefiniować parametry wejściowe, wymuszając tym samym odpowiedni typ zmiennych. Zabezpiecza to warstwę danych przed wstrzyknięciem kodu przy pomocy parametru pobieranego od użytkownika. Co więcej, używając parametrów wyjściowych (ang. *output*) programista jest w stanie precyzyjnie określić formę zwracanego wyniku. Takie podejście uniemożliwia serwerowi bazy danych wyświetlenie dodatkowych danych napastnikowi.

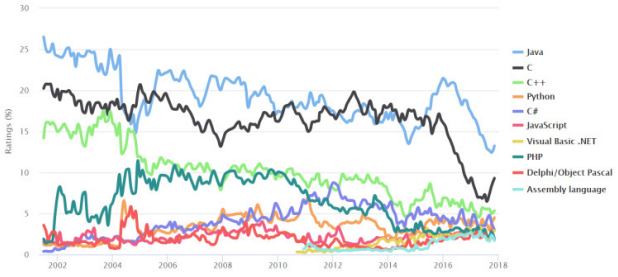
Zasada najmniejszego uprzywilejowania

Wyżej wymienione mechanizmy obronne mają na celu zablokowanie (niedopuszczenie) ataku SQL Injection. W odróżnieniu od nich, zasada najmniejszego uprzywilejowania (ang. *principle of least privilege*) skupia się głównie na minimalizacji strat powstałych w wyniku ataku SQL Injection. Zasada wymusza nadawanie możliwie najmniejszych (ale wystarczających) uprawnień każdemu elementowi systemu realizującego dane zadanie [13]. Jest to dobra praktyka znajdująca zastosowanie nie tylko w systemach teleinformatycznych, ale i w jednostkach wojskowych czy służbach specjalnych.

Stosując zasadę najmniejszego uprzywilejowania w odniesieniu do bazodanowej aplikacji internetowej, nie wystarczy skupić się na odseparowaniu poszczególnych jej modułów. Systemy zarządzania bazami danych umożliwiają tworzenie różnych profili użytkowników, posiadających różne poziomy uprawnień. Administrator systemu, tworząc profil użytkownika, powinien nadać mu możliwie najmniejsze uprawnienia, umożliwiające realizacje funkcjonalności. Klient, który w założeniu, ma mieć dostęp wyłącznie do wyświetlania danych, nie powinien mieć uprawnień do ich modyfikacji, czy usuwania [6].

5. Metodyka badawcza

W celu przeprowadzenia analizy możliwości obrony przed atakami SQL Injection postanowiono stworzyć autorską aplikację internetową. Implementacja przeprowadzona została w taki sposób, aby uzyskać możliwie jak największą podatność aplikacji na ataki typu SQL Injection. W oparciu o przeprowadzone badania literaturowe wyselekcjonowano najpopularniejsze techniki ataków, które szczegółowo przebadano wykorzystując do tego celu niezabezpieczoną wersję aplikacji testowej. W kolejnym kroku, do aplikacji dodano mechanizmy obronne opisane w rozdziale 4. Na koniec ponownie przeprowadzono serię ataków, w obu przypadkach badając „zachowanie" aplikacji oraz operacje przeprowadzone w bazie danych.



Rys. 3. Ranking 10 najpopularniejszych języków programowania na przestrzeni ostatnich 15 lat [14]

Z uwagi na szeroki zakres zagadnienia jakim są ataki SQL Injection, konieczne było zawężenie obszaru badawczego. Głównym kryterium jakim kierowano się podczas wyboru narzędzi była ich popularność. Aplikacja zaimplementowana została w technologii JSP (ang. *JavaServer Pages*) z wykorzystaniem serwera aplikacyjnego Apache Tomcat, oraz serwera bazodanowego MySQL. Java to obiektowy język programowania, utrzymujący się w czołówce rankingów popularności na przestrzeni ostatnich 15 (rysunek 3.) [14].

Przed przystąpieniem do przeprowadzenia eksperymentu, na podstawie przeprowadzonych badań literaturowych, sformułowano następujące hipotezy badawcze:

1. Prewencja umożliwia obronę przed najpopularniejszymi technikami ataków SQL Injection.
2. Stosowanie zasady najmniejszego uprzywilejowania minimalizuje straty.
3. Zapytania parametryzowane, spośród dostępnych rozwiązań, cechują się najwyższym poziomem bezpieczeństwa.

6. Eksperyment

W tym rozdziale przedstawione zostały różne typy ataków SQL Injection. Opisy poszczególnych (wybranych) technik zostały uzupełnione o konkretne warianty poszczególnych ataków. Pierwsza faza polegała na przeprowadzeniu serii ataków na wersję aplikacji całkowicie podatnej na ataki SQL Injection. Co więcej, każdy atak w tej fazie, przeprowadzony został z poziomu użytkownika o nieograniczonych uprawnieniach.

Tautologia

Cel ataku: Wyświetlanie danych, Sprawdzanie podatności parametrów, Nieuprawniona autoryzacja

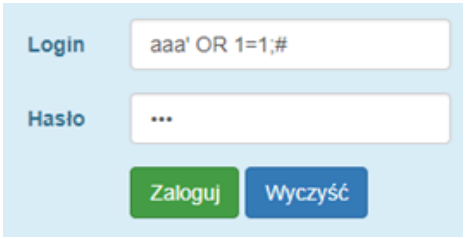
Atak polega na modyfikacji oryginalnej treści zapytania do postaci, w której staje się ono prawdziwie niezależnie od wartości pozostałych parametrów. W logice tautologia, to zdanie prawdziwe na mocy swojej budowy. W kontekście składni języka SQL oznacza to, że niezależnie od warunków zawartych w klauzuli WHERE ich wypadkowa wartość zawsze będzie równa prawdzie. Zdecydowana większość aplikacji internetowych funkcjonujących w sieci WWW, w procesie autoryzacji wymaga od użytkownika podania loginu i hasła. Następnie na podstawie wprowadzonych przez użytkownika danych odbywa się jego identyfikacja.

Przykład 1. Zapytanie SQL – autoryzacja

```
SELECT * FROM uzytkownicy WHERE
 login = 'podanyLogin' AND
 haslo = 'podaneHaslo';
```

Zapytanie realizujące pobranie rekordu z tabeli *uzytkownicy* o podanym przez użytkownika loginie i haśle, przedstawiono na przykładzie 1. W przypadku podania poprawnego loginu oraz hasła – czyli wartości dokładnie takich, które znajdują się w odpowiednich kolumnach tabeli *uzytkownicy*, identyfikacja zostanie zakończona pomyślnie. W przypadku podania niepoprawnych danych w formularzu logowania – zapytanie nie zwróci żadnego rekordu – brak użytkownika o podanym loginie i haśle w tabeli *uzytkownicy*.

Analizując strukturę powyższego zapytania należy zauważyć, że fragment „*SELECT * FROM uzytkownicy*” zostanie wykonany tylko i wyłącznie w momencie, gdy prawdziwy będzie warunek występujący po klauzuli WHERE. Umiejętna manipulacja oryginalnego kodu SQL, z wykorzystaniem operatora logicznego OR oraz dopisaniu do niego zawsze prawdziwego warunku, pozwala obejść proces autoryzacji.



Rys. 4. Formularz logowania – tautologia

Możliwy wariant takiego ataku przedstawiono na rysunku 4., natomiast przykład 2. zawiera treść zapytania generowanego dynamicznie w oparciu o wprowadzone parametry.

Przykład. 2. Zapytanie SQL – tautologia

```
SELECT * FROM uzytkownicy
WHERE login='aaa' OR 1=1;
  #' AND haslo='xxx';
```

W rezultacie pobrane z bazy danych zostaną wszystkie rekordy z tabeli *uzytkownicy*, a aplikacja wyświetli stronę glowna.jsp, co przedstawiono na rysunku 5. Jak widać, napastnik dostał nieuprawniony dostęp do listy wszystkich studentów, ponieważ został zalogowany na konto administratora.

Rys. 5. Tautologia – rezultat ataku na formularz logowania

UNION SELECT

Cel ataku: Nieuprawniona autoryzacja, Wyświetlenie danych, Odtwarzanie struktury bazy danych

Język SQL umożliwia łączenie wyników dwóch lub więcej zapytań, do jednej postaci wynikowej. Struktura takiego zapytania została przedstawiona na przykładzie 3.

Przykład 3. Zapytanie SQL – struktura UNION SELECT

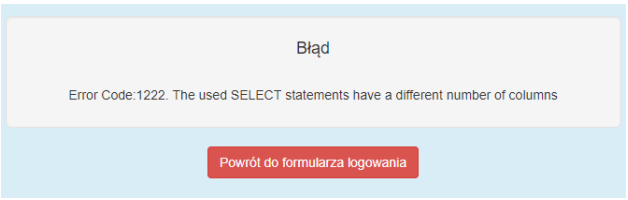
```
SELECT kol_1, kol_2,(...),kol_X FROM tab_1
UNION
SELECT kol_1, kol_2,(...),kol_X FROM tab_2;
```

Aby zapytanie z operatorem UNION było poprawnie wykonane, konieczne jest spełnienie dwóch warunków:

1. Wszystkie zapytania połączone operatorem UNION muszą pobierać dane z dokładnie tej samej liczby kolumn,

2. typy danych odpowiadających sobie kolumn muszą być takie same lub kompatybilne.

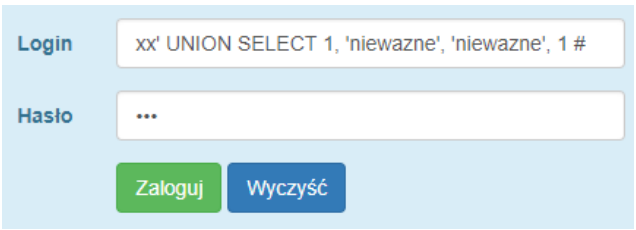
Jeśli warunki nie zostaną spełnione, zwrócony zostanie wyjątek, a zapytanie nie wykona się. Treść wiadomości wyświetlanej w aplikacji po podaniu niewłaściwej liczby kolumn przedstawiono na rysunku 6.



Rys. 6. Błąd – UNION SELECT – niepoprawna liczba kolumn

Z treści wiadomości nie wynika wprost informacja o liczbie kolumn z oryginalnego zapytania. Oznacza to, że w celu ustalenia tej liczby, napastnik musi zastosować metodę prób i błędów. To, w większości przypadków, oznacza konieczność generowania dużej liczby zapytań. Hackerzy bardzo często automatyzują ten proces pisząc własne skrypty lub korzystając z istniejących narzędzi [6].

Innym zastosowaniem techniki UNION SELECT jest obejście procesu autoryzacji. Omawiając atak z użyciem tautologii przybliżony został przebieg procesu autoryzacji użytkownika. Odbywa się ona na podstawie wprowadzonych przez użytkownika parametrów: login oraz hasło.



Rys. 7. Formularz logowania – UNION SELECT – obejście autoryzacji

Na rysunku 7. przedstawiono możliwy wariant ataku UNION SELECT. Wprowadzając frazę *„xx' UNION SELECT 1,'niewazne','niewazne', 1#"* w polu *login*, dokonana zostanie modyfikacja oryginalnego zapytania SQL, do postaci przedstawionej na przykładzie 4.

Przykład 4. UNION SELECT – obejście autoryzacji – zapytanie SQL

```
        select * from uzytkownicy where login='xx'
UNION SELECT 1, 'niewazne', 'niewazne', 1 #' and haslo='';
```

W wyniku tego, zamiast pustej listy zwrócony zostanie rekord o wartościach: 1, niewazne, niewazne, 1, co przedstawiono na rysunku 8. W konsekwencji napastnik zostanie zalogowany do systemu na konto z poziomem uprawnień administratora systemu.

	idUzytkownik	login	haslo	rodzajUprawnien
	1	niewazne	niewazne	1

Rys. 8. UNION SELECT – autoryzacja, wynik zapytania

Piggy-backed

Cel ataku: Wstawianie i modyfikowanie danych, Wyświetlanie danych, Wykonanie poleceń zdalnych

Opisane wyżej formy ataków skupiały się na modyfikacji kodu oryginalnego zapytania. Technika *piggy-backed* umożliwia wstrzyknięcie dodatkowego zapytania SQL, które zostaje wykonane bezpośrednio po oryginalnym zapytaniu. Szansa powodzenia tego ataku, jest uwarunkowana przede wszystkim konfiguracją serwera bazy danych [11]. Domyślnie MySQL nie umożliwia wykonywania serii zapytań [15]. Niemniej jednak, podczas nawiązywania połączenia z bazą danych, możliwe jest odblokowanie tej opcji. Przykład 5. przedstawia nawiązanie połączenia wraz z odblokowaniem opcji wykonywania wielu zapytań (ang. *multiple queries*).

Przykład 5. Odblokowanie opcji wykonywania wielu zapytań

```
("jdbc:mysql://localhost:3306/mojabaza?
allowMultiQueries=true", "root", "root");
```

Technika piggy-backed daje napastnikowi dużą elastyczność. Zakres możliwych działań jest uwarunkowany rodzajem uprawnień użytkownika, których dokonuje się konfigurując serwer bazy danych. Jeżeli użytkownik posiada uprawnienia do modyfikowania czy usuwania danych, może się to wiązać z nieodwracalną utratą danych. Poniżej przedstawiono możliwy wariant ataku piggy-backed.

Przykład 6. Piggy-backed – usunięcie tabeli *studenci*

```
SQLProjection/edytujstudenta.jsp?id=7'; drop table studenci; #
```

Przykład 6. przedstawia fragment kodu, który wstrzyknięto za pośrednictwem adresu URL. W rezultacie, zapytanie SQL przekazane do bazy danych wyglądać będzie tak, jak zaprezentowano to na przykładzie 7.

Przykład 7. Piggy-backed – usunięcie tabeli *studenci* - zapytanie SQL

```
SELECT * FROM studenci
WHERE idStudent ='7';
drop table studenci; #';
```

Powyższy przykład bardzo dobrze obrazuje zagrożenie, jakie niesie za sobą umożliwienie realizacji więcej niż jednego zapytania. Po wykonaniu oryginalnie zaprojektowanego zapytania wyświetlającego studenta o podanym identyfikatorze, cała tabela *studenci* zostanie usunięta. Warto zaznaczyć, że napastnik po znaku średnika, może dopisywać kolejne zapytania.

Blind SQL Injection

Cel ataku: Sprawdzanie podatności parametrów, Odtwarzanie struktury bazy danych, Wyświetlanie danych

Tzw. „ślepy atak przeprowadzany jest najczęściej, gdy napastnik nie dysponuje żadnymi informacjami dotyczącymi bazy danych (system zarządzania bazą danych, wersja, struktura itd.) [6]. Z reguły jest to mozolny i długotrwały proces, polegający na wstrzykiwaniu kolejnych zapytań zbudowanych przy pomocy operatorów logicznych (AND/OR). Na podstawie zwracanych rezultatów (prawda lub fałsz) napastnik sukcesywnie gromadzi kolejne informacje, począwszy od sprawdzenia czy wstrzykiwanie kodu SQL w danym systemie jest w ogóle możliwe. W przypadku standardowego ataku SQL Injection, wynik zapytania jest wyświetlany przez przeglądarkę. Natomiast stosując „ślepy" napastnik nie dostaje wyników w sposób jawny. Poniżej przedstawiono możliwe warianty tzw. ślepego ataku.

Error based

Korzystając z konstrukcji przedstawionej na przykładzie 8. możliwe jest ustalenie numeru wersji serwera bazy danych, z wykorzystaniem tzw. ślepego ataku bazującego na błędach.

Przykład 8. Konstrukcja do sprawdzenia numeru wersji serwera

```
3' AND substring(version(),1,1)=4;#
```

W rezultacie, wykonany zostanie następujący kod SQL – przykład 9.

Przykład 9. Blind SQL – zapytanie SQL sprawdzające numer wersji serwera

```
SELECT * FROM studenci
WHERE idStudent ='3'
AND substring(version(),1,1)=5;# '";
```

Jeśli nastąpi przekierowanie do widoku studenta o identyfikatorze 3, to numer wersji serwera bazy danych zaczyna się od cyfry '5'. Jeśli nie, napastnik może manipulować cyfrą wersji, do momentu ustalenia właściwej.

Time based

Podobny mechanizm występuje w przypadku tzw. ślepych ataków bazujących na czasie odpowiedzi. Napastnik w analogiczny sposób ustala istotne informacje, tym razem analizując czas odpowiedzi. Funkcja *BENCHMARK* umożliwia realizację danej instrukcji określona liczbę razy. Stosując technikę tzw. ślepego ataku bazującego na czasie odpowiedzi, hackerzy bardzo często wykorzystują funkcję SLEEP, który „usypia" serwer na określony czas (podany w sekundach). Możliwy wariant takiego ataku przedstawiono na przykładzie 10.

Przykład 10. Blind SQL – zapytanie SQL sprawdzające numer wersji serwera

```
1 UNION SELECT IF
(SUBSTRING(version(),1,1)=5,
BENCHMARK(5000000,
MD 5(CHAR(1))),null),null
```

Modyfikacja URL

Bardzo często parametry przesyłane przez adres URL są kodowane przy użyciu kodowania procentowego (ang. *percent encoding*). Każdy bajt zamieniony zostaje na jego dwucyfrowy odpowiednik, poprzedzony znakiem procenta (%). Korzystając z dostępnych w Internecie źródeł, możliwe jest zakodowanie fragmentu kodu SQL np. do postaci przedstawionej na rysunku 10.

http://localhost:8084/SQLProjection/delete.jsp?
id=25%35%27%20%4F%52%20%27%35%27%3D%27%35
||
25' OR '5' = '5

Rys. 10. Wstrzyknięcie kodu – URL, kodowanie procentowe

W rezultacie do zapytania realizującego (w założeniu) usunięcie studenta o podanym identyfikatorze (25), dodana zostanie fraza „*OR '5' = '5*". Przedstawia to przykład 11.

Przykład 11. Wstrzyknięcie kodu – URL, zapytanie SQL

```
DELETE FROM studenci WHERE idStudent ='25' OR '5' = '5';
```

Druga faza eksperymentu przeprowadzona została w sposób analogiczny do fazy pierwszej. Poniżej uwzględniono poszczególne mechanizmy obronne.

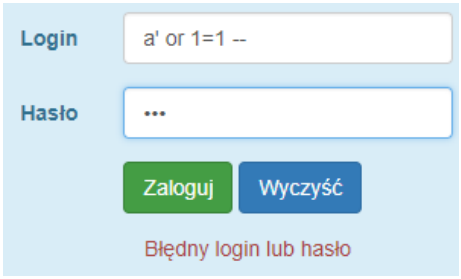
Zapytania parametryzowane

Na przykładzie 12. zaprezentowano fragment kodu źródłowego aplikacji, realizującego autoryzacje użytkownika (logowanie). W odróżnieniu od wcześniejszego rozwiązania, do wykonania zapytania wykorzystano zapytanie sparametryzowane. Dzięki niemu zapytanie nie jest tworzone na zasadzie konkatenacji ciągu znakowego. Szkielet zapytania wyposażony w znaki zapytania (symbole zastępcze), umożliwia wstawienie wartości pobranych od użytkownika.

Przykład 12. Zapytanie sparametryzowane – autoryzacja użytkownik

```
String sqlQuery = "SELECT * FROM uzytkownicy
WHERE login = ? and haslo = ? ";
PreparedStatement pstmt = conn.prepareStatement(sqlQuery);
pstmt.setString(1, login);
pstmt.setString(2, haslo);
ResultSet rs = pstmt.executeQuery();
```

Na załączonym przykładzie zaobserwować można również dwuetapowy mechanizm wykonania zapytania SQL. Funkcja *prepareStatement* realizuje wstępne wykonanie zapytania, do którego następnie dołączone są brakujące wartości parametrów. W ten sposób uniemożliwiona zostaje potencjalna ingerencja użytkownika w strukturę zapytania SQL.



Rys. 11. Zapytania parametryzowane – nieudana próba ataku przy użyciu tautologii

Procedury składowane

Stosowanie procedur składowanych to dobra praktyka, która pozwala poprawić nie tylko bezpieczeństwo, ale i wydajność aplikacji. Stosowanie parametrów wejściowych pozwala zadbać o kompatybilność typów danych. Dodatkowo pozwala zablokować próbę wykonanie zapytania w przypadku wprowadzenia np. zbyt długiego ciągu znaków dla parametru typu *varchar*. Jest to bardzo istotne, ponieważ wstrzyknięcia kodu SQL z reguły są dosyć rozbudowane

Przykład 13. Procedura wyszukująca studenta o podanym nazwisku

```
CREATE DEFINER=`root`@`localhost`
PROCEDURE `wyszukiwarkaAdmin`(IN vnazwisko varchar(20))
BEGIN
    SELECT idStudent, imie, nazwisko, wydzial, kierunek
    FROM studenci WHERE nazwisko LIKE CONCAT('%',vnazwisko,'%');
END
```

Przykład 13. przedstawia procedurę realizującą wyszukanie studenta o podanym nazwisku. Zapis „(IN vnazwisko varchar(20))" oznacza, że procedura przyjmuje jeden parametr wejściowy (pobierany od użytkownika), typu varchar, nie dłuższy niż 20 znaków. Podczas próby podania

dłuższego ciągu znaków, próba wykonania procedury zakończy się niepowodzeniem.

Docelowo, warto zadbać o zdefiniowanie parametrów wyjściowych, ich typu i zakresu. Jest to dodatkowe zabezpieczenie, które udaremni ewentualną próbę wyprowadzenia dodatkowych danych.

Filtrowanie danych

Według przeglądu literatury z zakresu ataków SQL Injection, wynika, że filtrowanie danych to mechanizm obronny niegwarantujący maksymalnego poziomu zabezpieczeń. Niemniej jednak zaleca się stosowanie filtrowania. Należy pamiętać o tym, że ograniczenie się do walidacji danych po stronie klienta, nie jest wystarczające. Znacznie bardziej znacząca jest walidacja po stronie serwera. Taki stan rzeczy wynika z tego, że istnieją narzędzia, które umożliwiają spreparowanie żądania HTTP.

Przykład 14. Funkcja walidująca długość parametru

```
private boolean czyZwalidowany1(String parametr) {
    return parametr.length() > 50 ? false : true;
}
```

Przykład 14. przedstawia kod funkcji odpowiedzialnej za sprawdzenie, czy wartość podanego parametru jest typu string (tekstowego) i jest nie dłuższa niż 50 znaków. Oprócz tego, konieczne jest „przeszukanie" parametru pod kątem słów kluczowych języka SQL, czy znaków specjalnych (np. #, -, --, spacja, ‘,"). Funkcja zaprezentowana na przykładzie 15. jest bardziej rozbudowana. Dzięki niej walidowana jest nie tylko długość, ale i zawartość przekazywanego parametru. Dopuszcza ona wprowadzenie wyłącznie (małych i dużych) liter. Jest to przykład zastosowania tzw. białej listy.

Przykład 15. Funkcja walidująca długość i zawartość parametru

```
private boolean czyZwalidowany2(String parametr) {
Pattern p = Pattern.compile("^[a-zA-Z]{0,20}$");
Matcher m = p.matcher("parametr");
return m.find() ? true : false;
}
```

Maskowanie błędów

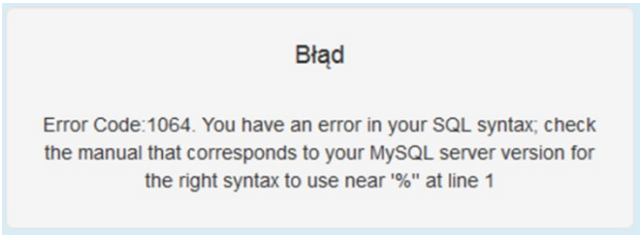
Implementując logikę aplikacji należy mieć świadomość, że dla potencjalnego napastnika treść zwracanych błędów posiada znaczący potencjał informacyjny. Aplikacja w wersji produkcyjnej powinna zwracać kompletne treści komunikatów (wraz z kodem błędu) wyświetlanych w sytuacjach wyjątkowych. Takie podejście pomaga programiście szybciej rozwiązywać problemy występujące podczas tworzenia oprogramowania.

Ważne, aby przed wdrożeniem aplikacji zadbać o umiejętne maskowanie błędów. Należy kierować się tym, aby użytkownik uzyskiwał taką porcję informacji, jaka jest mu potrzebna do realizacji danego zadania. Przykładowo, w sytuacji niepowodzenia autoryzacji (logowania) nie należy precyzować, który parametr jest nieprawidłowo (login, hasło, czy oba).

Poniżej zaprezentowano odpowiedź aplikacji podczas próby zalogowania z użyciem parametrów:

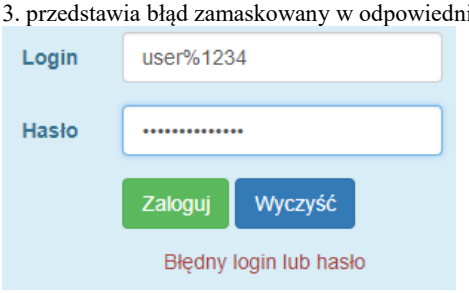
- Login: user%1234
- Hasło: xxxxxxxx

Na rysunku 12. zaprezentowano błąd wyświetlany w aplikacji podatnej na ataki SQL Injection.



Rys. 12. Błąd zwracany przez aplikację (niezamaskowany)

Rysunek 13. przedstawia błąd zamaskowany w odpowiedni sposób.



Rys. 13. Błąd zwracany przez aplikację (zamaskowany)

7. Wnioski

Przeprowadzone badania wykazały prawdziwość wszystkich hipotez badawczych sformułowanych w rozdziale 5. Spośród wszystkich wykorzystanych mechanizmów obronnych, najwyższą skutecznością wykazały się zapytania sparametryzowane oraz procedury składowane.

Istotnym wnioskiem uzyskanym w wyniku analizy jest fakt, że nieprawidłowa obsługa błędów stanowi podatność aplikacji na ataki SQL Injection. Implementując aplikację należy pamiętać o odpowiednim maskowaniu zwracanych błędów. Takie działanie w znaczący sposób utrudni działanie napastnikowi.

Na podstawie przeprowadzonych badań potwierdzono, że zasada najmniejszego uprzywilejowania pozwala dokonać minimalizacji strat. Co więcej, zgodnie z oczekiwaniami, pozwoliła ona zablokować jedynie niektóre ataki. uniemożliwiając przeprowadzenie modyfikacji danych, bez wymaganych uprawnień. W wyniku dyskusji, opracowano przykładowy model uprawnień użytkowników w systemie. Model przedstawiono w tabeli 1.

Tabela 1. Model uprawnień użytkowników systemu

Typ konta	Uprawnienia
użytkownik	SELECT
edytor treści	SELECT, INSERT, UPDATE
moderator serwisu	CREATE, DROP, ALTER, EXECUTE, DELETE
administrator	GRANT, BACKUP, CREATE USER, SHUTDOWN

Niewątpliwie przeprowadzone badania nie wyczerpują zagadnienia. W niniejszym opracowaniu skupiono się na analizie statycznych metod obrony przed atakami SQL Injection. Należy pamiętać o dynamicznych metodach ochrony aplikacji. Monitorowanie pracy serwera bazy danych z całą pewnością gwarantuje zwiększenie poziomu bezpieczeństwa. Prowadzona w czasie rzeczywistym detekcja i obsługa incydentów bezpieczeństwa w wielu

przypadkach pozwala zapobiec naruszeniu dostępności, poufności i integralności danych.

Literatura

[1] How Was SQL Injection Discovered? https://www.esecurityplanet.com/network-security/how-was-sql-injection-discovered.html [20.11.2017]

[2] Top 10 Attack Techniques – 2015 vs. 2014 http://www.hackmageddon.com/2016/01/11/2015-cyber-attacks-statistics [12.11.2017]

[3] Co oferuje nam OWASP? http://websecurity.pl/co-oferuje-nam-owasp [15.11.2017]

[4] OWASP: The 10 Most Critical Web Application Security Risks, https://www.owasp.org/images/b/b0/OWASP_Top_10_2017_RC2_Final.pdf

[5] Norma PN-SIO/IEC-17799:2005 Technika informatyczna. Praktyczne zasady zarządzania bezpieczeństwem informacji, PKN, 2007.

[6] J. Clarke, SQL Injection Attacks and Defense, Syngress Publishing, Inc., 2012.

[7] SQL Injection through HTTP Headers, http://resources.infosecinstitute.com/sql-injection-http-headers [14.11.2017]

[8] Amirmohammad Sadeghian, Mazdak Zamani, Suhaimi Ibrahim, SQL Injection is Still Alive: A Study on SQL Injection Signature EvasionTechniques, 2013 International Conference on Informatics and Creative Multimedia, 2013.

[9] OWASP: SQL Injection Prevention Cheat Sheet, https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet [17.11.2017]

[10] Chandershekhar Sharma, S.C. Jain, Analysis and Classification of SQL Injection Vulnerabilities and Attacks on Web Applications, Konferencja: International Conference on Advances in Engineering & Technology Research, ICAETR – 2014.

[11] William G.J. Halfond, Jeremy Viegas, Alessandro Orso, A Classification of SQL Injection Attacks and Countermeasures, Proceedings of the International Symposium on Secure Software Engineering, 2006.

[12] How to: Protect From SQL Injection in ASP.Net, https://msdn.microsoft.com/en-us/library/ff648339.aspx [16.11.2017]

[13] Microsoft Sercurity Overview, https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/security-overview [22.11.2017]

[14] Tiobe Index for November 2017, https://www.tiobe.com/tiobe-index/ [23.11.2017]

[15] M. Dymek, M. Nycz, A. Gerka, Analiza statycznych metod obrony przed atakami SQL, ZESZYTY NAUKOWE POLITECHNIKI RZESZOWSKIEJ 294, Elektrotechnika 35 RUTJEE, z. 35 (2/2016), kwiecień-czerwiec 2016, s. 47-56.