Programowanie Defensywne

Podatność SQL Injection

Arkadiusz Chrobot

Katedra Systemów Informatycznych

14 marca 2024

Plan

1. SQL Injection
2. Wyrażenia regularne
3. Podsumowanie

SQL Injection

Scenariusz

Wyobraźmy sobie programistę pracującego w [*tutaj proszę wstawić nazwę swojej ulubionej korporacji*]. Jest on na „ASAPie”, bo to piątek przez 16:00, czyli „deadline” i trzeba wydać wersję dla klienta. Okazuje się, że w aplikacji brakuje możliwości zarejestrowania i uwierzytelniania użytkowników i trzeba dodać mechanizm, który je zapewni. Nasz bohater wie, że hasła w bazie danych powinny być przechowywane w postaci skrótów (ang. *hash*), a nie jawnym tekstem. Wybiera zatem najlepszą z obecnie znanych funkcji skrótu (Argon2). Niestety implementując komponent do uwierzytelniania i rejestracji popełnia kilka bardzo groźnych błędów. Przyjrzyjmy się zatem jego rozwiązaniu.

SQL Injection

Problem

Ponieważ nasz programista ma mało czasu, żeby dokładnie zapoznać się z działaniem metod implementujących algorytm Argon2, to postanawia „uprościć” jego użycie, tak aby to samo hasło wprowadzone przez użytkownika dawało zawsze ten sam skrót, który będzie zapisywany w bazie danych. W tym celu ustawia długość soli (ang. *salt*) na 0, a liczbę iteracji umożliwiających rozciągnięcie klucza (ang. *key stretching*) na 1 zamiast zalecanych 2 (pierwszy i ostatni argument konstruktora):

```
private String hashPassword(String password) {
        Argon2PasswordEncoder encoder = new
 ↪  Argon2PasswordEncoder(0,64,1,15*1024,1);
        return encoder.encode(password);
}
```

SQL Injection

Problem

Tak wygenerowany skrót hasła mechanizm programisty zapisuje w bazie danych przy rejestrowaniu użytkownika, a przy uwierzytelnianiu będzie on wyszukiwany razem z loginem dzięki następującym zapytaniom:

```
@Override
public String getInsertQuery() {
    return "insert into USERS values('" + login + "','"
 ↪  + hashedPassword + "');";
}

@Override
public String getSelectQuery() {
    return "select * from USERS where name='"+login+"'
 ↪  and password='"+hashedPassword+"';";
}
```

SQL Injection

Konsekwencje

Po wdrożeniu aplikacji konta założyły trzy osoby: Alicja, która starannie dobrała długie hasło, Bogdan, który użył stosunkowo krótkiego hasła, ale zawierającego „niestandardowe” znaki i Cyryl, który wybrał proste, krótkie hasło. Niestety po miesięcznej przerwie w używaniu aplikacji Alicja nie jest w stanie przypomnieć sobie hasła, a twórcy oprogramowania zapomnieli o możliwości jego resetowania. Ponieważ jednak ta użytkowniczka jest inteligentną osobą i zna się trochę na bezpieczeństwie, to próbuje zalogować się podając następujący login: `alicja'--`. Okazuje się, że zostaje uwierzytelniona niezależnie od tego jakie hasło wprowadzi! Co więcej, po pewnym czasie odkrywa, że przy rejestracji może dodać nowego użytkownika, bez określonego hasła, a nawet kilku takich użytkowników. Wystarczy, że jako login wprowadzi np.:

```
dorota','x'); insert into USERS values('anna','y');--
```

SQL Injection

Konsekwencje

To, że te użytkowniczki nie będą miały prawidłowych haseł nie jest przeszkodą — mogą się logować podobnie jak Alicja np.: `anna'--`. Niestety to nie koniec problemów. W czasie, kiedy Alicja odkryła sposób na „uwierzytelnienie” bez podawania hasła, Edward przekonał się, że może zalogować się do aplikacji, bez wcześniejszej rejestracji. Wystarczy tylko, aby baza danych nie była pusta. Wówczas login `edward' or 1=1--` gwarantuje mu dostęp do aplikacji.

W opisywanym oprogramowaniu jest zapewne wiele innych defektów, ale te, które powodują wspomniane zachowanie są dwa. Programista źle zastosował funkcję skrótu Argon2. Zalecanym jest, aby jednym z jej argumentów była sól o wielkości co najmniej 32 bajtów oraz aby funkcja wykonała co najmniej 2 iteracje celem rozciągnięcia klucza. Podczas uwierzytelniania nie porównuje się skrótu wygenerowanego z hasła podanego przez użytkownika ze skrótem zapisany w bazie danych bezpośrednio, tylko za pomocą metody *matches()* obiektu enkodera, do której przekazuje się wspomniane hasło.

SQL Injection

Przyczyny

Więcej informacji na temat implementacji tego algorytmu w Javie można znaleźć w artykule „How to do password hashing in Java applications the right way!”.

W ramach tego wykładu zajmiemy się jednak drugą z podatności, czyli wstrzyknięciem SQL (ang. *SQL Injection*). Jest to jedna z wielu podatności polegających na wstrzyknięciu obcego kodu do aplikacji, który ona następnie wykona. W tym konkretnym przypadku chodzi o wykonanie poleceń w relacyjnej bazie danych. Przyjrzyjmy się, co stanie się, jeśli Alicja poda jako swój login `alicja'--`. Okazuje się, że zapytanie `select` z metody `getSelectQuery()` przybierze mniej więcej następującą postać:

```
select * from USERS where name='alicja'-- and …
```

Znak `'` zakończy kwerendę, a wszystko co znajduje się za znakami `--` jest przez interpreter SQL traktowane jako komentarz, czyli zostanie zignorowane — dlatego nie podaję w całości reszty polecenia.

SQL Injection

Przyczyny

Ponieważ aplikacja sprawdza jedynie, czy zapytanie zwróciło jakikolwiek niepusty wynik, to uzna, że użytkownik z takim loginem jest wiarygodny.
Login Edwarda (`edward' or 1=1--` powoduje wygenerowanie następującej kwerendy:

```
select * from USERS where name='edward' or 1=1 -- and …
```

Warunek `1=1` jest zawsze prawdziwy, więc zapytanie zwróci wszystkie rekordy z bazy danych, więc użytkownik również zostanie „rozpoznany” przez aplikację, mimo, że jego danych nie ma nawet w bazie.
Podanie loginu:

```
dorota','x'); insert into USERS values('anna','y');--
```

przy rejestracji generuje dwa zapytania:

```
insert into USERS values ('dorota','x');
```

oraz

```
insert into USERS values('anna','y');--
```

SQL Injection

Warianty

Ten ostatni wariant wstrzyknięcia nazywa się w języku angielski „*stacked queries*”, co na język polski można przetłumaczyć jako „nałożone zapytania”. Jest on możliwy ponieważ polecenia SQL `insert` w aplikacji są wykonywane przez metodę `execute()` klasy `Statement`, która dopuszcza wykonanie kilku poleceń, w przeciwieństwie do metody `executeQuery()`, która stosowana jest w przypadku `select`. Ominięcie uwierzytelniania nie jest jedynym celem ataku ze wstrzyknięciem SQL. Częściej jest ono stosowane do wydobycia utajnionych danych (ang. *exfiltration*)[1]. Jeśli np. aplikacja internetowa bezpośrednio wyświetla dane zwracane przez bazę, to można spróbować wariantu wstrzyknięcia SQL ze słowem kluczowym `UNION`, które pozwala połączyć wyniki dwóch zapytań `select`. Problemem są nazwy i ewentualnie typy kolumn dla drugiego zapytania.

[1] W polskiej terminologii informatycznej spotykane jest słowo „eksfiltracja”, jednakże nie jestem pewien, czy jest językowo poprawne.

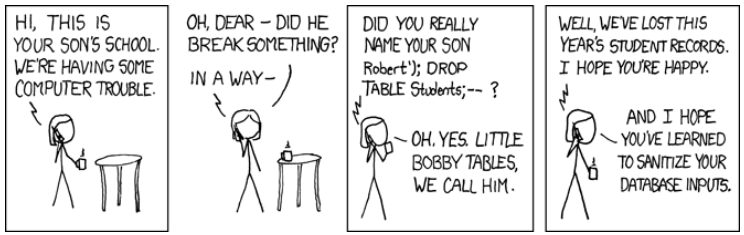
SQL Injection

Warianty

Te jednak można ustalić podając w drugim zapytaniu, w ich miejscu odpowiednią liczbę wartości `null` lub eksperymentując z pierwszym zapytaniem przez dodawanie klauzuli `ORDER BY`.
Inny wariant wykorzystania tej podatności polega na doprowadzeniu do błędu (ang. ERROR-*based*), o którym komunikat aplikacja wyświetli na ekranie. Może on zawierać interesujące dane.
Jeśli wcześniej opisane warianty nie zadziałają można użyć podejścia ślepego (ang. BLIND) bazującego na zawartości (ang. *content based*) lub czasie (ang. *time based*). W pierwszym przypadku wstrzykuje się do zapytania warunki logiczne i sprawdza się wartości przez nie generowane. W drugim dodaje się do zapytania elementy, które powodują spowolnienie jego wykonania, jeśli określony warunek jest lub nie jest spełniony.

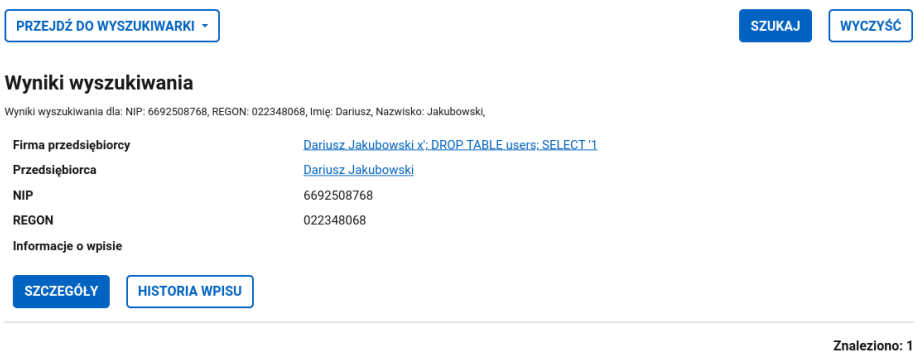
SQL Injection

Cele ataku mogą być różne, podobnie jak ich skutki:



Autor: Randall Munroe, źródło: ▸ XKCD

Nadanie nietypowego imienia nie jest proste, ▸ nawet w USA, ale problem może dotyczyć nie tylko imienia i pojawić się nie tylko w Stanach Zjednoczonych:



Źródło: CEIDG (https://ceidg.gov.pl)

SQL Injection

Zapobieganie

Podstawową ochroną w przypadku wszystkich typów wstrzyknięć jest **walidacja niezaufanych informacji**. Są to dane wejściowe pochodzące od użytkownika. W przypadku aplikacji internetowych walidacja powinna być wykonywana po stronie zaufanej, czyli serwera. Może też występować po stronie klienta, ale nie może być jedyną opcją i nie można ufać jej wynikom. Operacja ta może mieć różny charakter. Może w niej chodzić o usunięcie niebezpiecznych znaków lub całych fraz na zasadzie akceptacji poprawnych (ang. *white list*) lub odrzuceniu niepoprawnych (ang. *black list*). Jej celem może być także zamiana niebezpiecznych znaków na bezpieczne (ang. *escaping*). Wszystko zależy od kontekstu w którym jest ona wykonywana.

SQL Injection

Zapobieganie

W przypadku SQL Injection warto zapoznać się z zaleceniami OWASP w tym względzie. Do najczęściej stosowanych środków ochrony należą między innymi:

1. użycie parametryzowanych zapytań,
2. zastosowanie odpowiednio skonstruowanych procedur składowanych,
3. walidacja typów danych,
4. zastosowanie rozwiązań ORM.

Dodatkowo ta ochrona jest uzupełniana odpowiednią konfiguracją bazy danych (ang. *database hardening*). Przyjrzymy się dwóm z tych metod ochrony, czyli parametryzowanym zapytaniom i walidacji danych.

SQL Injection

Zapobieganie

Kod odpowiedzialny za operacje na bazie danych można w aplikacji odpowiednio zmienić, np.:

```
String selectStatement = "select * from USERS where
 ↪  name=? and password=?";
try(PreparedStatement statement =
 ↪  connection.prepareStatement(selectStatement)) {
        statement.setString(1,login);
        statement.setString(2,hashedPassword);
        ResultSet result =  statement.executeQuery();
        …
```

Zapytania parametryzowane (ang. *prepared statements*) automatycznie zamieniają niebezpieczne znaki na bezpieczne sekwencje (ang. *escape sequence*), które mogą być składowane w bazie danych i nie powodują wstrzyknięcia kodu. Ten sposób jest najbardziej uniwersalny.

SQL Injection

Zapobieganie

Z kolei walidację typów danych można przeprowadzić za pomocą *wyrażeń regularnych*. W aplikacji można np. zdefiniować metodę, która w loginie odrzuca wszystkie znaki, nie będące literami lub cyframi:

```
private String sanitizeLogin(String login) {
        Pattern patter =
↪   Pattern.compile("[^A-Za-z\\d]");
        return patter.matcher(login).replaceAll("");
}
```

Zastosowanie wyrażeń regularnych zostanie omówione pod koniec wykładu. W tym miejscu warto jednak zaznaczyć, że to rozwiązanie musi być uzupełnione dodatkowymi elementami. Przede wszystkim użytkownik powinien być informowany przez oprogramowanie o ograniczonym zbiorze znaków, jaki może wykorzystać w loginie. Nie jest to także metoda, którą można zastosować w przypadku wszystkich danych (patrz przykład z firmą).

Wyrażenia regularne

Wyrażenia regularne są specjalnym językiem do wyszukiwania wzorców w tekście. Dzięki temu nadają się one do zaawansowanego przetwarzania tekstu. Osoby zainteresowane ich teoretycznymi podstawami, w szczególności związkami z językami formalnymi i automatami NFA oraz DFA powinny sięgnąć do pozycji [1]. Stosunkowo krótkie, ale ciekawe wprowadzenie do teorii automatów DFA pojawiło się na kanale ▸ Computerphile. Opis wyrażeń regularnych niezależny od języka programowania, w którym są stosowane znajduje się również na stronie [2]. Ich zastosowanie w języku Java opisują z kolei pozycje [3] i [4].

Wyrażenia regularne

Wyrażenia regularne są omawiane w ramach tego wykładu z racji ich stosowania w walidacji danych. Należy jednak pamiętać o tym, że jeśli mechanizm ich realizacji bazuje na automatach NFA, to stosuje algorytm z nawrotami, które mają dużą złożoność obliczeniową i umożliwiają intruzom przeprowadzenie ataku typu ReDoS. W związku z tym wyrażenia regularne z rozwiązania mogą stać się one problemem, zgodnie ze osławionym zdaniem, którego autorem jest Jamie Zawinski: „Some people, when confronted with a problem, think «I know, I'll use regular expressions.» Now they have two problems.". Strona ▸ https://regexr.com/ zawiera aplikację, która pozwala debugować wyrażenia regularne. Z wyrażeń regularnych korzysta między innymi Spring Security. Proste wprowadzenie do ich konstruowania stanowi strona ▸ Regular Expressions For Regular Folks.

Wyrażenia regularne

Wprowadzenie do implementacji w Javie

Wyrażenia regularne mogą być argumentami takich metod klasy `String`, jak `split()` (podział łańcucha na części), `matches()` (sprawdzenie, czy cały łańcuch odpowiada wzorcowi, `replaceFirst()` oraz `replaceAll()` (odpowiednio: zastępowanie pierwszego wystąpienia wzorca i wszystkich wystąpień).
Jednakże najwięcej możliwości przy stosowaniu wyrażeń regularnych dają klasy `Pattern` i `Matcher` z pakietu `java.util.regex`. Pierwsza z nich posiada statyczną metodę `compile()`, która jako argument przyjmuje wyrażenie i na jego podstawie tworzy automat, który będzie je rozpoznawał, reprezentowany przez obiekt tej klasy. Z kolei metoda `matcher()` przyjmuje jako argument tekst, w którym należy sprawdzić pod kątem występowania wzorca i zwraca obiekt klasy `Matcher`. Ten z kolei posiada szereg metod wyszukujących wzorzec. Część z nich jest przedstawiona na następnym slajdzie.

Wyrażenia regularne

Wprowadzenie do implementacji w Javie

Metody `matches()`, `lookingAt()` i `find()` zwracają wartość typu `boolean`. Pierwsza z nich sprawdza, czy cały tekst pasuje do wzorca. Druga sprawdza, czy wzorzec znajduje się na początku tekstu, a trzecia, czy występuje gdziekolwiek. Istnieje również przeciążona wersja metody `find()`, która przyjmuje argument określający pozycję znaków w tekście, od którego należy zacząć sprawdzanie dopasowania. Po ich wywołaniu można użyć innych metod, które umożliwiają uzyskanie informacji na temat odnalezionego dopasowania. Przykładowo, metody `start()` i `end()` zwracają pozycję w tekście znaku początkowego i końcowego dopasowania. Jeśli we wzorcu użyto grup, to metoda `groupCount()` zwróci liczbę ich dopasowań, a metoda `group()`, wywołana z numerem grupy jako argumentem, zwróci dopasowanie związane z tą grupą. Uwaga: wersja przeciążona tej metody, bez argumentu, zwraca grupę główną, czyli cały dopasowany wzorzec. Ponadto grupa główna nie jest uwzględniana przez metodę `groupCount()`.

Wyrażenia regularne

Znaki

Następujące znaki można zastosować we wzorcu:

A	Konkretna litera, w tym przypadku A
\xhh	Znak o wartość szesnastkowej 0xhh
\uhhhh	Znak Unicode reprezentowany przez liczbę szesnastkową 0xhhhh
\t	Znak tabulacji
\n	Znak nowego wiersza
\r	Znak początku wiersza
\f	Znak wysunięcia strony
\e	Znak escape

Wyrażenia regularne

Klasy znaków

Poniższe klasy znaków stosuje się jako wzorce lub we wzorcach, aby zlokalizować interesujące fragmenty tekstu:

.	Dowolny znak
[abc]	Dowolny ze znaków a, b i c
[^abc]	Dowolny ze znaków, oprócz a, b i c
[a-zA-Z]	Dowolna litera
\s	Białe znaki (np. spacje)
\S	Dowolny znak, oprócz białych (np. spacji)
\d	Cyfra
\D	Dowolny znak, z wyjątkiem cyfry
\w	Dowolny ze znaków tworzących słowa
\W	Dowolny ze znaków nietworzących słowa

Uwaga: w języku Java \ jest znakiem specjalnym, więc trzeba go poprzedzić kolejnym takim znakiem, np. \\d.

Wyrażenia regularne

Operatory logiczne

We wzorcach można stosować następujące operatory logiczne:

XY	Znak X, a po nim Y.
X|Y	Znak X lub Y.
(X)	Grupa, do której można się odwołać w dalszej części wyrażenia.

Wyrażenia regularne

Znaki kotwiczące

Znaki kotwiczące w wyrażeniu pozwalają uszczegółowić, jaka część tekstu powinna być sprawdzona, czy pasuje do wzorca.

$\widehat{}$	Początek wiersza
\$	Koniec wiersza
\b	Granica słowa
\B	Negacja granicy słowa
\G	Koniec poprzedniego dopasowania

Wyrażenia regularne

Kwantyfikatory

Kwantyfikatory określają ile stojących przed nimi znaków lub klas znaków może wystąpić w określonym miejscu we wzorcu.

Zachłanny	**Leniwy**	**Dzierżawczy**	**Opis**
X?	X??	X?+	Wyrażenie X może wystąpić raz, lub nie wystąpić
X*	X*?	X*+	Wyrażenie X nie wystąpi lub wystąpić dowolną liczbę razy
X+	X+?	X++	Wyrażenie X wystąpi co najmniej jeden raz
X{n}	X{n}?	X{n}+	Wyrażenie X wystąpi dokładnie *n* razy
X{n, }	X{n, }?	X{n, }+	Wyrażenie X wystąpi co najmniej *n* razy
X{n,m}	X{n,m}?	X{n,m}+	Wyrażenie X wystąpi co najmniej *n* i co najwyżej *m* razy

Wyrażenia regularne

Kwantyfikatory

Dopasowanie wzorców, w których użyto kwantyfikatorów *zachłannych*, domyślnie stosowanych, dokonywane jest algorytmem *z nawrotami*, którego wykonanie może być kosztowne i wykorzystane do przeprowadzenia ataku typu DoS (ang. *Denial of Service*). Algorytm *leniwy* jest w tym względzie bezpieczniejszy, ale działa na innej zasadzie niż ten z nawrotami — dopasowuje najmniejszą liczbę znaków, zamiast sprawdzać wszystkie możliwości dopasowania. Algorytm *dzierżawczy* jest również stosunkowo bezpieczny, bo nie zachowuje stanów pośrednich starając się znaleźć dopasowanie, ale jest trudniejszy w użyciu.

Wyrażenia regularne

Przykład

```
package pl.kielce.tu;

import java.util.regex.*;

public class RegularExpressionsDemo {
    public static void main(String[] args) {
        String text = "a.chrobot@tu.kielce.pl";
        //RFC 5322
        Matcher matcher =
↪   Pattern.compile("^([a-zA-Z0-9_!#$%&'*+/=?`{|}~^.-
↪   ]+)(@)([a-zA-Z0-9.-]+)$").matcher(text);
        System.out.println(matcher.find());
        System.out.println(matcher.groupCount());
        for(int i=0;i<matcher.groupCount()+1;i++)
            System.out.println(matcher.group(i));

    }
}
```

Wyrażenia regularne

Program z poprzedniego slajdu używa wyrażenia regularnego skonstruowanego na podstawie dokumentu ▸ RFC 5322, które służy do walidacji prawidłowości adresu e-mail[2]. W pierwotnej postaci to wyrażenie miało postać:

```
^[a-zA-Z0-9_!#$%&'*+/=?`{|}~^.-]+@[a-zA-Z0-9.-]+$
```

Zostały jednak dodane do niego trzy grupy, aby móc wyodrębnić część lokalną adresu, znak i domenę. Te operacje wykonywane są w pętli `while`. Proszę zwrócić uwagę, że wypisywana jest również grupa główna.

OWASP udostępnia ▸ repozytorium wyrażeń regularnych do walidacji danych wejściowych. Warto również zapoznać się z ▸ artykułem dotyczącym ataku DoS z wykorzystaniem wyrażeń regularnych (ReDoS).

[2]Wyrażenie to nie jest w pełni kompatybilne z tym dokumentem, bo nie uwzględnia pewnych przypadków. Z drugiej strony są w użyciu adresy e-mail, które w ogóle nie są zgodne ze standardem.


John E. Hopcroft i Jeffrey D. Ullman. *Wprowadzenie do teorii automatów, języków i obliczeń*. Warszawa: Wydawnictwa Naukowe PWN, 1994.


Bruce Eckel. *Thinking in Java*. Gliwice: Helion, 2006.


Regular-Expressions.info. 2023. URL: https://www.regular-expressions.info/.


Regular Expressions in Java. 2023. URL: https://www.vogella.com/tutorials/JavaRegularExpressions/article.html.

Pytania

?

Koniec

Dziękuję Państwu za uwagę!