

Computerforensik

Netzwerkforensik: Erkennung von SQL-Injections

Master Informatik

Stefan Braun

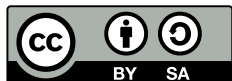
Matrikel 42482

Prüfer *Prof. Dr. Stefan Hahndel*

15. Juli 2016



Technische Hochschule
Ingolstadt



Dieses Material steht unter der Creative-Commons-Lizenz Namensnennung – Weitergabe unter gleichen Bedingungen 4.0 International. Um eine Kopie dieser Lizenz zu sehen, besuchen Sie <http://creativecommons.org/licenses/by-sa/4.0/>.

INHALTSVERZEICHNIS

1	SEITE 5	SQL-Injections im Jahr 2016
1.1	Verhinderung von SQL-Injections	5
1.2	Alternativen	6

2	SEITE 8	Arten von SQL-Injections
2.1	Tautologie-basierte Injections	8
2.2	UNION based Injection	9
2.3	Statement Injection	9
2.4	Error based SQL Injection	9
2.5	Time based SQL injection	10
2.6	SQL-Injection zweiten Grades	11

3	SEITE 12	Versuchsbeschreibung
3.1	Grundlegende Überlegungen	12
3.2	Verwendete Werkzeuge	12
3.3	Automatisierung der SQL-Injections mit sqlmap	12
3.4	Verwundbare Beispielapplikation	14
3.5	Apache und ModSecurity	15
3.6	MySQL Enterprise: Datenbank und Firewall	15
3.7	Versuchsablauf	16
3.8	Verwendete Versionen und Alternativen	17

4	SEITE 19	Ergebnisse
----------	----------	------------

ABBILDUNGSVERZEICHNIS

1.1	Webserverarchitektur	6
1.2	Webserverarchitektur mit WAF und Datenbankfirewall	7
3.1	Ablaufdiagramm der MySQL Enterprise Datenbankfirewall	17

QUELLCODEVERZEICHNIS

1.1	Dynamische Auswahl von Spalten und Sortierung	6
2.1	Verwundbare Anmeldungslogik	8
2.2	Auszug aus einem verwundbaren Wordpress-Plugin	10
2.3	Ausgabe einer Error-based Injection	10
2.4	Payload einer Time-based Injection	10
3.1	Logdatei von ModSecurity	16
3.2	Errorlog von MySQL	16

SQL-INJECTIONS IM JAHR 2016

ALLE DREI JAHRE veröffentlicht das *Open Web Application Security Project* – kurz OWASP – eine Liste der derzeit als am kritischsten eingestuften Sicherheitsrisiken in Webapplikationen. Und auch in der derzeit aktuellsten Fassung der Liste aus dem Jahr 2013 findet sich die Kategorie „Injections“ auf Platz Eins wieder.

Kategorie	
1	Injection
2	Broken Authentication
3	Cross-Site-Scripting

Derartige Angriffe basieren darauf, dass Benutzereingaben ungeprüft in Abfragen an LDAP-Dienste und vor allem SQL-Datenbanken als Parameter eingefügt werden. Entsprechend geformte Eingaben können somit die grundlegende Struktur der Anfrage manipulieren. Diese Manipulation kann Verlust der Informationsvertraulichkeit oder der Datenintegrität zur Folge haben, unter Umständen kann ein Angreifer Vollzugriff auf die zugrundeliegende Serverstruktur erhalten. Die vorliegende Arbeit konzentriert sich hierbei insbesondere auf gefährdete SQL-Anfragen.

1.1 Verhinderung von SQL-Injections

Es stellt sich folglich die Frage, wie derartige Angriffe verhindert werden können. Die übliche Vorgehensweise stellt hierbei die Überprüfung der vom Client übergebenen Parameter dar. Etwa könnte unter PHP ein Parameter, für den nur Ganzzahlen vorgesehen sind, per Konvertierung durch `intval()` abgesichert werden. Bei beliebigen Zeichenketten escaped die Funktion `mysql_real_escape_string()` bestimmte Zeichen, die einen Ausbruch aus der Abfrage erlauben können. Sicherer sind allerdings sogenannte *Prepared Statements*, die die Anfrage und die zugehörigen Parameter getrennt voneinander übertragen und dadurch Injections verhindern.

Wenn also die Verhinderung von SQL-Injections eine triviale Angelegenheit ist, weshalb bestimmen auch heutzutage Nachrichten über aktuelle, derartige Angriffe die Fachpresse? Die Gründe hierfür sind vielfältig. Möglicherweise

1

Derzeit werden Daten für die kommende OWASP Top Ten 2016 gesammelt.

Tabelle 1.1: Die ersten drei Kategorien der aktuellen OWASP Top Ten aus dem Jahr 2013, nach www.owasp.org

„Don't trust user input.“

Diese PHP-Funktion ersetzt beispielsweise ' durch \'. Dadurch wird es erschwert, das aktuelle String-Literal im SQL-Statement zu beenden und zusätzliche Befehle anzufügen.

ist veraltete Software im Einsatz oder dem Entwickler mangelt es schlicht an Vorwissen im Bereich der IT-Sicherheit. Ebenfalls vorstellbar ist Software, die nicht mehr geändert werden kann – etwa weil der Aufwand zu groß wäre, keine Entwickler vorhanden sind, oder aber der zugehörige Sourcecode nicht zur Verfügung steht. Außerdem können Queries konstruiert werden, die ein fachliches Problem zwar auf einfache Weise lösen, andererseits jedoch die Verwendung eines parametrisierten Prepared Statements unmöglich machen.

Ein weiteres Beispiel könnte zugekaufte Fremdsoftware darstellen, die im eigenen Netzwerk betrieben wird.

```

1 $query = ""
2     . "SELECT"
3     . "    $chosenText AS myText,"
4     . "    name"
5     . "FROM"
6     . "    report"
7     . "ORDER BY"
8     . "    name $sorting"
9     . ";";
mysqli_query($connection, $query);

```

Listing 1.1: In diesem PHP-Code wird mit der Variable `chosenText` eine Spalte und mit `sorting` eine Sortierreihenfolge ausgewählt. In beiden Fällen können keine Parameter für Prepared Statements verwendet werden.

1.2 Alternativen

In all diesen Fällen muss die gefährdete Applikation also auf andere Art und Weise abgesichert werden. Ein gängiger Ansatz zur Realisierung einer solchen Schutzmaßnahme stellt eine vorgeschaltete Softwarekomponente dar, welche auf Basis von Filterregeln einzelne Requests verwirft oder modifiziert. Hierzu soll zuerst ein übliches Schema einer Client-Server-Architektur skizziert werden.

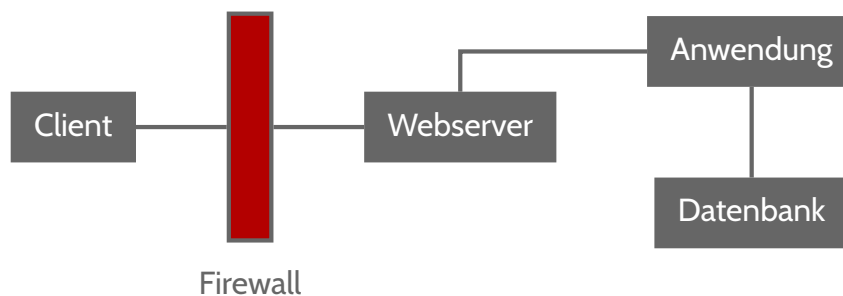


Abbildung 1.1: Zugriffe auf eine Webserverapplikation passieren üblicherweise zuerst eine Paketfilter-firewall und werden anschließend von einem Webserver – etwa *Apache* – zur Applikation weitergeleitet. Diese Applikation kann anschließend auf den Datenbankserver zugreifen.

In aktuellen Webserverarchitekturen können weitere Komponenten enthalten sein, die an dieser Stelle jedoch vernachlässigt und abstrahiert werden sollen.

In dem abstrakten Schema eines Requests aus Abbildung 1.1 bieten sich zwei Stellen an, an welchen einzelne Parameter der Anfrage auf ihre Gefährlichkeit in Bezug auf SQL-Injections hin untersucht werden können. Analysiert man die beispielsweise die GET und POST Parameter eines Requests noch bevor

Ein Beispiel hierfür stellen etwa *Load Balancer* zur Lastverteilung auf mehrere Server dar.

sie beim Webserver ankommt, spricht man von einer *Web Application Firewall*. Alternativ können auch die Zugriffe auf den Datenbankserver selbst untersucht werden – und zwar von einer vorgeschalteten *Datenbank-Firewall*. Es stellt sich die Frage, inwiefern die beiden Ansätze in Bezug auf ihre Effektivität miteinander vergleichbar sind – und ob sie einen wirksamen Schutz vor SQL-Injections bieten können.

Sowohl Web Application Firewall als auch Datenbank-Firewall stellen *Intrusion Detection* Systeme dar.

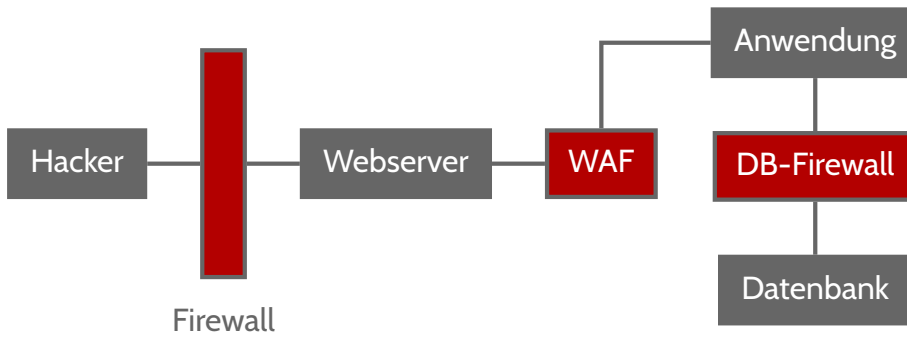


Abbildung 1.2: In die Abbildung 1.1 wurden an den entsprechenden Stellen Schutzmechanismen eingefügt. Möglichkeiten hierfür sind eine *Web-Application Firewall* – kurz WAF – und eine *Datenbank-Firewall*. Es stellt sich die Frage, wie effektiv die jeweiligen Maßnahmen SQL-Injections mitigieren können.

ARTEN VON SQL-INJECTIONS

SQL-INJECTIONS LASSEN SICH nach unterschiedlichen Kriterien klassifizieren. Dies geschieht in Hinblick auf die Art und Weise wie ein Angreifer die verwundbare SQL-Anfrage entdeckt, wie er den Schadcode einfügt und schließlich Daten auslesen kann. In diesem Kapitel soll ein grober Überblick über verschiedene Ansätze von SQL-Injections vermittelt werden. Diese Auflistung ist keineswegs erschöpfend, stellt aber häufig auftretende Varianten dar.

2

2.1 Tautologie-basierte Injections

Die einfachste Variante einer SQL-Injection sorgt dafür, dass eine logische Überprüfung im **WHERE**-Teil der Abfrage immer zu true evaluiert und somit alle betroffenen Zeilen zurückgegeben werden. Das Paradebeispiel hierfür ist eine Abfrage zur Authentisierung eines Benutzers.

```
1 $query = ""
2 . "SELECT"
3 . "    username"
4 . "FROM"
5 . "    users"
6 . "WHERE"
7 . "    username = '$username'"
8 . "AND"
9 . "    password = '$password'";
10
11 $result = mysqli_query($connection, $query);
12
13 if(mysqli_num_rows($result) != 0){
14     $_SESSION['user_logged_in'] = $username;
15 }
```

Listing 2.1: Eine einfache Anmelde-logik: Wird in der Datenbank ein Nutzer mit dem übergebenen Nutzernamen und Passwort gefunden, wird eine Sessionvariable gesetzt. Die Variablen username und password sind üblicherweise POST-Parameter, die an dieser Stelle ungeprüft in das SQL-Statement eingebunden werden. Stattdessen sollten besser Prepared Statements verwendet werden, die auch in PHP verfügbar sind.^a

^a<http://php.net/manual/de/mysqli.quickstart.prepared-statements.php>

Wenn ein Angreifer die Logindaten jeweils auf ' OR ' = ' setzt, wird jeweils überprüft, ob ein Leerstring identisch zu einem Leerstring ist. Da dieser Vergleich immer true ergibt, werden alle in der Tabelle users enthaltenen Zeilen zurückgegeben. Der Benutzer wird eingeloggt.

2.2 UNION based Injection

Etwas komplexer als die im vorherigen Abschnitt vorgestellte Methode sind Anfragen, welche die Menge der zurückgegebenen Zeilen eines **SELECT** Statements durch ein angefügtes **UNION** erweitern. Wichtig ist hierbei, dass die Anzahl der Spalten der mit **UNION** angefügten Query identisch mit der Spaltenanzahl der ursprünglichen **SELECT** Abfrage sein muss. Führt der restliche Teil der originalen Abfrage zu Problemen, kann er gegebenenfalls auskommentiert werden. Hierzu muss an das Ende des injizierten **UNION** Teils per `--` oder `#` ein Kommentar eingeleitet werden.

2.3 Statement Injection

Statement Injection oder auch Stacked Queries funktioniert ähnlich wie die **UNION** basierte Variante, jedoch wird hier eine komplette, zusätzliche SQL-Abfrage eingefügt. Hierzu wird zuerst die aktuelle Anfrage valide vervollständigt und anschließend per Semikolon beendet. Nun kann ein eigenständiges SQL-Statement angehängt werden – beispielsweise **DROP DATABASE** wordpress;. Folgender SQL-Code kann wie im vorherigen Abschnitt erläutert einfach auskommentiert werden.

Damit derartige Angriffe unter PHP funktionieren, muss für die Anfrage statt der Funktion `mysqli_query()` die Variante `mysqli_multi_query()` verwendet werden. Andernfalls ist die Verwendung konkatenierter SQL-Statements nicht möglich, denn nur die zweite Variante lässt es zu, mehrere Anfragen in einem Schritt durchzuführen.

2.4 Error based SQL Injection

Wenn eine Anfrage zwar anfällig für SQL-Injections ist, die zugehörige Webseite allerdings keine Daten direkt ausgibt, können möglicherweise dennoch Daten ausgelesen werden. In manchen Webanwendungen wird bei einem Fehler in einer Datenbankabfrage die entsprechende Fehlermeldung ausgegeben. Kann diese Ausgabe durch die Injection provoziert und geändert werden, so ist eine *Error based SQL Injection* möglich. Eine hierfür verwundbare Codestelle zeigt Listing 2.2.

Übergibt man als id etwa

```
1 EXTRACTVALUE(1,(SELECT CONCAT(0x0a, USERNAME())))
```

so wird der Inhalt von Listing 2.3 ausgegeben. Die Funktion `EXTRACTVALUE` erwartet als zweiten Parameter eine gültige XPath-Anweisung. Der Benutzername, der per `USERNAME()` in einer Subquery ausgelesen wird, ist kein gültiges

Angemerkt: In einem xkcd Webcomic unter <https://xkcd.com/327/> löscht eine Mutter Daten der Schule: Sie hatte ihren Sohn „Robert“); `DROP TABLE Students; --` getauft – ein klassisches Beispiel für eine Statement Injection.

Ein reales Beispiel hierfür findet sich in den handgeschriebenen Stimmzetteldaten^a einer Wahl in Schweden: Diese enthielten ebenfalls `DROP` Statements. Der Injection-Versuch blieb allerdings folgenlos, es wurden keine Daten verändert.

^averöffentlicht unter <http://www.val.se/val/val2010/handskrivna/handskrivna.skv>

```

1 $sql = "update `".DC_MV_CAL."` set"
2   . " `exdate`='" . esc_sql($exdate) . "' "
3   . "where `id`='" . $id;
4
5 if ($wpdb->query($sql)=== FALSE){
6     $ret['IsSuccess'] = false;
7     $ret['Msg'] = $wpdb->last_error;
8 }

```

Argument, da ein nicht druckbares Zeichen vorangestellt wurde – und wird daher in der Fehlermeldung mit ausgegeben.

```

1 {
2     "IsSuccess":false,
3     "Msg":"XPath syntax error: '@localhost'"
4 }

```

Listing 2.2: Ein Auszug aus dem Wordpress-Plugin cp multi view calendar^a. Die Variable id wird nicht überprüft und ermöglicht so SQL-Injections. Im Fehlerfall wird die Meldung in Zeile 7 in eine lokale Variable geschrieben und später ausgegeben.

^a<https://github.com/wp-plugins/cp-multi-view-calendar>

Listing 2.3: Ausgabe der *Error based* SQL-Injection. In der Fehlermeldung ist das Resultat der Subquery zu sehen, in diesem Fall der Rückgabewert der Funktion USERNAME().

2.5 Time based SQL injection

Ändert sich an der Ausgabe der Seite trotz erfolgreich ausgeführter SQL-Injection nichts, so ist es dennoch möglich, Daten auszulesen. Hierfür werden SQL Funktionen wie SLEEP() oder BENCHMARK() verwendet, die die Ausführungsdauer einer Query erhöhen können. Verbindet man dies mit einer IF() Bedingung und misst die Dauer des gesamten Requests, so erlaubt dies erneut Rückschlüsse auf Datenbankinhalte. Zudem stellt SLEEP() auch eine probate Möglichkeit dar, einen verwundbaren Parameter zu entdecken.

```

1 (SELECT
2   IF(
3     SUBSTRING(user.Password,1,1) = CHAR(65)
4     ,SLEEP(5)
5     ,2
6   )
7 FROM mysql.user
8 LIMIT 1)

```

Listing 2.4: Diese Query vergleicht ein einzelnes Zeichen einer Zeichenkette mit einem bestimmten ASCII-Code. Liefert der Vergleich true, so wird fünf Sekunden gewartet.

Ein Beispiel für eine Time-based SQL-Injection könnte die Subquery aus Listing 2.4 darstellen. Hierbei wird überprüft ob ein Zeichen des Passworts

gleich dem ASCII Zeichen A ist. Ist dies der Fall, so wird eine Pause eingelegt, was wiederum die Antwort des Requests verzögert. Diese Verzögerung kann auf Clientseite gemessen werden, wodurch ein Rückschluss auf die IF Anfrage der Query gezogen werden kann.

2.6 SQL-Injection zweiten Grades

Auch in Bezug auf die Art und Weise, wie der zusätzliche Code in die SQL-Anweisung gelangt, gibt es Unterscheidungsmöglichkeiten. Eine Variante hiervon sind SQL-Injection *zweiten Grades*. Hierbei wird der Payload zuerst als valider String in der Datenbank abgespeichert. Im späteren Verlauf lädt die Applikation den String zuerst aus der Datenbank und fügt ihn wiederum in eine neue SQL-Anfrage ein – welche an dieser Stelle für SQL-Injections anfällig ist. Während sich ein Entwickler bei empfangenen GET- und POST-Parametern oft im Klaren über die Möglichkeit einer Injection-Attacke ist, ist dieses Bewusstsein bei Daten aus der Datenbank möglicherweise nicht in der gleichen Höhe ausgeprägt. Dies kann dazu führen, dass bei Daten aus der Datenbank gängige Sicherheitsmaßnahmen vernachlässigt werden.

VERSUCHSBESCHREIBUNG

FOLGENDE ABSCHNITTE SOLLEN aufzeigen, wie im Zuge dieser Seminararbeit die Tauglichkeit verschiedener automatisierter Angriffs- und Verteidigungsmechanismen rund um SQL-Injections getestet wurde.

3

3.1 Grundlegende Überlegungen

Wenn die Bewertung einer derartigen Abwehrmaßnahme zum Thema wird, kommen zwei einfache Metriken in Frage: die Anzahl der *false positives* und die der *true negatives*. Wie viele HTTP-Anfragen führen zwar zu SQL-Injections, werden jedoch nicht erkannt – und auf der anderen Seite: Wie viele Anfragen werden verworfen, obwohl sie eigentlich ungefährlich sind?

Je eines der Beiden zu 100% zu erfüllen ist einfach: *true negatives* werden verhindert, indem die Firewall alle Anfragen blockiert, *false positives* treten nicht auf, wenn keine einzige Überprüfung stattfindet oder die Abwehrmaßnahme komplett deaktiviert wird. Da beide Extreme üblicherweise unerwünscht sind, gilt es ein passendes Mittelmaß zu finden. Es sei zudem darauf hingewiesen, dass die Nichterkennung einer SQL-Injection größeren Schaden verursachen kann als das versehentliche Verwerfen einer normalen Anfrage.

Um ein Beispiel zu geben: Einem harmlosen Benutzer, der sich bei der Registrierung "DROP DATABASE" nennen möchte, eine Fehlermeldung zu präsentieren, ist der Wirtschaftlichkeit der Website meist weniger abträglich als das Löschen der kompletten Datenbank.

3.2 Verwendete Werkzeuge

Bezugnehmend auf die Abbildung 1.2 wird jeweils eine Web Application Firewall und eine Database Firewall getestet. Damit diese Tests möglich sind, muss zum einen die grundlegende Infrastruktur aufgebaut werden: Die Basis bildet ein Apache 2.4 Webserver auf einem Linux-Serversystem. Damit Angriffe möglich sind, muss eine für SQL-Injections verwundbare Applikation eingerichtet werden. Mögliche Schwachstellen werden schließlich automatisiert mit dem Tool *sqlmap* gesucht und getestet.

Zum Einsatz kommt ein 64-bit Ubuntu Server. Siehe auch <http://www.ubuntu.com/download/server>

3.3 Automatisierung der SQL-Injections mit sqlmap

Zum Auffinden verwundbarer Request-Parameter wurde das Python basierte Werkzeug *sqlmap* ausgewählt. Es bietet verschiedene Kommandozeilenoptionen an, deren Verwendung an dieser Stelle kurz erläutert wird. Das frei verfügbare Tool kann von <http://sqlmap.org/> aus installiert werden.

Mit dem Parameter `-u` wird die anzugreifende URL angegeben. Bereits dieses Argument genügt für eine erste Analyse: Standardmäßig wird ein GET-Request verwendet und dabei versucht, alle in der URL enthaltenen Parameter

Vergleiche hierzu <https://github.com/sqlmapproject/sqlmap/wiki/Usage>

```
1 > python sqlmap.py -u "http://www.example.com/app.php?id=1"
```

anzugreifen. Hierbei setzt sqlmap verschiedene mögliche Payloads für die angegebenen Parameter ein und überprüft anhand der Serverantwort, ob die SQL-Injection erfolgreich war.

```
1 > python sqlmap.py -u "http://www.example.com/app.php"
  ↳ --data={username=&password=}
```

sqlmap ist ebenfalls in der Lage, POST-Requests durchzuführen. Hierzu werden die Parameter mit dem Argument `--data` angegeben.

Ebenfalls wichtig ist die Angabe des UserAgents, den sqlmap für die Anfrage verwenden soll. Wird dieser mit dem Parameter `--user-agent` nicht spezifiziert, wird sqlmap/1.0.6.33#dev verwendet. Ein derartiger UserAgent Header kann jedoch dazu führen, dass eine Anfrage von einer WAF abgelehnt wird ohne auch nur die einzelnen Parameter zu betrachten.

```
1 sqlmap identified the following injection point(s) with a
  ↳ total of 307 HTTP(s) requests:
2 ---
3 Parameter: username (POST)
4 Type: boolean-based blind
5 Title: OR boolean-based blind - WHERE or HAVING clause (MySQL
  ↳ comment) (NOT)
6 Payload: uname=") OR NOT 7407=7407#&passwd=
7
8 Type: AND/OR time-based blind
9 Title: MySQL >= 5.0.12 OR time-based blind (comment)
10 Payload: uname=") OR SLEEP(5)#&passwd=
```

Auch die weiteren Filterregeln von ModSecurity – siehe Abschnitt 3.5 – testen, ob der UserAgent den String sqlmap enthält. Tests könnten somit verfälscht werden, da die von sqlmap gestellten Anfragen gar nicht von den SQL-Injections betreffenden Filterregeln geblockt werden.

Findet sqlmap verwundbare Parameter, so werden diese zusammen mit dem genutzten Payload ausgegeben. Im obigen Beispiel war der Parameter `username` angreifbar, sqlmap liefert hierzu eine Time-based und eine Boolean-based SQL-Injection.

```
1 > python sqlmap.py --level=5 -u
  ↳ "http://www.example.com/app.php"
  ↳ --data={username=&password=}
```

Gelingt es sqlmap nicht, eine SQL-Injection zu finden, so kann mit `--level` die Anzahl der getesteten SQL-Injection-Varianten erhöht werden – wobei 1 die niedrigste Stufe und 5 jene Stufe mit der größten Zahl an Anfragen darstellt. Neben der Anzahl der möglichen Varianten pro Parameter werden mit höheren Stufen auch Cookies und HTTP-Header getestet. Die größere Anzahl an Anfragen hat allerdings auch eine längere Ausführungsdauer des Befehls zur Folge.

```
1 > python sqlmap.py -o --dbms=MySQL -u
   ↪ "http://www.example.com/app.php?id"
```

Gerade hierfür kann die Verwendung der Argumente `-o` und `--dbms` sinnvoll sein. Mit `-o` werden allgemeine optimierende Optionen aktiviert – etwa Multithreading – welche die Ausführungsdauer reduzieren können. Ist das verwendete Datenbankmanagementsystem bekannt, so kann auch mit der Angabe von `--dbms` verhindert werden, dass SQL-Injections, die für andere DBMS spezifischen Code enthalten, durchgeführt wird.

Damit bei einem erneuten Suchlauf mit der selben URL Zeit gespart werden kann, legt sqlmap Sessionfiles an. Diese Dateien könnten im forensischen Sinne auch Hilfe bei der Aufklärung eines Angriffes durch sqlmap bieten. Unter Linux werden standardmäßig im Verzeichnis `/.sqlmap/output/<target>/` drei Dateien angelegt. Eine davon ist eine `session.sqlite` Datenbank, die Base64 kodierte Metainformationen enthält, die beiden anderen sind Textdateien. Die Datei `target.txt` enthält das `-u` Argument – also die angegriffene URL und die Datei `log` enthält die Ausgabe von sqlmap mit gefundenen Schwachstellen, Anzahl der durchgeführten Requests und Informationen über den angegriffenen Server.

3.4 Verwundbare Beispielapplikation

Um die Möglichkeiten sowohl von sqlmap als auch der Firewall-Anwendungen auszuloten zu können, wird eine verwundbare Beispielapplikation benötigt. Verwendet wird an dieser Stelle *SQLi-labs*. Diese beinhaltet 53 einfache Formulare auf Basis von PHP 5 und ist als Lernplattform für SQL-Injections gedacht. Die einzelnen Formulare greifen auf GET-Parameter wie `id` oder `sort` zu, auch Loginfelder mit Benutzername und Passwort sind gegeben.

Von diesen 53 Tests konnte sqlmap bei 32 mögliche SQL-Injections finden, jeweils zwischen einer und vier verschiedenen. Diese summieren sich zu 90 verschiedenen GET- und POST-Anfragen auf, die die Grundlage für den Versuch darstellen. Die gefundenen SQL-Injections lassen sich verschiedenen Kategorien zu ordnen, unter anderem Time-based und Error-based Injections waren vertreten.

Alternativ kann auch mit `--threads` die Anzahl der zu verwendenden Threads explizit angegeben werden. Der Parameter `-o` setzt standardmäßig drei Threads ein.

Was wenn der Computer eines Hackers beschlagnahmt wurde, der verdächtigt wird, mit sqlmap automatisiert nach Schwachstellen auf Firmenwebsites gesucht zu haben? In diesem Fall könnten die sqlmap Logdateien weiterhelfen.

Vergleiche <https://github.com/Audi-1/sqli-labs>

3.5 Apache und ModSecurity

Die eingesetzte WAF-Lösung *ModSecurity* ist ein Webserverplugin, welches für nginx, IIS und Apache verfügbar ist. Es basiert darauf, eingehende Serveranfragen anhand von Filterregeln auf potentielle Angriffe hin zu untersuchen. Matchen eine oder mehrere dieser Regeln, so kann die Anfrage abgelehnt oder geloggt werden. Da immer die Möglichkeit besteht, dass ModSecurity legale Anfragen blockiert und somit den normalen Betrieb einer Webanwendung stört, ist es meist sinnvoll zu Beginn nur den Loggingmechanismus zu verwenden. Erst wenn über einen längeren Zeitraum keine false positives mehr entdeckt wurden, kann in Erwägung gezogen werden, den Abwehrmechanismus zu aktivieren.

Die Filterregeln sind nicht Bestandteil von ModSecurity und müssen zusätzlich installiert und aktiviert werden. Neben dem kostenpflichtigen Angebot der Firma Trustwave – des Herstellers von ModSecurity – können auch frei verfügbare Regelsätze des OWASP verwendet werden. Zusätzlich zur Filterung von SQL-Injections werden auch XSS-Attacken und Schwachstellenscanner abgewehrt. Nach Installation der Erweiterung wird diese per Kommandozeilenbefehl aktiviert.

```
1 > a2enmod security2
2 > service apache2 restart
```

Treffen installierte Regellisten auf eine Anfrage zu, so wird diese in der Logdatei `/var/log/apache2/modsec_audit.log` protokolliert. Ist ModSecurity entsprechend konfiguriert, werden potentiell gefährliche Anfragen nicht nur geloggt sondern auch abgelehnt. In diesem Fall gelangt die Anfrage gar nicht erst zur eigentlichen Applikation, der Aufrufer erhält einen HTTP-403 Fehlercode *Forbidden*.

3.6 MySQL Enterprise: Datenbank und Firewall

Als Datenbanksystem wird MySQL eingesetzt. In der kostenpflichtigen Enterprise-Variante bietet dieses DBMS zusätzlich eine Datenbankfirewall an. Wie diese Firewallmechanik funktioniert, wird im Ablaufdiagramm in Abbildung 3.1 dargestellt. Ist die Firewall entsprechend konfiguriert, so kann sie SQL-Anfragen, die nicht in der Whitelist enthalten sind, ablehnen. Diese Anfragen sind somit mutmaßliche SQL-Injections. In PHP erhält man in diesem Fall mit `mysql_error()` folgende Fehlermeldung:

```
1 Statement was blocked by Firewall
```

Entsprechend ist die Verhaltensweise die eines *Intrusion Detection System* oder eines *Intrusion Prevention System*.

Die Regelsätze von Trustwave^a können für jährliche 495 Dollar lizenziert werden und beinhalten die OWASP Regeln. OWASP stellt seine Regeln unter Github^b zur Verfügung. Die Beschreibung der Regelsätze lässt vermuten, dass in der kostenpflichtigen Variante keine zusätzlichen Filter bezüglich SQL-Injections enthalten sind. Die zusätzlichen Regeln beziehen sich eher auf Malware-Erkennung und DoS Detection^c.

^a<https://ssl.trustwave.com/web-application-firewall>

^b<https://github.com/SpiderLabs/owasp-modsecurity-crs>

^c<http://www.modsecurity.org/rules.html>

```

1 Message: Access denied with code 403 (phase 2). Operator GE
  ↳ matched 3 at TX:sql_select_statement_count.
2 [file "modsecurity_crs_41_sql_injection_attacks.conf"]
3 [line "108"] [id "981317"] [rev "2"] [msg "SQL SELECT
  ↳ Statement Anomaly Detection Alert"]
4 [data "Matched Data: Cache-control found within
  ↳ TX:sql_select_statement_count: 3"]
5 [maturity "8"] [accuracy "8"] [tag
  ↳ "OWASP_CRS/WEB_ATTACK/SQL_INJECTION"]
6 Action: Intercepted (phase 2)
7 Stopwatch: 1466544262572653 1265 (- - -)
8 Stopwatch2: 1466544262572653 1265; combined=692, p1=186,
  ↳ p2=461, p3=0, p4=0, p5=44, sr=55, sw=1, l=0, gc=0
9 Response-Body-Transformed: Dechunked
10 Producer: ModSecurity for Apache/2.9.0
  ↳ (http://www.modsecurity.org/); OWASP_CRS/2.2.9.
11 Server: Apache/2.4.18 (Ubuntu)
12 Engine-Mode: "ENABLED"

```

Listing 3.1: Auszug aus der Logdatei von ModSecurity. Die Anfrage wurde abgelehnt, weil ein GET-Parameter ein UNION ALL SELECT enthielt.

Im DETECTING Modus werden Queries, die nicht in der Whitelist enthalten sind, im Errorlog von MySQL eingetragen.

```

1 2016-06-23T14:42:17.079455Z 543 [Note] Plugin MYSQL_FIREWALL
  ↳ reported: 'SUSPICIOUS STATEMENT from 'root@%'. Reason: No
  ↳ match in whitelist.
2 Statement: SELECT * FROM `users` ORDER BY ? AND `SLEEP` (?) '

```

Listing 3.2: Eintrag im Errorlog von MySQL: Eine SQL-Anfrage wurde in das Log eingetragen, weil sie nicht in der Whitelist enthalten ist.

Queries, deren Ausführungsdauer eine bestimmte Zeit überschreiten werden ebenfalls in eine <host>_slow.log protokolliert. Dies kann bei Time-based SQL-Injections dazu führen, dass die erzeugten Queries in dieser Logdatei gespeichert werden. Die Protokollierung langsamer Queries ist jedoch eine Standardfunktion von MySQL und steht nicht in direktem Zusammenhang mit der Datenbankfirewall.

Die Grenze für diese Ausführungsdauer wird in der MySQL Variable `long_query_time` definiert – standardmäßig liegt sie bei zehn Sekunden.

3.7 Versuchsablauf

Schwachstellen, die mit sqlmap entdeckt wurden, werden als Request Objekte in einem Pythonskript zusammengefasst. Anschließend wird je eine der beiden Abwehrmaßnahmen aktiviert. Das Pythonskript führt die insgesamt 90 Requests der Reihe nach aus und überprüft ob die SQL-Injection erfolgreich abgefangen wurde. Abschließend wird eine Statistik der erfolgreichen und der abgewehrten

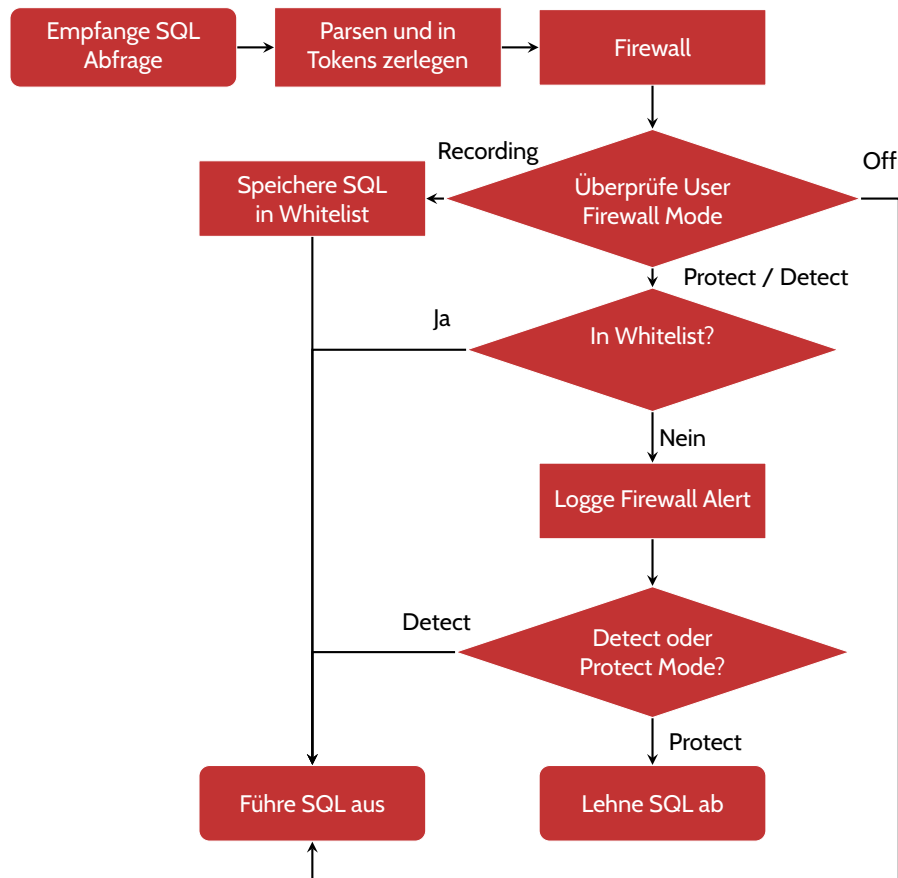


Abbildung 3.1: Diese Grafik skizziert die Funktionsweise der MySQL Datenbankfirewall.^a Eingehende SQL-Anfragen werden geparkt und in Tokens zerlegt. Pro Datenbankbenutzer existiert nun eine Whitelist mit zulässigen Abfragen: Ist der Lernmodus aktiv, so werden alle neuen Anfragen in die Whitelist mit aufgenommen. Wird die Firewall aktiviert, so werden neue Anfragen je nach Einstellung nur geloggt oder auch abgelehnt. Entsprechend kann die Datenbankfirewall wieder als IDS oder IDP konfiguriert werden.

^anach <http://dev.mysql.com/doc/refman/5.6/en/firewall.html>

SQL-Injections ausgegeben. Die Überprüfung erfolgt bei ModSecurity anhand des HTTP-Statuscodes. Die Tests mit der MySQL Enterprise Datenbankfirewall wurden so modifiziert, dass sie bei einer erkannten SQL-Injection eine entsprechende Warnung im HTML ausgeben.

3.8 Verwendete Versionen und Alternativen

Hier werden im Folgenden die verwendeten Programme und Anwendungen aufgelistet, die im Rahmen dieser Seminararbeit zum Einsatz kamen.

- ModSecurity 2.9.0
- OWASP Regelset (Stand 01.06.2016, github)
- SQLi-labs Anwendungen (Stand 01.06.2016, github)
- Apache 2.4.18
- PHP 5.6

- MySQL Enterprise 5.7.13, Trialversion
- SQLmap (1.0.6.33)

Neben ModSecurity und der integrierten Datenbankfirewall existieren auch noch alternative Lösungen, wie *Barracuda WAF* oder die Datenbankfirewall der Firma *HexaTier*. ModSecurity selbst kann auch für die Webserveralternativen IIS und nginx installiert werden. Auch für SQLmap existieren Alternativen, zu nennen ist hier *sqlninja* und *safe3 sql injector*. Anstatt der SQLi-labs könnten für die Tests auch selbstgeschriebene Anwendungen oder eventuell das *OWASP Mutillidae 2 Project* verwendet werden.

HexaTier ist zudem Hersteller der vormals als OpenSource vertriebenen Datenbankfirewall *GreenSQL*. Seit Anfang des Jahres 2016 ist GreenSQL nicht länger frei verfügbar, sondern wird als kommerzielle Software vertrieben.

ERGEBNISSE

4

DIE GUTE NACHRICHT vorweg: Sowohl ModSecurity als auch die Datenbankfirewall erkannten jeweils 90 von 90 Angriffen, was eine Erkennungsrate von 100% ergibt. Nicht getestet wurde auf *false positives*. Es ist jedoch davon auszugehen, dass die Datenbankfirewall konstruktionsbedingt keine Fehler dieser Art verursacht – vorausgesetzt, die Whitelist ist vollständig. Anders ist die Situation bei dem regelbasierten ModSecurity: Hier es durchaus möglich, dass Anfragen abgelehnt werden, die im Kontext zu einer spezifischen Query „harmlos“ sind. Auch war es bei ModSecurity möglich, Anfragen derart zu konstruieren, sodass ModSecurity diese *nicht* abgelehnt hat, aber die entstehende Query syntaktisch falsch war und somit einen serverseitigen Fehler erzeugt hat.

Die große Schwierigkeit bei der Datenbankfirewall ist wiederum die Einrichtung der Whitelist. Bei großen Applikationen mit dynamischen oder von Frameworks erstellten Datenbankabfragen steigt entsprechend der Aufwand, da alle legitimen Queries mindestens einmal aufgerufen werden müssen. Insbesondere ORM-Frameworks wie Hibernate sind an dieser Stelle zu nennen.

Da das Ergebnis aus Sicht eines Angreifers etwas unbefriedigend erscheinen mag, wurde abschließend getestet, ob sich die Filterregeln von ModSecurity umgehen lassen. Hierbei wurde mit verschiedenen Informationskanälen getestet, ob die Filterregeln von ModSecurity anschlagen. Hierzu wurde der String ' OR ' = ' ' in Parametern, Cookies und HTTP-Headern eingesetzt. Dies ergab, dass ModSecurity GET-Parameter, POST-Parameter und Cookies auf Muster von SQL-Injections hin untersucht. Nicht blockiert wurden Anfragen, die diesen String in den HTTP-Headern enthielten – etwa der UserAgent oder der Referrer. Inhalte von im Request enthaltenen Dateien werden ebenso wenig erkannt wie Base64-kodierte Parameter. An dieser Stelle wäre vermutlich eine Erweiterung der Filterregeln notwendig.

LITERATUR


- [CAo9] Justin Clarke und Rodrigo Marcos Alvarez, Hrsg. *SQL injection attacks and defense*. eng. Burlington, Mass.: Syngress, 2009. 473 S. ISBN: 978-1-59749-424-3. URL: <http://site.ebrary.com/lib/alltitles/docDetail.action?docID=10310739>.
- [DS16] Bernardo Damele und Miroslav Stampar. *sqlmap user manual*. 2016. URL: <https://github.com/sqlmapproject/sqlmap/wiki> (besucht am 19.06.2016).
- [Ora16] Oracle. *MySQL Enterprise Firewall*. 2016. URL: <https://www.mysql.com/products/enterprise/firewall.html> (besucht am 19.06.2016).
- [Tru16] Trustwave. *ModSecurity Documentation*. 2016. URL: <http://www.modsecurity.org/documentation.html> (besucht am 19.06.2016).

ERKLÄRUNG

ICH ERKLÄRE HIERMIT, dass ich die Arbeit selbständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen oder Hilfsmittel benützt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

nach „Muster für die Erklärung nach
§ 18 Abs. 4 Nr. 7 APO HI“

Ingolstadt, den 12. Juli 2016


Stefan Braun