9

SQL Injection

Wenn eine Schwachstelle bei einer Datenbank-basierten Site es dem Angreifer ermöglicht, die Datenbank der Site über *SQL (Structured Query Language)*, abzufragen oder anzugreifen, spricht man von *SQL-Injection (SQLi)*.

SQLi-Angriffe werden oft großzügig entlohnt, da sie verheerende Auswirkungen haben können: Angreifer können Informationen manipulieren oder extrahieren und möglicherweise sogar einen Administrator-Log-in für sich selbst anlegen.

9.1 SQL-Datenbanken

SLQ-Datenbanken speichern Informationen in Datensätzen (Records) und Feldern in einer Reihe von Tabellen. Tabellen enthalten eine oder mehr Spalten, und eine Zeile in der Tabelle repräsentiert einen Datensatz in der Datenbank.

Die Nutzer verwenden SQL, um Datensätze in einer Datenbank zu erzeugen, zu lesen, zu aktualisieren und zu löschen. Der Nutzer sendet SQL-Befehle (Anweisungen oder Queries) an die Datenbank, und – vorausgesetzt, die Befehle

werden akzeptiert – die Datenbank interpretiert die Anweisungen und führt entsprechende Aktionen durch. Beliebte SQL-Datenbanken sind MySQL, PostgreSQL, MSSQL und so weiter. In diesem Kapitel werden wir MySQL verwenden, doch die grundlegenden Konzepte gelten für alle SQL-Datenbanken.

SQL-Anweisungen bestehen aus Schlüsselwörtern und Funktionen. Die folgende Anweisung weist beispielsweise die Datenbank an, Informationen aus der Spalte `name` in der Tabelle `users` abzurufen, bei der die Spalte ID den Wert `1` hat.

```
SELECT name FROM users WHERE id = 1;
```

Viele Websites nutzen Datenbanken, um Informationen zu speichern, und verwenden diese Informationen dann, um dynamische Inhalte zu generieren. Hält die Site *https:// www.<example>.com/* beispielsweise Ihre bisherigen Bestellungen in einer Datenbank fest, auf die Sie zugreifen können, sobald Sie sich mit Ihrem Account angemeldet haben, dann würde die Datenbank der Site abgefragt und HTML basierend auf den zurückgelieferten Informationen generiert werden.

Hier ein theoretisches Beispiel für den PHP-Code eines Servers, der einen MySQL-Befehl erzeugt, nachdem der Nutzer die Seite *https://www.<example>.com?name=peter* besucht:

```
   $name = ❶$_GET['name'];
   $query = "SELECT * FROM users WHERE name = ❷'$name' ";
❸ mysql_query($query);
```

Der Code verwendet `$_GET[]` ❶, um auf den Namenswert aus den URL-Parametern zuzugreifen. Der gewünschte Wert wird zwischen den eckigen Klammern angegeben und in der Variablen `$name` gespeichert. Der Parameter wird dann ungeprüft an die Variable `$query` vübergeben. Die Variable `$query` enthält die auszuführende Query (also Abfrage), die alle Daten aus der Tabelle `users` abruft, bei denen der Wert der Spalte `name` mit dem Wert des URL-Parameters `name` übereinstimmt. Die Query wird ausgeführt, indem man die `$query`-Variable an die PHP-Funktion `mysql_query` ❸ übergibt.

Die Site erwartet, dass `name` normalen Text enthält. Doch wenn ein Nutzer die bösartige Eingabe `test' OR 1='1` im URL-Parameter übergibt, also `https://www.example.com?name=test' OR 1='1`, dann wird die folgende Query ausgeführt:

```
$query = "SELECT * FROM users WHERE name = 'test❶' OR 1='1❷' ";
```

Die bösartige Eingabe schließt das öffnende einfache Anführungszeichen (`'`) nach dem Wert `test` ❶ und hängt den SQL-Code `OR 1='1` an das Ende der Query an. Das hängende einfache Anführungszeichen in `OR 1='1` öffnet das schließende

einfache Anführungszeichen, das fest codiert hinter ❷ steht. Würde die eingeschleuste Query kein öffnendes einfaches Anführungszeichen enthalten, würde das hängende Anführungszeichen zu SQL-Syntaxfehlern führen, die eine Ausführung der Query verhindern würden.

SQL kennt die logischen Operatoren `AND` und `OR`. In diesem Fall modifiziert die SQLi die `WHERE`-Klausel so, dass nach allen Datensätzen gesucht wird, wo die Spalte `name` den Wert `test` enthält oder die Gleichung `1='1'` den Wert `true` zurückgibt. MySQL betrachtet `'1'` freundlicherweise als Integerwert, und da `1` immer gleich `1` ist, ist die Bedingung erfüllt (`true`), und die Query gibt alle Datensätze der `users`-Tabelle zurück. Das Einschleusen von `test' OR 1='1` würde allerdings nicht funktionieren, wenn andere Teile der Query gefiltert werden. So könnte zum Beispiel eine Query wie die folgende genutzt werden:

```
$name = $_GET['name'];
$password =❶mysql_real_escape_string($_GET['password']);
$query = "SELECT * FROM users WHERE name = '$name' AND password = '$password' ";
```

In diesem Fall wird der Parameter `password` ebenfalls vom Benutzer kontrolliert, diesmal aber sauber gefiltert ❶. Wenn Sie die gleiche Payload, `test' OR 1='1`, als Namen übergeben und Ihr Passwort 12345 angeben, würde die Anweisung wie folgt aussehen:

```
$query = "SELECT * FROM users WHERE name = 'test' OR 1='1' AND password = '12345' ";
```

Die Query sucht nach allen Datensätzen, bei denen der `name` `test` oder `1='1'` lautet und `password` den Wert `12345` hat (wir ignorieren die Tatsache, dass diese Datenbank Passwörter im Klartext speichert, was ebenfalls eine Schwachstelle darstellt). Da die Passwort-Prüfung einen `AND`-Operator nutzt, gibt diese Query nur Daten zurück, wenn das Passwort für den Datensatz `12345` lautet. Das hebelt zwar unseren SQLi-Versuch aus, hindert uns aber nicht daran, eine andere Angriffsmethode auszuprobieren.

Wir müssen den `password`-Parameter eliminieren, was wir mit `;--, test' OR 1='1;--` erreichen. Diese Injection erreicht zwei Dinge: Das Semikolon (`;`) beendet die SQL-Anweisung, und die beiden Striche (`--`) weisen die Datenbank an, den restlichen Text als Kommentar zu betrachten. Dieser eingeschleuste Parameter ändert die Query in `SELECT * FROM users WHERE name = 'test' OR 1='1';`. Der Code `AND password = '12345'` in der Anweisung wird zu einem Kommentar, und der Befehl liefert uns alle Datensätze der Tabelle zurück. Wenn Sie zwei Striche (--) als Kommentar verwenden, müssen Sie daran denken, dass MySQL ein Leerzeichen hinter den Strichen und dem restlichen Text verlangt. Anderenfalls gibt MySQL eine Fehlermeldung aus, ohne den Befehl auszuführen.

9.2 SQLi-Gegenmaßnahmen

Einen Schutz vor SQLi bieten sogenannte *Prepared Statements*. Dieses Datenbank-Feature führt sich wiederholende Queries aus. Die genauen Details würden den Rahmen dieses Buchs sprengen, doch sie schützen vor SQLi, weil Queries nicht länger dynamisch ausgeführt werden. Die Datenbank nutzt die Queries wie Templates und verwendet Platzhalter für Variablen. Selbst wenn ein Nutzer ungefiltert Daten an eine Query übergeben kann, verändert die Injection das Query-Template der Datenbank nicht, und SQLi wird verhindert.

Web-Frameworks wie Ruby on Rails, Django, Symphony und so weiter bieten fest integrierte Schutzmaßnahmen, um SQLi zu verhindern. Diese sind aber nicht perfekt und können Schwachstellen nicht überall verhindern. Die beiden einfachen SQLi-Beispiele, die Sie gerade gesehen haben, funktionieren bei mit solchen Frameworks entwickelten Sites üblicherweise nicht, es sei denn, die Entwickler folgen nicht den bewährten Praktiken oder erkennen nicht, dass die Schutzmaßnahmen nicht automatisch bereitgestellt werden. Die Site *https://rails-sqli.org/* pflegt zum Beispiel eine Liste gängiger SQLi-Muster in Rails, die daraus resultieren, dass Entwickler Fehler machen. Wenn Sie nach SQLi-Schwachstellen suchen, halten Sie am besten nach alten Websites Ausschau, die selbst entwickelt wurden oder Web-Frameworks und Content-Management-Systeme nutzen, die nicht alle Schutzmaßnahmen aktueller Systeme integrieren.

9.3 Blinde SQLi bei Yahoo! Sports

Schwierigkeitsgrad: Mittel
URL: *https://sports.yahoo.com*
Quelle: entfällt
Meldezeitpunkt: 16. Februar 2014
Gezahltes Bounty: $ 3.705

Eine *blinde SQLi* liegt vor, wenn Sie SQL-Anweisungen einschleusen können, aber keinen Zugriff auf die direkte Ausgabe der Query haben. Der Schlüssel zur Ausnutzung blinder Injections besteht darin, Informationen abzuleiten, indem man die Ergebnisse der nicht modifizierten Queries mit denen der modifizierten Queries vergleicht. Zum Beispiel fand Stefano Vettorazzi im Februar 2014 eine SQLi, als er die Yahoo!-Sports-Subdomain untersuchte. Die Seite nahm Parameter über den URL entgegen, fragte die Datenbank ab und gab eine Liste von NFL-Spielern basierend auf den Parametern zurück.

Vettorazzi änderte den URL

sports.yahoo.com/nfl/draft?year=2010&type=20&round=2

der die NFL-Spieler des Jahrs 2010 zurücklieferte, wie folgt ab:

sports.yahoo.com/nfl/draft?year=2010--&type=20&round=2

Vettorazzi hängte im zweiten URL zwei Striche (--) an den Parameter `year` an. Abbildung 9–1 zeigt, wie die Seite bei Yahoo! aussah, bevor Vettorazzi die beiden Striche einfügte. Abbildung 9–2 zeigt das Ergebnis, nachdem Vettorazzi die beiden Striche hinzugefügt hatte.

Die in Abbildung 9–1 zurückgegebenen Spieler waren andere als die in Abbildung 9–2. Wir können die eigentliche Query nicht sehen, da der Code im Backend der Site liegt. Doch sehr wahrscheinlich hat die ursprüngliche Query jeden URL-Parameter an eine SQL-Query übergeben, die in etwa so aussah:

```
SELECT * FROM players WHERE year = 2010 AND type = 20 AND round = 2;
```

Durch das Anhängen der beiden Striche an den `year`-Parameter machte Vettorazzi daraus die folgende Query:

```
SELECT * FROM PLAYERS WHERE year = 2010-- AND type = 20 AND round = 2;
```

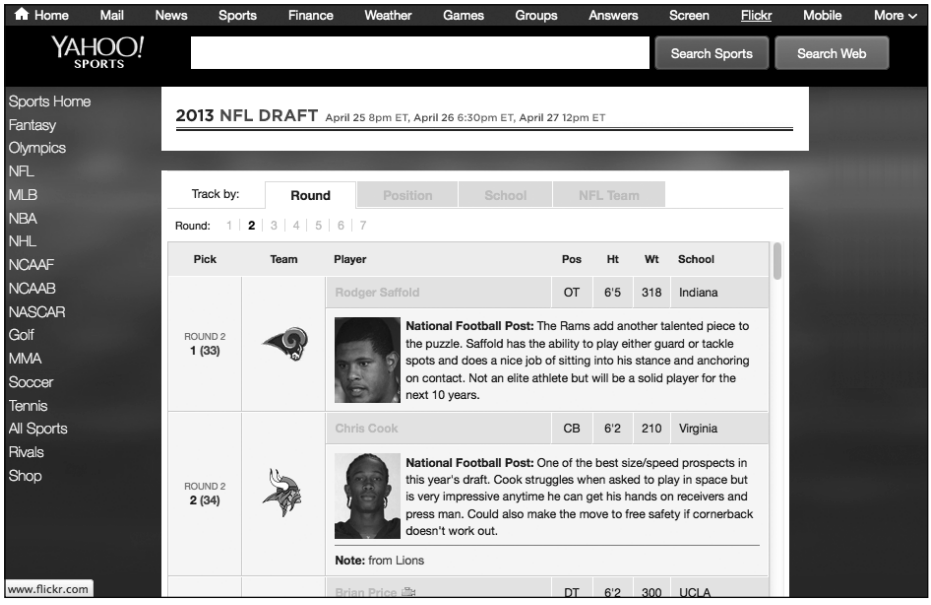

Abb. 9–1 *Ergebnisse der Yahoo!-Spielersuche mit nicht modifiziertem `year`-Parameter*

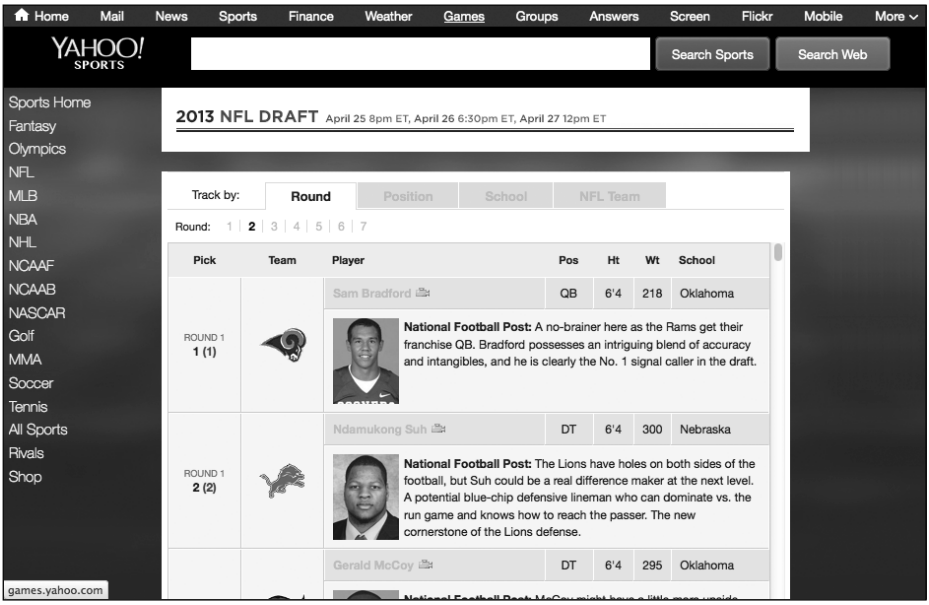

Abb. 9–2 *Ergebnisse der Yahoo!-Spielersuche mit modifiziertem `year`-Parameter*

Dieser Yahoo!-Bug ist etwas ungewöhnlich, da Queries bei den meisten, wenn nicht allen, Datenbanken mit einem Semikolon enden müssen. Da Vettorazzi nur zwei Striche eingeschleust und das Semikolon der Query damit auskommentiert hat, hätte diese Query fehlschlagen und einen Fehler oder keine Datensätze ausgeben müssen. Einige Datenbanken erlauben Queries ohne Semikola, also nutzte Yahoo! entweder diese Funktionalität, oder der Code bügelte den Fehler auf andere Weise aus. Warum auch immer es so war, nachdem Vettorazzi die unterschiedlichen Ergebnisse der Queries sah, versuchte er, die von der Site verwendete Datenbank-Version zu ermitteln, indem er den folgenden Code als `year`-Parameter übergab:

```
(2010)and(if(mid(version(),1,1))='5',true,false))--
```

Die `version()`-Funktion der MySQL-Datenbank gibt die aktuelle Version der verwendeten MySQL-Datenbank zurück. Die `mid`-Funktion liefert den durch das zweite und dritte Argument festgelegten Teil des im ersten Argument übergebenen Strings zurück. Das zweite Argument legt die Anfangsposition des zurückgelieferten Teilstrings fest und das dritte Argument die Länge. Vettorazzi überprüfte, ob die Site MySQL nutzte, indem er `version()` aufrief. Er versuchte dann, die erste Ziffer der Versionsnummer zu ermitteln, indem er der `mid`-Funktion eine `1` für die

Startposition und eine `1` für die Länge des Substrings übergab. Der Code überprüfte dann die erste Ziffer der MySQL-Version mithilfe einer `if`-Anweisung.

Die `if`-Anweisung verlangt drei Argumente: eine logische Prüfung, die durchzuführende Aktion, wenn die Prüfung erfolgreich ist (wahr, true), und die Aktion, die durchgeführt werden soll, wenn die Prüfung nicht erfolgreich ist (falsch, false). In diesem Fall prüft der Code, ob die erste Ziffer der `version` `5` ergibt. Ist das der Fall, gibt die Query `true` zurück, anderenfalls `false`.

Vettorazzi verband seine true/false-Ausgabe dann mit dem `year`-Parameter über den `and`-Operator. Wäre 5 die Hauptversion der MySQL-Datenbank, würden die Spieler des Jahres 2010 auf der Yahoo!-Webseite erscheinen. Das funktioniert, weil die Bedingung `2010 and true` immer `true` ergibt, während `2010 and false` immer `false` ist und keine Datensätze zurückliefert. Vettorazzi führte die Query aus und erhielt keine Datensätze zurück (siehe Abbildung 9–3), das heißt, die erste Ziffer des `version`-Werts war nicht `5`.

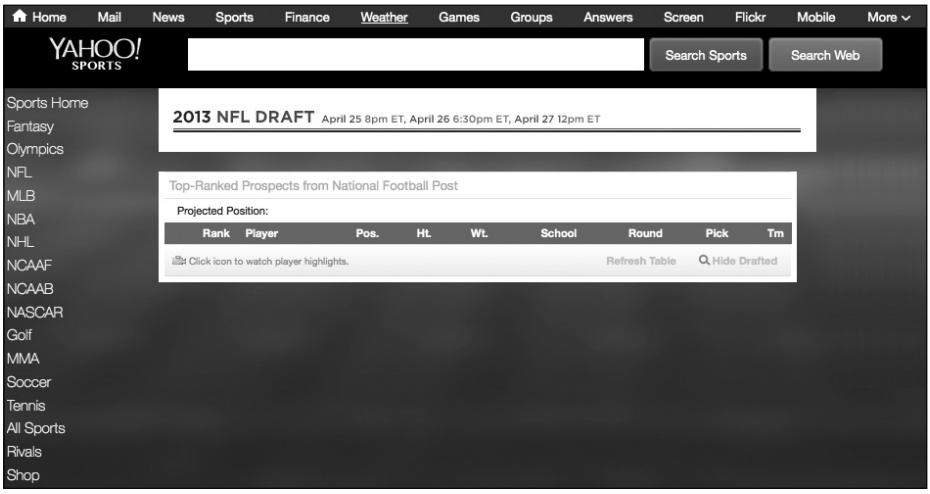

Abb. 9–3 *Das Ergebnis der Yahoo!-Spielersuche war leer, als der Code prüfte, ob die Datenbank-Version mit der Zahl 5 beginnt.*

Dieser Bug ist eine blinde SQLi, weil Vettorazzi seine Query nicht einschleusen und die Ausgabe nicht direkt auf der Seite sehen konnte. Doch Vettorazzi konnte dennoch Informationen über die Site sammeln. Durch Einschleusen boolescher Prüfungen wie der `if`-Anweisung zur Ermittlung der Version konnte Vettorazzi die von ihm benötigten Informationen ermitteln. Er hätte noch weitere Informationen aus der Yahoo!-Datenbank abrufen können. Doch die Ermittlung der MySQL-Version durch seine Test-Query reichte aus, um Yahoo! zu überzeugen, dass die Schwachstelle existierte.

9.3.1 Was wir mitnehmen

SQLi-Schwachstellen sind, wie andere Injection-Schwachstellen auch, oft relativ leicht auszunutzen. Eine Möglichkeit, SQLi-Schwachstellen aufzuspüren, besteht darin, die URL-Parameter zu untersuchen und nach subtilen Änderungen in den Query-Ergebnissen zu suchen. In diesem Fall veränderten die zwei eingefügten Striche die Ergebnisse von Vettorazzis Ausgangs-Query und offenbarten die SQLi-Schwachstelle.

9.4 Blinde SQLi bei Uber

Schwierigkeitsgrad: Mittel
URL: *http://sctrack.email.uber.com.cn/track/unsubscribe.do/*
Quelle: *https://hackerone.com/reports/150156/*
Meldezeitpunkt: 8. Juli 2016
Gezahltes Bounty: $ 4.000

Neben Webseiten finden Sie blinde SQLi-Schwachstellen auch an anderen Stellen, etwa in E-Mail-Links. Im Juli 2016 erhielt Orange Tsai eine Werbe-E-Mail von Uber. Er bemerkte, dass der Abbestell-Link einen base64-codierten String als URL-Parameter enthielt. Dieser Link sah wie folgt aus:

http://sctrack.email.uber.com.cn/track/unsubscribe.do?p=eyJ1c2VyX2lkIjogIjU3NTUiLCAicmVjZWl2ZXIiOiAib3JhbmdlQG15bWFpbCJ9

Die Decodierung des `p`-Parameters `eyJ1c2VyX2lkIjogIjU3NTUiLCAicmVjZWl2ZXIiOiAib3JhbmdlQG15bWFpbCJ9` mittels base64 ergab den JSON-String `{"user_id": "5755", "receiver": "orange@mymail"}`. In den decodierten String fügte Orange den Code `and sleep(12) = 1` ein. Dieser harmlose Zusatz sorgt dafür, dass die Datenbank länger braucht, um auf die Abmeldung von `{"user_id": "5755 and sleep(12)=1", "receiver": "orange@mymail"}` zu reagieren. Ist die Site anfällig, evaluiert die Query `sleep(12)` und macht 12 Sekunden lang nichts, bevor sie die Ausgabe des `sleep`-Befehls mit `1` vergleicht. Bei MySQL gibt der `sleep`-Befehl normalerweise 0 zurück, das heißt, der Vergleich schlägt fehl. Doch das spielt keine Rolle, weil die Ausführung mindestens 12 Sekunden benötigt.

Nachdem Orange die modifizierte Payload neu codiert hatte, übergab er sie als URL-Parameter im Abbestell-Link, um zu prüfen, ob die HTTP-Response mindestens 12 Sekunden brauchte. Da er konkretere Beweise für eine SQLi benötigte, bevor er sie an Uber meldete, ermittelte er per Brute-Force-Angriff den Nutzernamen, den Hostnamen und den Datenbanknamen. Auf diese Weise demons-

trierte er, dass er Informationen aus der SQLi-Schwachstelle extrahieren konnte, ohne auf vertrauliche Daten zuzugreifen.

Eine SQL-Funktion namens `user` gibt den Benutzer- und Hostnamen der Datenbank in der Form *<user>@<host>* zurück. Da Orange auf die Ausgabe seiner eingeschleusten Queries nicht zugreifen konnte, konnte er `user` nicht aufrufen. Stattdessen erweiterte Orange seine Query um eine konditionale Prüfung, wenn die Query seine Nutzer-ID abfragte. Dabei verglich er nacheinander jeweils ein Zeichen des Nutzer- und Hostnamens der Datenbank mithilfe der `mid`-Funktion. Ähnlich der blinden SQLi-Schwachstelle im obigen Beispiel nutzte Orange eine Vergleichsanweisung und einen Brute-Force-Angriff (also »rohe Gewalt«), um jedes Zeichen des Nutzer- und Hostnamens zu ermitteln.

So bestimmte Orange das erste Zeichen des von der `user`-Funktion zurückgegebenen Strings mithilfe der `mid`-Funktion. Dann verglich er das Zeichen nacheinander mit den Buchstaben `'a'`, `'b'`, `'c'` und so weiter. Gibt der Vergleich wahr zurück, führt der Server den Abbestell-Befehl aus. Das würde anzeigen, dass das erste von der `user`-Funktion zurückgelieferte Zeichen mit dem Zeichen identisch ist, mit dem es gerade verglichen wird. Ist die Anweisung nicht erfolgreich, würde der Server auch nicht versuchen, Orange abzumelden. Durch die Überprüfung jedes Zeichens des Rückgabewerts der `user`-Funktion konnte Orange schließlich den vollständigen Benutzer- und Hostnamen ermitteln.

Das manuelle Brute-Forcing eines Strings braucht Zeit, weshalb Orange das folgende Python-Skript entwickelte, das die notwendigen Payloads für ihn erzeugte und an Uber sendete:

```
❶ import json
  import string
  import requests
  from urllib
  import quote from base64
  import b64encode
❷ base = string.digits + string.letters + '_-@.'
❸ payload = {"user_id": 5755, "receiver": "blog.orange.tw"}
❹ for l in range(0, 30):
❺     for i in base:
❻         payload['user_id'] = "5755 and mid(user(),%d,1)='%c'#"%(l+1, i)
❼         new_payload = json.dumps(payload)
          new_payload = b64encode(new_payload)
          r = requests.get('http://sctrack.email.uber.com.cn/track/unsubscribe.
  do?p='+quote(new_payload))
❽         if len(r.content)>0:
                  print i,
                  break
```

Das Python-Skript beginnt mit fünf `import`-Anweisungen ❶, die die Bibliotheken einlesen, die Orange zur Verarbeitung von HTTP-Requests, JSON und String-Codierungen benötigt.

Nutzer- und Hostname einer Datenbank können aus einer beliebigen Kombination von Großbuchstaben, Kleinbuchstaben, Zahlen, Minuszeichen (–), Unterstrichen (_), at-Symbolen (`@`) oder Punkten (`.`) bestehen. Bei ❷ legt Orange die Variable `base` an, die diese Zeichen enthält. Der Code an ❸ erzeugt eine Variable, die die Payload enthält, die das Skript an den Server sendet. Die Codezeile an ❻ ist die Injection, die die `for`-Schleifen an ❹ und ❺ verwendet.

Sehen wir uns den Code an ❻ genauer an. Orange referenziert seine Benutzer-ID, 5755, über den String `user_id` an ❸, um seine Payload zu erzeugen. Er nutzt die `mid`-Funktion und die Stringverarbeitung, um eine Payload ähnlich dem vorhin beschriebenen Yahoo!-Bug zu erzeugen. Die `%d`- und `%c`-Zeichen in der Payload sind Platzhalter, die durch die entsprechenden Strings ersetzt werden. Das `%d` steht für eine Ziffer und das `%c` für Zeichen (Character).

Der Payload-String beginnt mit dem ersten Paar doppelter Anführungszeichen (`"`) und endet mit dem zweiten Paar doppelter Anführungszeichen vor dem dritten Prozentzeichen ❻. Das dritte Prozentzeichen weist Python an, die Platzhalter `%d` und `%c` durch die dem Prozentzeichen in Klammern folgenden Werte zu ersetzen. Der Code ersetzt also `%d` durch `l+1` (die Variable `l` plus `1`) und `%c` durch die Variable `i`. Das Hash-Zeichen (`#`) ist eine andere Möglichkeit, etwas in MySQL zu kommentieren, und verwandelt alles, was in der Query auf Oranges Injection folgt, in einen Kommentar.

Die Variablen `l` und `i` sind die Schleifen-Iteratoren an ❹ und ❺. Wenn der Code zum ersten Mal in die Schleife `l in range (0,30)` an ❹ eintritt, ist `l` gleich `0`. Der Wert von `l` ist die Position im Benutzername- und Hostname-String (die von der `user`-Funktion zurückgegeben werden); diesen Wert l will das Skript per Brute-Force herausfinden. Sobald das Skript eine Position innerhalb der zu testenden Benutzername- und Hostname-Strings hat, tritt der Code an ❺ in eine verschachtelte Schleife ein, die jedes Zeichen im `base`-String durchgeht. Beim ersten Durchlauf der beiden Schleifen ist `l` gleich `0` und `i` gleich `a`. Diese Werte werden an die `mid`-Funktion ❻ übergeben, um die Payload `"5755 and mid(user(),0,1)='a'#"`. zu erzeugen.

Bei der nächsten Iteration der verschachtelten `for`-Schleife hat `l` immer noch den Wert `0` und `i` den Wert `b`, erzeugt also die Payload `"5755 and mid(user(),0,1)='b'#"`. Die Position `l` bleibt gleich, während die Schleife jedes Zeichen in `base` durchgeht, um die Payload an ❻ zu erzeugen.

Bei jeder neu generierten Payload wandelt der auf ❼ folgende Code diese in JSON um, codiert den String mit der `base64encode`-Funktion und sendet den HTTP-Request an den Server. Der Code an ❽ prüft, ob der Server mit einer

Nachricht reagiert. Ist das Zeichen in `i` mit dem Benutzernamen an der untersuchten Position identisch, beendet das Skript die Zeichenprüfung an dieser Position und macht mit der nächsten Position im `user`-String weiter. Die verschachtelte Schleife wird beendet und kehrt zur Schleife an ❹ zurück, die `l` um `1` erhöht, um die nächste Position im Benutzernamen zu verarbeiten.

Mit diesem Machbarkeitsnachweis konnte Orange ermitteln, dass der Benutzername und der Hostname der Datenbank `sendcloud_w@10.9.79.210` lauteten und dass die Datenbank `sendcloud` hieß. (Um den Namen der Datenbank zu ermitteln, ersetzen Sie `user` durch `database` ❻). In der Reaktion auf den Report bestätigte Uber, dass die SQLi nicht auf den Servern des Unternehmens stattgefunden hat. Die Injection erfolgte auf einem von Uber genutzten Server eines Dienstleisters. Uber zahlte dennoch ein Bounty, auch wenn das nicht alle Programme so handhaben. Uber hat das Bounty wohl gezahlt, weil der Exploit es Angreifern erlaubte, alle E-Mail-Adressen der Uber-Kunden aus der `sendcloud`-Datenbank abzugreifen.

Auch wenn Sie wie Orange eigene Skripte entwickeln können, um Datenbank-Informationen aus einer verwundbaren Website abzugreifen, gibt es dafür auch automatisierte Tools. In Anhang A finden Sie Informationen zu einem dieser Tools namens sqlmap.

9.4.1 Was wir mitnehmen

Achten Sie auf HTTP-Requests, die codierte Parameter verarbeiten. Nach der Decodierung und dem Einschleusen einer eigenen Query in einen Request müssen Sie Ihre Payload wieder codieren, damit die Codierung dem entspricht, was der Server erwartet.

Das Extrahieren von Datenbank-, Benutzer- und Hostnamen ist generell harmlos, Sie müssen aber darauf achten, sich innerhalb der vom Bounty-Programm festgelegten Grenzen zu bewegen. Im manchen Fällen reicht ein `sleep`-Befehl als Machbarkeitsnachweis aus.

9.5 Drupal-SQLi

Schwierigkeitsgrad: Hoch
URL: Jede Drupal-Site mit Version7.32 oder älter
Quelle: *https://hackerone.com/reports/31756/*
Meldezeitpunkt: 17. Oktober 2014
Gezahltes Bounty: $ 3.000

Drupal ist ein beliebtes Open-Source-Content-Management-System zum Aufbau von Websites, ähnlich Joomla! und WordPress. Es ist in PHP geschrieben und *modular*, das heißt, Sie können eine Drupal-Site schrittweise um neue Funktionen ergänzen. Jede Drupal-Installation beinhaltet *Drupal Core*, einen Satz von Modulen für den Betrieb der Plattform. Diese Kernmodule benötigen eine Verbindung zu einer Datenbank wie MySQL.

2014 veröffentliche Drupal ein wichtiges Sicherheits-Update für Drupal Core, da alle Drupal-Sites von einer SQLi-Schwachstelle betroffen waren, die sehr einfach von anonymen Nutzern missbraucht werden konnte. Die Folge der Schwachstelle war, dass ein Angreifer jede ungepatchte Drupal-Site übernehmen konnte. Stefan Horst entdeckte die Schwachstelle, als er einen Bug in der Prepared-Statement-Funktionalität von Drupal Core bemerkte.

Die Drupal-Schwachstelle ist in Drupals Datenbank-API (Application Programming Interface) aufgetreten. Das Drupal-API verwendet die Erweiterung PHP Data Objects (PDO) als *Schnittstelle* für den Zugriff auf Datenbanken in PHP. Eine Schnittstelle (Interface) ist ein Programmierkonzept, das die Ein- und Ausgaben einer Funktion garantiert, ohne zu definieren, wie die Funktion implementiert wird. Mit anderen Worten versteckt PDO die Unterschiede zwischen den Datenbanken vor dem Programmierer, sodass er unabhängig von der verwendeten Datenbank die gleichen Funktionen zur Abfrage und für den Abruf von Daten verwenden kann.

Drupal hatte ein Datenbank-API entwickelt, das die PDO-Funktionalität nutzte. Das API erzeugte eine Abstraktionsschicht für die Drupal-Datenbank, sodass die Entwickler die Datenbank niemals direkt mit eigenem Code abfragen mussten. Doch sie konnte dennoch Prepared Statements nutzen und ihren Code mit jedem Datenbank-Typ verwenden. Die Details des APIs würden den Rahmen dieses Buchs sprengen, doch Sie müssen wissen, dass das API die SQL-Anweisungen zur Abfrage der Datenbank erzeugt und Sicherheitsprüfungen durchführt, um SQLi-Schwachstellen zu verhindern.

Erinnern Sie sich daran zurück, dass Prepared Statements SQLi-Schwachstellen verhindern, weil ein Angreifer die Struktur der Query nicht mit bösartigen Eingaben verändern kann, selbst wenn die Eingaben nicht gefiltert werden. Doch Prepared Statements können nicht vor SQLi-Schwachstellen schützen, wenn die Injection erfolgt, während das Template erzeugt wird. Kann ein Angreifer bösartigen Code einschleusen, während das Template erzeugt wird, kann er sein eigenes bösartiges Prepared Statement erzeugen. Die von Horst entdeckte Schwachstelle war in SQLs `IN`-Klausel begründet, die prüft, ob ein Wert in einer Liste von Werten existiert. Zum Beispiel wählt der Code `SELECT * FROM users WHERE name IN ('peter', 'paul', 'ringo');` die Daten aus der Tabelle `users` aus, bei denen die `name`-Spalte den Wert `peter`, `paul` oder `ringo` enthält.

Um zu verstehen, warum die `IN`-Klausel angreifbar ist, sehen wir uns den Code hinter Drupals API an:

```
$this->expandArguments($query, $args);
$stmt = $this->prepareQuery($query);
$stmt->execute($args, $options);
```

Die Funktion `expandArguments` ist für die Erzeugung der Queries verantwortlich, die die `IN`-Klausel verwenden. Nachdem `expandArguments` die Queries erzeugt hat, übergibt sie diese an `prepareQuery`, die die Prepared Statements erzeugt, die dann von der `execute`-Funktion ausgeführt werden. Um die Bedeutung dieses Prozesses zu verstehen, werfen wir auch einen Blick auf den relevanten Code für `expandArguments`:

```
   --schnipp--
❶ foreach(array_filter($args, `is_array`) as $key => $data) {
❷   $new_keys = array();
❸   foreach ($data as $i => $value) {
      --schnipp--
❹     $new_keys[$key . '_' . $i] = $value;
    }
    --schnipp--
  }
```

Dieser PHP-Code arbeitet mit Arrays. PHP kann assoziative Arrays verwenden, die explizit Schlüssel definieren:

```
['red' => 'apple', 'yellow' => 'banana']
```

Die Schlüssel in diesem Array sind `'red'` und `'yellow'`, und die Werte sind die Früchte rechts vom "Pfeil" (`=>`).

Alternativ kann PHP ein *strukturiertes Array* verwenden:

```
['apple', 'banana'].
```

Bei strukturierten Arrays sind die Schlüssel implizit und basieren auf der Position des Werts innerhalb der Liste. So ist der Schlüssel für `'apple'` hier `0`, und der Schlüssel für `'banana'` ist `1`.

Die PHP-Funktion `foreach` geht das Array durch und kann den Schlüssel von dessen Wert trennen. Sie kann jeden Schlüssel und jeden Wert auch einer eigenen Variablen zuweisen und diese dann an einen Codeblock zur Verarbeitung weitergeben. An ❶ nimmt `foreach` jedes Element eines Arrays und prüft, ob der übergebene Wert ein Array ist, indem er `array_filter($args, 'is_array')` aufruft. Sobald

die Anweisung bestätigt, dass es sich um einen Array-Wert handelt, weist sie bei der Iteration der `foreach`-Schleife jeden Schlüssel des Arrays an `$key` und jeden Wert an `$data` zu. Der Code modifiziert die Werte des Arrays, um Platzhalter zu erzeugen, das heißt, der Code an ❷ initialisiert ein neues, leeres Array, das später die Platzhalter-Werte enthält.

Um die Platzhalter zu erzeugen, geht der Code an ❸ das `$data`-Array durch und weist jeden Schlüssel an `$i` und jeden Wert `$value` zu. An ❹ wird dann das `new_keys`-Array (das an ❷ initialisiert wurde) mit dem Schlüssel des ersten Arrays und dem Schlüssel an ❸ verknüpft. Das angedachte Ergebnis dieses Codes sind Platzhalter der Form `name_0`, `name_1` und so weiter.

Eine typische Query zur Abfrage einer Datenbank mit Drupals `db_query`-Funktion sieht dann etwa so aus:

```
db_query("SELECT * FROM {users} WHERE name IN (:name)",
  array(':name'=>array('user1','user2')));
```

Die `db_query`-Funktion erwartet zwei Parameter: eine Query mit benannten Platzhaltern für Variablen und einem Array von Werten, um diese Platzhalter zu ersetzen. In diesem Beispiel also `:name` als Platzhalter und ein Array mit den Werten `'user1'` und `'user2'`. Bei einem strukturierten Array lautet der Schlüssel für `'user1'` `0` und für `'user2'` `1`. Führt Drupal die Funktion `db_query` aus, ruft es `expandArguments` auf, das die Schlüssel mit dem Wert verknüpft. Die resultierende Query verwendet `name_0` und `name_1` anstelle der Schlüssel, wie hier zu sehen:

```
SELECT * FROM users WHERE name IN (:name_0, :name_1)
```

Das Problem tritt nun ein, wenn Sie `db_query` mit einem assoziativen Array nutzen wie im folgenden Code:

```
db_query("SELECT * FROM {users} where name IN (:name)",
  array(':name'=>array('test);-- ' => 'user1', 'test' => 'user2')));
```

In diesem Fall ist `:name` ein Array mit den Schlüsseln `'test);--'` und `'test'`. Wenn `expandArguments` das `:name`-Array verarbeitet und die Query erzeugt, kommt Folgendes heraus:

```
SELECT * FROM users WHERE name IN (:name_test);-- , :name_test)
```

Wir haben einen Kommentar in das Prepared Statement eingeschleust. Der Grund ist, dass `expandArguments` alle Elemente des Arrays durchgeht, um Platzhalter zu erzeugen, aber davon ausgeht, dass ein strukturiertes Array vorliegt. Bei

der ersten Iteration wird `$i` der Wert `'test);--'` zugewiesen und `$value` der Wert `'user1'`. Der `$key` ist `':name'`, und die Kombination mit `$i` ergibt `name_test);--`. Bei der zweiten Iteration wird `$i` der Wert `'test'` zugewiesen und `$value` der Wert `'user2'`. Die Kombination von `$key` und `$i` ergibt den Wert `name_test`.

Dieses Verhalten erlaubt bösartigen Angreifern das Einschleusen von SQL-Anweisungen in Drupal-Queries, die die `IN`-Klausel nutzen. Die Schwachstelle wirkt sich auf Drupals Log-in-Funktion aus, was sie zu einer schwerwiegenden Schwachstelle macht, weil jeder Nutzer der Site, auch ein anonymer Nutzer, sie missbrauchen kann. Was die Dinge aber noch schlimmer machte, war die Tatsache, dass PHP PDO standardmäßig die Möglichkeit unterstützt, mehrere Queries auf einmal auszuführen. Das bedeutet, dass der Angreifer zusätzliche Queries an die Log-in-Query anhängen und SQL-Befehl ohne `IN`-Klausel ausführen konnte. Beispielsweise könnte ein Angreifer `INSERT`-Anweisungen verwenden, um Datensätze in die Datenbank einzufügen. So könnte er einen administrativen Nutzer anlegen, über den er sich dann auf der Website anmelden könnte.

9.5.1 Was wir mitnehmen

Bei dieser SQLi-Schwachstelle ging es nicht einfach darum, ein einfaches Anführungszeichen einzuschleusen und die Query zu knacken. Hier musste man vielmehr verstehen, wie das Datenbank-API von Drupal Core die `IN`-Klausel handhabt. Was wir bei dieser Schwachstelle mitnehmen, ist, dass man nach Gelegenheiten Ausschau halten muss, die Struktur der an eine Site übergebenen Eingaben zu verändern. Wenn ein URL den Parameter `name` verwendet, fügen Sie `[]` hinzu, um aus dem Parameter ein Array zu machen, und schauen, wie die Site damit umgeht.

9.6 Zusammenfassung

SQLi kann eine erhebliche Schwachstelle sein und eine Gefahr für eine Site darstellen. Findet ein Angreifer eine SQLi, könnte er die vollständigen Rechte über eine Site erlangen. In manchen Fällen kann die SQLi-Schwachstelle ausgeweitet werden (wie im Drupal-Beispiel), indem Daten in die Datenbank eingefügt werden, um an administrative Rechte für die Site zu gelangen. Wenn Sie nach SQLi-Schwachstellen suchen, achten Sie auf Orte, an denen Sie ungefiltert einfache oder doppelte Anführungszeichen an eine Query übergeben können. Wenn Sie eine Schwachstelle finden, können die Anzeichen für einen Erfolg, wie bei blinden Injections, sehr subtil sein. Sie sollten auch nach Stellen suchen, an denen Sie Daten in unerwarteter Form an die Site übergeben können, beispielsweise wenn Sie wie beim Uber-Bug Array-Parameter in den Request-Daten ersetzen können.