Web Security – Identifizieren und Beheben von Sicherheitslücken in Webanwendungen

Stefan Schurtz, Dr. Philipp Walter

Webanwendungen haben sich in den vergangenen Jahren als eigenständige Anwendungsform großflächig etabliert, daher stehen gerade diese Anwendungen, ständig unter Beobachtung, so dass hier kontinuierlich Sicherheitslücken gefunden und gestopft werden müssen

IN DIESEM ARTIKEL ERFAHREN SIE…

- Wie vorhandene Schwachstellen in Webanwendungen erkannt, identifiziert und behoben werden können

WAS SIE VORHER WISSEN SOLLTEN…

- Linux-, Apache-, MySQL, PHP-Kenntnisse
- Kenntnisse zum Thema Web-Application-Security

Einführung

Webanwendungen haben sich in den vergangenen Jahren als eigenständige Anwendungsform großflächig etabliert. Sie sind einerseits für Nutzer attraktiv, andrerseits aber auch einfach zu entwickeln, da es eine Vielzahl von Programmiersprachen, Frameworks und anpassbaren Open-Source-Standardlösungen gibt. Das führt allerdings häufig dazu, dass „selbstgestrickte" Lösungen gravierende Sicherheitsmängel aufweisen, da den Entwicklern nur wenige Ressourcen zur Verfügung stehen oder sie schlicht nicht über die notwendige Erfahrung verfügen. Aber auch große, etablierte Pakete, z. B. Content-Management-Systeme (CMS), Blog- oder Forensoftware, stehen gerade wegen ihrer großen Verbreitung ständig unter Beobachtung, so dass auch hier kontinuierlich Lücken gefunden und gestopft werden müssen. Oft stehen Entwickler – vor allem im professionellen Umfeld – unter Zeitdruck und haben meist nicht die Möglichkeit einer sauberen und sicheren Implementierung oder gar die Gelegenheit einer methodischen Sicherheitsüberprüfung. Ein weiteres Problem können zum Beispiel Plugins sein: immer wieder zeigt sich, dass eine Webanwendung durch den Einsatz einer unsicher programmierten Erweiterung auf einen Schlag kompromittiert werden kann. Typische Mängel sind z. B. Cross-Site-Scripting (XSS), Local *File Inclusion* (LFI), Remote *File Inclusion* (RFI) bis hin zu SQL-Injection.

Der vorliegende Artikel behandelt in diesem Zusammenhang die drei häufigsten Sorten von Schwachstellen: Cross-Site-Scripting, SQL-Injection und Blind SQL-Injection. Für jede dieser Schwachstellen wird an jeweils einem Fallbeispiel gezeigt, wie die Schwachstelle funktioniert, wie sie erkannt werden kann und wie sie

Abbildung 1. *Kommentarfeld mit eingefügtem JavaScript*

Abbildung 2. *Netsparker Community Edition: „Scan-Settings"*

Abbildung 3. *Netsparker Community Edition: Identifizierte XSS-Anfälligkeit*

behoben wird. Dabei kommen – ähnlich wie bei einem externen Angriff – verschiedene Tools, z. B. Web-Security-Scanner, zum Einsatz. Darüber hinaus wird aber auch aus Entwicklersicht gezeigt, wie die problematischen Stellen bzw. die genauen Umstände für das Problem durch ein Code-Review und durch Log-Auswertungen auf dem Server identifiziert werden können. Im letzten Schritt wird dann anhand der gewonnenen Informationen eine beispielhafte Lösung für die jeweilige Sicherheitslücke implementiert.

Fallbeispiel „Cross-Site-Scripting"

Cross-Site-Scripting (abgekürzt „XSS") tritt in Webanwendungen immer dann auf, wenn es versäumt wurde, den Inhalt von Benutzereingaben, die letztlich in einem vertrauenswürdigen Kontext wieder an den Browser gesendet werden, zu prüfen, oder diese Prüfung ungenügend bzw. fehlerhaft ist. Dadurch wird es einem potenziellen Angreifer unter gewissen Umständen ermöglicht, an sensible Daten (z. B. Cookies) eines Benutzers zu gelangen. Häufig werden Webanwendungen über vorhandene Suchfeldern, Kontaktformulare oder ähnliche Eingabemöglichkeiten für XSS anfällig.

Cross-Site-Scripting: Funktionsweise

Dieses erste Beispiel handelt von einer News-Seite, die genau solch ein Kommentarfeld beinhaltet, um Besuchern der Seite die Möglichkeit zu bieten, eigene Kommentare zu News-Einträgen abzugeben. Hierzu müssen Name, E-Mail-Adresse und eben der gewünschte Kommentar eingegeben werden. Augenfällig wird die XSS-Angreifbarkeit, wenn zwar wie vorgesehen ein Name und eine E-Mail-Adresse eintragen werden, in das Kommentarfeld aber statt einem Text das Codefragment `"><script>alert('XSS')</script>` (Abbildung 1) eingefügt und das Formular abgeschickt wird. Im Browser erscheint daraufhin ein kleines Fenster mit der Meldung „XSS", was nichts anderes bedeutet, als dass es an dieser Stelle möglich ist, fremden Code in das Formular einzufügen und im Browserkontext der Seite auszuführen.

Da damit offensichtlich eine Anfälligkeit besteht, wird als nächster Schritt mithilfe des Web-Security-Scanners „**Netsparker Community Edition**" ein kompletter Überblick über mögliche weitere Sicherheitsprobleme des zugrundeliegenden CMS erstellt. **Netsparker Community Edition** ist die kostenlose Version des gleichnamigen Web-Application-Security-Scanners „Netsparker" aus dem Hause „mavitunasecurity". Nach dem Start kann man eine URL eingeben, verschiedene „Security Tests" auswählen und die Überprüfung starten (Abbildung 2). Gegenüber der „Professional Version" hat diese frei erhältliche Version zwar einige Einschränkungen, u. a. bei den zur Auswahl stehenden Angriffsmöglichkeiten, dennoch bietet auch diese Version die Möglichkeit, nach Anfälligkeiten für Cross-Site-Scripting und SQL-Injections (Error und Boolean Based) zu suchen.

```
root@bt:/var/www/lighteasy# fgrep -r "commentmessage" *
addons/news/main.php: if($_POST['commentname']=="" || $_POST['commentmessage']=="")
addons/news/main.php:    $commentmessage =  str_replace($order, "<br />",sanitize($_
addons/news/main.php:    $query="INSERT INTO ".$prefix."comments (newsid, poster, po
."\", \"".encode(sanitize($_POST['commentemail'])."\", ".time().", \"".encode(sani
addons/news/main1.php: $out.="<td><textarea name=\"commentmessage\" style=\"width:2
addons/news/main1.php: if($editar) $out.=sanitize($_POST['commentmessage']);
root@bt:/var/www/lighteasy#
```

Abbildung 4. *Suchergebnis von fgrep nach „commentmessage"*

```
$commentmessage =  str_replace($order, "<br />",sanitize($_POST['commentmessage']));
$query="INSERT INTO ".$prefix."comments (newsid, poster, postermail, time, text) VALUES \
(".$_POST['newsid'].",\" ".encode(sanitize($_POST['commentname']))."\", \
\"".encode(sanitize($_POST['commentemail']))."\", \
".time().", \"".encode(stripslashes(sanitize($commentmessage)))."\")";
```

Abbildung 5. *Funktion sendcomment() in der Datei main.php*

```
mysql> select * from LNE_comments;
+----+--------+--------+--------------+------------+-----------------------------------------------------------+
| id | newsid | poster | postermail   | time       | text                                                      |
+----+--------+--------+--------------+------------+-----------------------------------------------------------+
|  5 |      1 |  test  | test@test.de | 1318489938 | &quot;&gt;&lt;script&gt;alert('XSS')&lt;/script&gt; |
+----+--------+--------+--------------+------------+-----------------------------------------------------------+
1 row in set (0.00 sec)
```

Abbildung 6. *„Persistent XSS" in der MySQL-Datenbank*

Nach Abschluss des Security-Scans stehen in dem rechten unteren Fenster „Issues“ die Ergebnisse zur Verfügung. Dort wird auf der einen Seite die XSS-Anfälligkeit im Feld „*commentmessage*“ bestätigt, auf der anderen Seite werden weitere Anfälligkeiten für XSS in den Feldern „*commentemail*“ und „*commentname*“ angezeigt (Abbildung 3). Bis zu diesem Punkt kann der Angriff genauso von einem externen Angreifer durchgeführt werden, der bei Open-Source-Anwendungen überdies auch Einblick in den Quellcode nehmen kann. Bei Closed-Source-Anwendungen hat der Entwickler indes den Vorteil, dass er als Einziger die Sicherheitslücke durch Log-Auswertungen und Code-Reviews nachvollziehen kann, da er auf Quellcode und Server Zugriff hat. Letztlich ist er damit aber auch der Einzige, der durch Anpassungen und/oder Korrekturen die Sicherheitslücke im Quellcode beseitigen kann. Diese beiden Schritte werden in den folgenden beiden Abschnitten demonstriert.

Cross-Site-Scripting: Code Review

Ausgangspunkt des Code Reviews ist das gefundene XSS in „*commentmessage*“, das im Folgenden im vorliegenden Quellcode lokalisiert werden soll. Mit einem Wechsel auf der Konsole in das entsprechende Installationsverzeichnis und dem dort abgesetzten Kommando „*fgrep -r "commentmessage" **“, werden zunächst alle vorhanden Dateien nach dem String „*commentmessage*“ durchsucht, der sich schließlich in den zwei Dateien *addons/news/main.php* und *addons/news/main1.php* findet (Abbildung 4). Der erste Fundort befindet sich in der Datei *main.php* in den Zeilen 38 & 39 in der Funktion `sendcomment()` (Abbildung 5). Wie hier zu erkennen ist, wird die POST-Variable „*commentmessage*“ mit einem `str_replace()` und der Funktion `sanitize()` bearbeitet und der Variable „*$commentmessage*“ zugewiesen. Danach erfolgt in dem INSERT eine weitere Behandlung mit der Funktion `encode()`, `sanitize()` (*Anmerkung:* `sanitize()` *&* `encode()` *sind Funktionen in der Datei common.php)* und der PHP-Funktion `stripslashes()`. Wenngleich der vom Benutzer eingegebene String damit augenscheinlich mehrfach redundant gefiltert wird, bleibt die Variable offenbar dennoch für XSS anfällig.

In ähnlicher Weise können auch die XSS-Anfälligkeiten der beiden anderen Felder „*commentemail*“ und „*com-*

```
preg_replace('#</*(applet|meta|xml|blink|link|style|script|embed|object|iframe \
|frame|frameset|ilayer|layer|bgsound|title|base)[^>]*>#i', "", $text);
```

Abbildung 7. *Fehlerhaftes preg_replace() in der Datei common.php*

```
//preg_replace('#</*(applet|meta|xml|blink|link|style|script|
//embed|object|iframe|frame|frameset|ilayer|layer|bgsound|title|base)[^>]*>#i',
//"", $text);

$text = preg_replace('#</*(applet|meta|xml|blink|link|style|script|
embed|object|iframe|frame|frameset|ilayer|layer|bgsound|title|base)[^>]*>#i',
"", $text);
```

Abbildung 8. *Korrigiertes preg_replace() in der Datei common.php*

```
if(!is_intval(trim($_POST['newsid']))
|| !is_intval(trim($_POST['secCode']))
|| !is_intval ($_SESSION['operation'])
/* Patch: email validation */
|| !filter_INPUT(INPUT_POST,'commentemail', FILTER_VALIDATE_EMAIL))
/* Patch: email validation */
die ("#1 - aha! Clever!");
```

Abbildung 9. *Validierung der E-Mail-Adresse durch filter_INPUT()*

```
mysql> select * from LNE_comments;
+----+--------+--------+-------------+------------+-----------------------------------------------------------+
| id | newsid | poster | postermail  | time       | text                                                      |
+----+--------+--------+-------------+------------+-----------------------------------------------------------+
| 11 |      1 |  test  | test@test.de | 1318495368 | &quot;&gt;alert('XSS')                                    |
| 12 |      1 |  test  | test@test.de | 1318495424 | &quot;&gt;&lt;script&gt;alert('XSS')&lt;/script&gt;       |
+----+--------+--------+-------------+------------+-----------------------------------------------------------+
2 rows in set (0.00 sec)
```

Abbildung 10. *Mit sanitize() gefilterte Benutzereingabe (id 11) in der Datenbank*

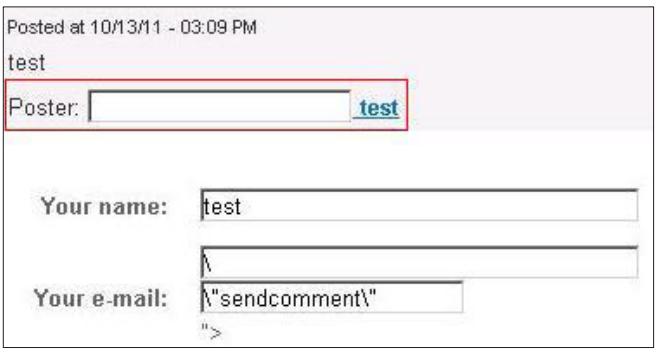

Abbildung 11. *Defacement der Webseite durch ungenügend gefilterte Benutzereingaben*

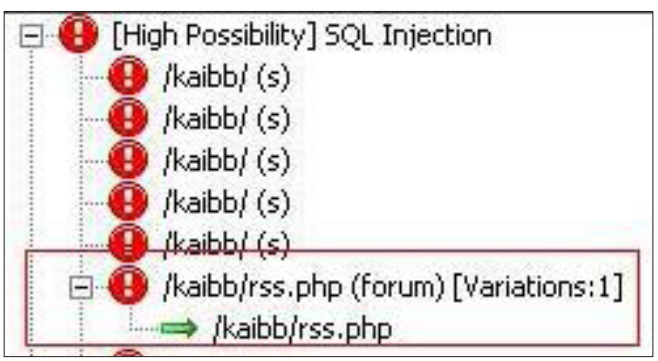

Abbildung 12. *Netsparker Community Edition: Identifizierte SQL-Injection Anfälligkeit*

mentname“ im Quellcode nachvollzogen werden. Um den Rahmen des Beitrags nicht zu sprengen, wird an dieser Stelle aber auf eine ausführliche Darstellung verzichtet.

Da die Benutzereingaben und somit auch der eingefügte XSS-Code in der Datenbank gespeichert werden (Abbildung 6), handelt es sich hierbei sogar um sogenannte „persistent XSS“. Das bedeutet, das eingeschleuste Script wird bei jedem Besuch der News-Seite erneut im Browser des Betrachters ausgeführt, ohne dass dafür vom Angreifer weitere Schritte unternommen werden müssen.

Cross-Site-Scripting: Problemlösung

Ein Blick in die Funktion `sanitize()` zeigt die Ursache für die XSS-Anfälligkeit: dort werden verschiedene Filter angewendet, um Eingaben von ungewolltem Inhalt zu befreien. Unter anderem werden beispielsweise die Zeichenketten „meta“, „embed“, „iframe“, „script“, „title“, „applet“ und „rfameset“ entfernt, was durch ein `preg_replace()` in Zeile 490 erreicht werden soll (Abbildung 7). Bei einer genaueren Betrachtung dieses `preg_replace()` fällt aber auf, dass dieses zwar den Inhalt der Variable „*$text*“ als Argument übergeben bekommt, der Rückgabewert dann aber keiner neuen Variable zugewiesen wird, womit dieser Teil der Filterung ins Leere läuft. Indem die von `preg_replace()` veränderte Zeichenkette wieder der Variablen „*$text*“ zugewiesen wird, wird dieser Umstand korrigiert und die Probleme beim Filtern mit der Funktion `sanitize()` sind damit behoben (Abbildung 8).

Eine weitere Anpassung behebt die XSS-Anfälligkeit der POST-Variablen „*commentemail*“, die in der Datei *main.php* (Zeile 30) zu finden ist. Die dort vorhandene *If*-Bedingung führt bereits Prüfungen für andere POST-Variablen durch, weshalb sie schlicht um die Anweisung zur zusätzlichen Validierung der Variable „*$_POST['commentemail']*“ erweitert wird. Dazu wird die PHP-Funktion `filter_INPUT()` genutzt und mit dieser hinzugefügten Anweisung erreicht, dass durch die Eingabe einer nicht gültigen E-Mail Adresse keine weitere Verarbeitung erfolgt bzw. der Benutzer nur mit der Eingabe einer gültigen E-Mail Adresse fortfahren kann (Abbildung 9).

Ein erneuter Angriffsversuch mit dem oben aufgeführten JavaScript auf das Kommentarfeld und ein erneuter Blick in die Datenbank offenbaren dann auch einen deutlichen Unterschied. Erstens wird das eingeschleuste Script nun nicht mehr einfach ausgeführt und zweitens werden nun bei den zukünftig in der Datenbank gespeicherten Eingabedaten die *script*-Tags herausgefiltert (Abbildung 10). Da die Funktion ebenfalls auf die Felder „*commentemail*“ und „*commentname*“ angewandt wird, sind die XSS-Anfälligkeiten auch in diesen beiden Feldern behoben. Um den Erfolg der bisherigen Maßnahmen einzuschätzen, wird eine zweite Überprüfung mit **Netsparker** durchgeführt, die allerdings für zwei der drei Felder weiterhin XSS-Anfälligkeiten feststellt, nämlich für „*commentname*“ und „*commentmessage*“. Das bedeutet, die bisherigen Anpassungen haben erst einen Teil des Problems gelöst, denn es ist nach wie vor z. B. möglich, durch die Eingabe von „*“><input type="hidden" name="submit" value="sendcomment"*“ ein Defacement der Webseite zu erreichen. Dieses Verhalten rührt daher, dass HTML-Tags offenbar auch weiterhin nicht korrekt herausgefiltert werden (Abbildung 11), wenngleich auch JavaScript nicht mehr ohne weiteres ausgeführt werden kann.

```
[15:00:05] [INFO] GET parameter 'forum' is 'MySQL >= 5.0 AND error-based - WHERE or HAVING clause' injectable
[15:00:05] [INFO] testing 'MySQL > 5.0.11 stacked queries'
[15:00:06] [INFO] testing 'MySQL > 5.0.11 AND time-based blind'
[15:00:06] [INFO] testing 'MySQL UNION query (NULL) - 1 to 10 columns'
[15:00:06] [INFO] target url appears to be UNION injectable with 7 columns
[15:00:07] [INFO] GET parameter 'forum' is 'MySQL UNION query (NULL) - 1 to 10 columns' injectable
GET parameter 'forum' is vulnerable. Do you want to keep testing the others? [y/N] n
```

Abbildung 13. *sqlmap bestätigt Anfälligkeit per SQL-Injection im Parameter „forum“*

```
SELECT * FROM kaibb_topics WHERE forum_id = '1' ORDER BY id DESC LIMIT 15
```

Abbildung 14. *Ausgeführte SELECT-Anweisung nach gewöhnlichem Seitenaufruf im MySQL-Log*

Abbildung 15. *Ausgabe der Webseite nach eingefügtem „UNION+ALL+SELECT+1,2,3,4,5,6,7+'"*

Um diese verbleibenden Probleme endgültig zu lösen, müssen die Funktion `commentform()` in der Datei *main1.php* (Zeile 13 2& 143) geändert werden. Dort werden die beiden Variablen *„$_POST['commentname']"* & *„$_POST['commentmessage']"* zwar mit der Funktion `sanitize()` gefiltert, nicht jedoch mit der Funktion `encode()`, die u. a. die für HTML-Defacements notwendigen spitzen Klammern („<") durch HTML-Entities („lt;") ersetzt. Um diese zusätzliche Filterung bzw. Prüfung erweitert sind auch die verbliebenen Probleme behoben.

Fallbeispiel „SQL-Injection"

Ähnlich wie bei Cross-Site-Scripting werden Angriffe mit SQL-Injection durch nicht ausreichend abgesicherte Benutzereingaben ermöglicht. So kann ein Angreifer z. B. durch die Eingabe eines Apostrophs einer vom Entwickler vorgesehenen SQL-Query seine eigene Query hinzufügen. Gelangt dieses Query-Paket in den SQL-Interpreter, kann der Angreifer so die Datenbank zu manipulieren und sensible Daten auslesen. Im Gegensatz zu XSS, das im Browser des Benutzers ausgeführt wird, zielt dieser Angriff per SQL-Injection direkt auf den Server bzw. die Anwendung und nur indirekt auf den Benutzer.

SQL-Injection: Funktionsweise

Im Mittelpunkt des zweiten Beispiels steht eine Forensoftware, die auf ihre Anfälligkeit für SQL-Injection überprüft werden soll. Zu Beginn wird zunächst wieder ein Security-Scan mit **Netsparker** durchgeführt. Dabei wird das Tool so eingestellt, dass nur nach SQL-Injections gesucht wird. Nach dem Scan werden unter „Issues" mehrere Probleme aufgeführt, in diesem Fall insbesondere potenzielle Anfälligkeiten für SQL-Injection an unterschiedlichen URLs (Abbildung 12).

Aus den so identifizierten Schwachstellen wird im Folgenden der Parameter *„forum"* auf seine Anfälligkeit für SQL-Injection untersucht. Dazu wird im ersten

```
if ( isset($_GET['forum']))
{
    $id = $secure->clean($_GET['forum']);
    $doGet = $db->query("SELECT * FROM " . $prefix . \
     "_topics WHERE forum_id = '".$_GET['forum']."' \
     ORDER BY id DESC LIMIT 15");
} else {
```

Abbildung 16. *Variable „$_GET[,forum']" in der Datei rss.php*

```
$doGet = $db->query("SELECT * FROM " . $prefix . "_topics
 WHERE forum_id = '". $id ."' ORDER BY id DESC LIMIT 15");
```

Abbildung 17. *Korrigierte SELECT-Anweisung in der Datei rss.php*

```
$doGet = $db->query("SELECT * FROM " . $prefix . "_topics WHERE
forum_id = '". intval($_GET['forum']) ."' ORDER BY id DESC LIMIT 15");

$doGet = $db->query(sprintf("SELECT * FROM " . $prefix . "_topics WHERE
forum_id = %d ORDER BY id DESC LIMIT 15;",$id));
```

Abbildung 18. *Alternative Problemlösung der SQL-Injection durch intval() bzw. sprintf()*

```
[2] Blind SQL Injection
~~~~~~~~~~~~~~~~~~~~~~
ID Hash:   a4ab6a0c6640e5f6b7ab75b01ce78e29
Severity: High
URL:       http://192.168.100.66/openengine/cms/website.php
Element:   link
Method:    get
Tags:      sql, blind, rdiff, injection, database
Variable: key
```

Abbildung 19. *Arachni: Anfälligkeit für eine Blind SQL-Injection im Parameter „key“*

```
[1] Blind SQL Injection
~~~~~~~~~~~~~~~~~~~~~~
ID Hash:   bc02dce8c0d8cf5cac4101c5ccf9d504
Severity: High
URL:       http://192.168.100.66/openengine/cms/website.php
Element:   link
Method:    get
Tags:      sql, blind, rdiff, injection, database
Variable: id
```

Abbildung 20. *Arachni: Anfälligkeit für eine Blind SQL-Injection im Parameter „id“*

Schritt die beim Security-Scan als verwundbar identifizierte URL aufgerufen und dabei dem Parameter „*forum*“ als Wert ein einzelner Apostroph (*http://<target>/rss.php?forum=')* zugewiesen. Im Webbrowser zeigen sich nach Abschluss der Transaktion zwar keine direkten Auffälligkeiten, aber ein Blick in den HTML-Quelltext offenbart dort die Fehlermeldung „*You have an error in your SQL syntax; …*“. Damit ist die Anfälligkeit des Parameters *„forum“* für SQL-Injection verifiziert, d. h. über Zuweisungen an diesen Parameter kann die Datenbank per SQL-Injection möglicherweise manipuliert werden. Um die Manipulationsmöglichkeiten auszuloten, wird in einem weiteren Schritt statt einem einfachen Apostroph der SQL-Befehl „*' UNION+ALL+SELECT+1,2+'* “ übergeben. In der Antwort zeigt die Fehlermeldung „*The used SELECT statements have a different number of columns*“ daraufhin an, dass die von der ursprünglich eingebauten SQL-Abfrage erzeugte Tabelle wahrscheinlich mehr als zwei Spalten hat. Ein Angreifer kann hier einfach weitere UNION-Queries mit wachsender Spaltenanzahl ausprobieren, um so die korrekte Anzahl der zurückgegebenen Spalten herauszufinden.

Diese beiden dargestellten Schritte stellen nur den Anfang eines SQL-Injection-Angriffs dar, der sich in der Regel noch aus zahlreichen weiteren SQL-Statements zusammensetzt. Um einen solchen Angriff zu automatisieren, kann ein externer Angreifer z. B. auf das Tool **sqlmap** zurückgreifen, um noch mehr Informationen über die Datenbank zu erhalten, Daten daraus zu extrahieren oder sogar zu verändern. Dieses Werkzeug ist u. a. in der Lage, einen Parameter auf die korrekte Anzahl der Spalten zu prüfen und dann durch den Einsatz unterschiedlicher SQL-Injection-Angriffstechniken auszunutzen. Darüber hinaus bietet es die Möglichkeit, automatisiert Benutzernamen und Passwörter (oder Hash-Wert) aus der verwundbaren Datenbank zu extrahieren. Im vorliegenden Beispiel wurde **sqlmap** von der Konsole mit dem Aufruf „*sqlmap.py --dbms=mysql --users -p forum -u 'http://<target>/rss.php?forum=1'*“ gestartet. Damit prüft **sqlmap** den offensichtlich verwundbaren Parameter „*forum*“ und versucht, die Verwundbarkeit auszunutzen und die Benutzerkonten aus der Datenbank auszulesen. Nach kurzer Laufzeit gelingt dies auch tatsächlich (Abbildung 13).

Abbildung 21. *Vergleich der Webseiten-Ausgabe nach „True/False“-Abfrage mit logischem ODER*

SQL-Injection: Code Review

Verfügt man über Zugang zu Server, Datenbank und Quelltext, kann man die Ursachen für die Verwundbarkeit im Quellcode nachvollziehen. Im vorliegenden Fall ist die MySQL-Datenbank dazu so eingerichtet, dass eine von der Anwendung ausgeführte Query im „*mysql.log*“ ausgegeben wird. Wird die o.g. URL wie vorgesehen per „*http://<target>/rss.php?forum=1*“ aufgerufen, wird das daraufhin ausgeführte SELECT-Statement auf die Tabelle „*kaibb_topics*“ ins Log geschrieben (Abbildung 14). Ein Blick in die Datenbank zeigt, dass diese Tabelle mit „*id, title, author_id, forum_id, date, sticky, locked*“ sieben Spalten hat.

Das UNION-Statement in der Form für sieben Spalten („*'UNION+ALL+SELECT+1,2,3,4,5,6,7+'*“) führt dazu, dass keine Fehlermeldung mehr erscheint, sich dafür aber die Ausgabe der Webseite verändert: anstelle der Meldung „*Welcome to your forum*“ wird nun die Zahl *2* ausgegeben (Abbildung 15) – dabei handelt es sich genau um die 2, die im inneren SELECT als Parameter angegeben wurde. Setzt man an dieser Stelle statt der Zahl ein `version()` ein (*'UNION+ALL+SELECT+1,version(),3,4,5,6,7+'*), so wird statt der Zahl die Version der eingesetzten MySQL-Datenbank des darunterliegenden Systems zurückgegeben.

Die Ursache des gefundenen Problems lässt sich auf die Datei *rss.php* eingrenzen, die in der URL aufgerufen wurde. Der String „*forum*“ kommt hier in Zeile 40 vor (Abbildung 16). In einem *if*-Block wird hier die GET-Variable „*$_GET['forum']*“ in einer Klasse „*secure*“ verarbeitet und der Variable „*$id*“ zugewiesen (Zeile 42). Die Klasse ist wiederum in der Datei *inc/function.php* in Zeile 247 definiert, in der die übergebene Variable „*$content*“ in der Funktion `clean()` mit `mysql_real_escape_string()` und htmlspecialchars() bearbeitet und zurückgegeben wird. Auch die die GET-Variable „*forum*“ wird so gefiltert, was die Frage aufwirft, warum sie dennoch für SQL-Injection anfällig ist. Die Antwort wird an der entsprechenden Stelle in der Datei *rss.php* offensichtlich: zwar wird „*forum*“ selbst korrekt behandelt und der Variable „*$id*“ zugewiesen, danach jedoch nicht genutzt. Die SELECT-Anweisung in der Zeile darunter beinhaltet nicht „*$id*“, sondern die ursprüngliche GET-Variable „*$_GET['forum']*“ (Abbildung 16).

SQL-Injection: Problemlösung

Im vorliegenden Fall ist die Problemlösung offensichtlich: die SELECT-Anweisung muss dahingehend angepasst werden, dass eben statt der Variable „*$_GET['forum']*“, die Variable „*$id*“ genutzt wird (Abbildung 17). Eine andere Möglichkeit, die Sicherheitslücke an dieser Stelle zu beheben, wäre die Nutzung von `sprintf()` oder `intval()`, da der Parameter „*forum*“ offenbar nur Integer-Werte erwartet (Abbildung 18).

Fallbeispiel „Blind SQL-Injection“

Bei einer sogenannten Blind SQL-Injection wird im Gegensatz zu einer „normalen“ SQL-Injection vom Server keine (Fehler-)Meldung über Erfolg oder Misserfolg des eingeschleusten SQL-Statements zurückgegeben. Stattdessen müssen unterschiedliche Verhaltensweisen, zum Beispiel veränderte Antwortzeiten und/oder

```
GET parameter 'id' is vulnerable. Do you want to keep testing the others? [y/N] y
sqlmap identified the following injection points with a total of 127 HTTP(s) requests:
---
Place: GET
Parameter: id
    Type: boolean-based blind
    Title: AND boolean-based blind - WHERE or HAVING clause
    Payload: id=/de/sendpage.htm') AND 3374=3374 AND ('GSFz'='GSFz&key=236

    Type: AND/OR time-based blind
    Title: MySQL > 5.0.11 AND time-based blind
    Payload: id=/de/sendpage.htm') AND SLEEP(5) AND ('pyHj'='pyHj&key=236

GET parameter 'key' is vulnerable. Do you want to keep testing the others? [y/N] n
sqlmap identified the following injection points with a total of 579 HTTP(s) requests:
---
Place: GET
Parameter: key
    Type: AND/OR time-based blind
    Title: MySQL > 5.0.11 AND time-based blind
    Payload: id=/de/sendpage.htm&key=236) AND SLEEP(5) AND (3781=3781
```

Abbildung 22. *sqlmap idenitifiziert Schwachstellen in den Parametern „id“ und „key“*

```
SELECT * FROM oe_page WHERE (page_key = 236 or 1=1) AND (page_status <= 0)  AND (page_status <> 3)
```

Abbildung 23. *SELECT-Anweisung mit angefügtem logischem ODER in mysql.log*

```
if (isset($_GET["id"]))
{
        $input = $_GET["id"];
}
else
{
        $input = $site_home;
}
$page = get_page($input);
```

Abbildung 24. *Zuweisung der GET-Variable „id" in der Datei page.php*

```
//$sendpage = get_page_key($_GET["key"]);

$cleankey = is_numeric($_GET["key"]);
$sendpage = get_page_key($cleankey);
```

Abbildung 25. *Problemlösung für den Parameter „id" durch is_numeric()*

ein verändertes Verhalten der Anwendung, z.B. Fehlerseiten, beobachtet werden, um eine Verwundbarkeit zu erkennen und auszunutzen.

Blind SQL-Injection: Funktionsweise

Das dritte und letzte Beispiel baut auf dem Open-Source Web-Application-Security-Scanner-Framework **Arachni** auf. Dieser Web-Security-Scanner bietet die Möglichkeit, Webanwendungen auf LFI, RFI, XSS, (Blind) SQL-Injection und andere Sicherheitslücken hin zu überprüfen. Der Angriff wird von der Konsole über den Befehl *„arachni --only-positives --mods=sqli_* 'http://<target>/cms/website.php?id=/de/index.htm"* gestartet. Mit diesem Aufruf wird die Suche auf unterschiedliche Angriffsarten für SQL-Injections eingeschränkt und es werden aus Gründen der Übersichtlichkeit nur potenziell gefundene Sicherheitsprobleme auf der Konsole ausgeben. In vorliegenden konkreten Beispiel findet Arachni zwei Lücken für Blind SQL-Injection, und zwar im Parameter *„key"* (Abbildung 19) und im Parameter *„id"* (Abbildung 20). Unterzieht man den Parameter „*key*" zunächst einem manuellen Test, indem der URL am Ende ein logisches ODER angefügt wird, ergibt sich damit die URL *„http://<target>/cms/website.php?id=/de/sendpage.htm&key=236 or 1=1*". Wenngleich keine auf der Antwortseite oder ihrem Quelltext sichtbare Fehlermeldung erzeugt wird, ist jedoch im Vergleich mit der Antwort auf den unveränderten Aufruf ein entscheidender Unterschied festzustellen: die Überschrift der Seite ändert sich je nach ODER-Statement. Wird rechts vom ODER ein „1=1" (äquivalent zu „true") angegeben, erscheint als Überschrift „Homepage (de)", für ein „1=2" (äquivalent zu „false") ändert sich die Überschrift in „Seite versenden" (Abbildung 21).

Um nun Daten aus der Datenbank zu extrahieren, kann ein Angreifer Abfragen einfügen, um durch die Auswertung der Überschrift Antworten auf Ja/Nein-Fragen zu erhalten. So kann z. B. mithilfe der SQL-Funktion REGEXP der Account eines Admin-Benutzers, der mit der E-Mail-Adresse „admin@example.com" und dem Passwort „admin" (MD5: 21232f297a57a5a743894a0e4a801fc3) in der Tabelle oe_account gespeichert ist, kompromittiert werden: mit „OR 1=(SELECT 1 FROM oe_account WHERE account_email='admin@example.com' AND account_password REGEXP '^[a-f]')" erscheint die die Überschrift „Seite versenden", d. h. die Abfrage ergibt „false". Das verrät dem Angreifer, dass der hexadezimale MD5-Hash des Kennworts nicht mit Buchstaben beginnt. Die Gegenprobe mit „OR 1=(SELECT 1 FROM oe_account WHERE account_email='admin@example.com' AND account_password REGEXP '^[0-9]')" ergibt dann auch die Überschrift „Homepage (de)", d. h. die Abfrage ergibt „true". Mittels binärer Suche kann ein Angreifer so in relativ kurzer (logarithmischer) Zeit den kompletten MD5-Hash „erraten". Eine automatisierte Form dieses Angriffs beherrscht z. B. auch das oben genannte Tool **sqlmap,** dass es auf diese Weise ermöglicht, Benutzernamen und Hash-Werte aus der Datenbank zu extrahieren (Abbildung 22).

Blind SQL-Injection: Code Review

Beginnt man die Suche nach der Ursache im Quellcode wie üblich mit fgrep nach dem Parameter *„key"* über alle Quellcodedateien, so werden unübersichtlich viele Treffer erzielt. Fokussiert man stattdessen zunächst die Datei *cms/website.php*, die in der URL steht, finden sich etliche `require()`, über die abhängige Quellcodeteile importiert werden. Verfolgt man die Importe und vergleicht die im MySQL-Log aufgezeichnetes Queries, führt die Spur zur Datei *cms/system/02_page/includes/admin.php.* Diese Datei enthält u. a. die Funktion `get_page_key()`, die offensichtlich ein zuvor im Log protokolliertes SELECT enthält (Abbildung 23). Mangels einer GET- oder POST-Variable ist jedoch noch nicht nachzuvollziehen, wie die missbräuchliche Eingabe in die Anwendung gelangt ist.

```
$query = sprintf("SELECT * FROM ".$db_praefix."page
WHERE (page_key = %d)
AND (page_status <= ".$account_status.") $access;",$page_key);
```

Abbildung 26. *Alternative Problemlösung für den Parameter „id" durch sprintf()*

```
$clean_page_path = mysql_real_escape_string($page_path);

$query = "SELECT * FROM ".$db_praefix."page
WHERE (page_path = '$clean_page_path')
AND (page_status <= ".$account_status.") $access";
```

Abbildung 27. *Problemlösung für den Parameter „key" durch mysql_real_escape_string()*

Ein fgrep nach „$_GET["key"]" ergibt eine überschaubare Anzahl von Treffern, wobei die unter den ausgegebenen Trefferzeilen vielversprechendste auf die Datei *html/modules/pagemailer/main.php* hindeutet, da hier unter anderem der Variablen „*$sendpage*" der Funktionsaufruf von `get_page_key()` mit der GET-Variable „*key*" zugewiesen wird. Die Suche nach dem zweiten Parameter „*id*" gestaltet sich ein wenig einfacher, da ein „*fgrep -r '$_GET["id"]' **"* nur zwei Dateien zurückgibt. In der Datei *cms/system/02_page/includes/page.php* wird dann auch die GET-Variable „*id*" der Variable „*$input*" zugewiesen, die wiederum der Funktion „*get_page*" übergeben wird (Abbildung 24). Die Funktion „*get_page*" ist – ähnlich wie die Funktion `get_page_key()` zuvor – in der Datei *cms/system/02_page/includes/admin.php* definiert. Damit sind die problemverursachenden Stellen im Quellcode identifiziert, denn in beiden Fällen werden keine Prüfungen oder Filterungen der Eingaben vorgenommen, somit sind diese unsicheren Parameterübergaben der Grund für die gefundenen Sicherheitslücken.

Blind SQL-Injection: Problemlösung

Ähnlich wie im vorangegangenen Fallbeispiel wird hier für die GET-Variable „*key*" ein Integer-Wert erwartet, somit ist dieses Problem auch in analoger Weise z. B. mit `is_numeric()` für die GET-Variable „*key*" (Abbildung 25) oder mit `sprintf()` in der SELECT-Anweisung (Abbildung 26) behebbar. Der problematische Parameter „*id*" kann korrigiert werden, indem die Variable „*$page_path*" vor Ausführung der SELECT-Anweisung mit der PHP-Funktion `mysql_real_escape_string()` gefiltert wird (Abbildung 27). Nach diesen Anpassungen halten die beiden Parameter einer erneuten Überprüfung mit **Arachni** und **sqlmap** stand.

Fazit

Die drei gezeigten Fallbeispiele decken mit XSS, SQL-Injection und Blind SQL-Injection die derzeit häufigsten Typen von Sicherheitslücken in Webanwendungen ab. Diese Sicherheitslücken wurden sowohl manuell und automatisiert angegriffen, als auch per Code-Review analysiert und durch entsprechende Anpassungen am Code behoben. Wenngleich Entwickler nicht immer über die Zeit und die Ressourcen verfügen, so tiefgehende Untersuchungen wie die hier dargestellten durchzuführen, zeigen die hier angeführten und aus der Praxis gegriffenen Beispiele anschaulich, dass zumindest die Überprüfung der eigenen Anwendung mithilfe des einen oder anderen (kostenlosen) Web-Security-Scanners dringend anzuraten ist. Wie man aus den Beispielen ersieht, können auch bei Verwendung von Filterfunktionen Lücken auftreten, wenn z. B. der Kontext um die Filterung nicht korrekt implementiert ist. Eine automatisierte Prüfung kostet im Grunde nicht viel Zeit und bietet den Vorteil, schon frühzeitig offensichtliche Fehler aufdecken und beheben zu können. Darüber hinaus bewirken die gezielte Suche nach Sicherheitslücken und ihre Analyse in der Regel einen signifikanten Lerneffekt. Neben einem Blick für das Wesentliche stellt sich auch schnell die Fähigkeit ein, IT-Strukturen ganzheitlich zu betrachten, das Zusammenspiel ihrer Komponenten aus sicherheitstechnischer Sicht zu beurteilen und Erfahrung mit Stärken und Schwächen verschiedener Security-Tools zu gewinnen.

Unabhängig davon, ob eine gefundene Sicherheitslücke nun zufällig oder durch eine gezielte Suche gefunden wurde, sollte wenn möglich der Kontakt mit dem oder den zuständigen Entwicklern gesucht werden, um sie über die gefundenen Probleme zu informieren. Nur so haben sie die Gelegenheit, die Sicherheitslücke kurzfristig zu beheben. In Absprache mit den Entwicklern bzw. nach Verstreichen einer angemessenen Zeitspanne, z. B. ein oder zwei Wochen, sollte auch eine Mitteilung an entsprechende Security-Mailinglisten, Sicherheitsforen oder ähnliche Stellen im Internet in Betracht gezogen werden. Zum einen werden hiermit eine große Anzahl möglicher Nutzer erreicht, zum anderen kann so eine offene Diskussion über die Sicherheitsprobleme angestoßen werden. Ein verantwortungsvolles Vorgehen ist in jedem Fall unverzichtbar, da mit einer Veröffentlichung auch böswillige Angreifer auf die Lücke aufmerksam werden können.

Im Internet

- *http://www.mavitunasecurity.com/communityedition/* – Netsparker Community Edition
- *http://sqlmap.sourceforge.net* - sqlmap - automatic SQL injection and database takeover tool
- *http://arachni.segfault.gr* - Arachni - Web Application Security Scanner Framework

STEFAN SCHURTZ, DR. PHILIPP WALTER

Stefan Schurtz
arbeitet bei der INFOSERVE GmbH in Saarbrücken
Kontakt mit dem Autor: s.schurtz@infoserve.de
Dr. Philipp Walter
ist IT-Leiter der INFOSERVE GmbH in Saarbrücken.
Kontakt mit dem Autor: p.walter@infoserve.de