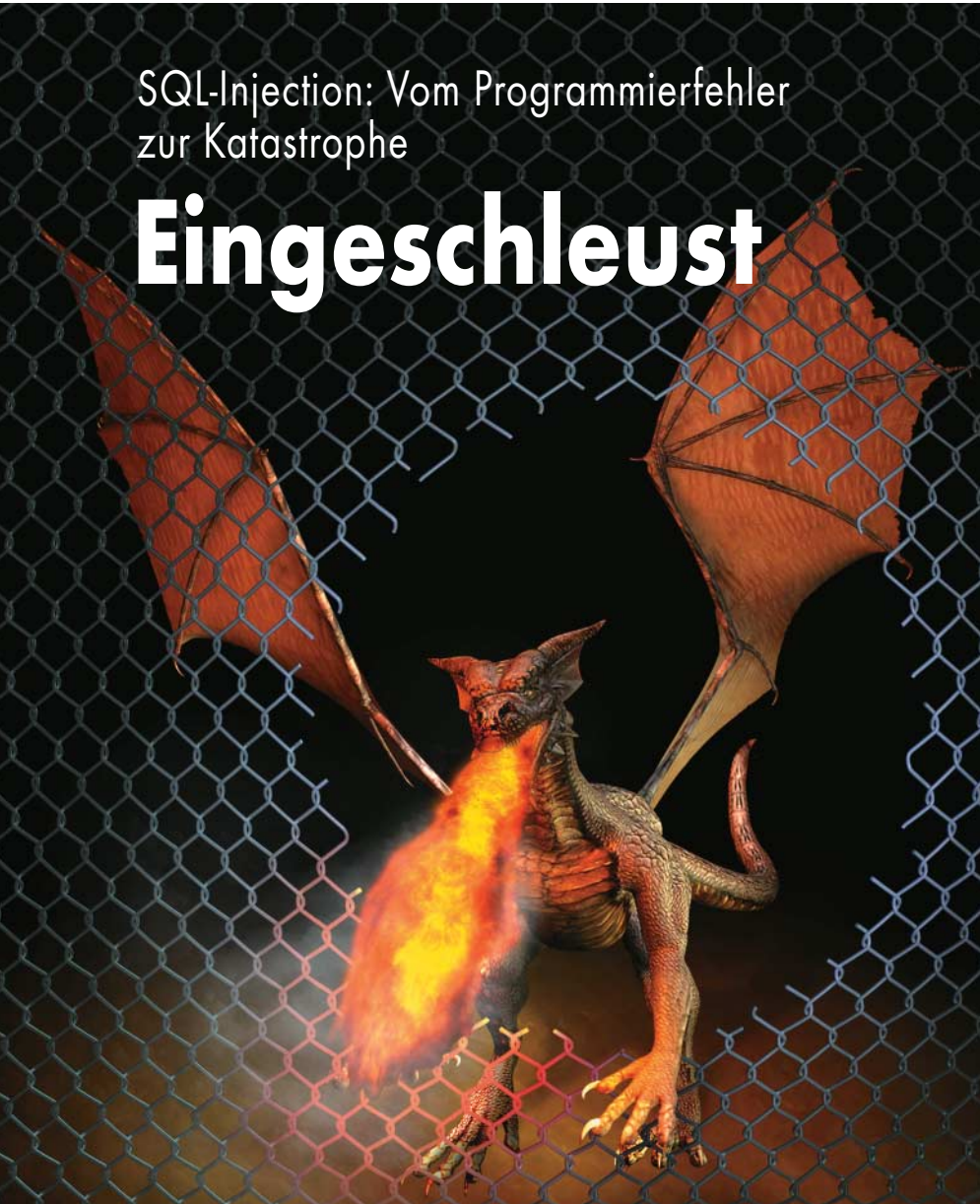


SQL-Injection: Vom Programmierfehler
zur Katastrophe

Eingeschleust



Daniel Marner, Matthias Hauß

SQL-Injection-Angriffe gehören zu den größten Bedrohungen für Webanwendungen. *iX* zeigt, wie sie im Detail funktionieren, was man dagegen unternimmt und welche fortgeschrittenen Angriffe es gibt.

Injection Flaws machen den Löwenanteil der Schwachstellen moderner Webapplikationen aus. Sie treten so häufig auf, dass das Open Web Application Security Project (OWASP) sie in seinem neuen Release-Kandidaten der zehn häufigsten Sicherheitslücken in Webanwendungen auf Platz eins führt, dicht gefolgt von der ebenfalls großen Bedrohung durch Cross-Site Scripting.

Bedeutenden Anteil an der vorderen Platzierung der Injection Flaws haben SQL-Injection-Angriffe.

Sie basieren auf Sicherheitslücken bei der Kommunikation zwischen (Web-)Anwendungen und SQL-Datenbanken. Übergibt die Webanwendung aufgrund von Programmierfehlern Eingaben unmaskiert von der Benutzeroberfläche an den zugehörigen SQL-Interpreter, kön-

nen sie Sonderfunktionen im SQL-Interpreter auslösen und damit das Ausführen von Datenbankbefehlen bewirken. Auf diese Art können Außenstehende über die schlichte Eingabe von Metazeichen in die Eingabemaske von Webanwendungen SQL-Befehle in die dahinterliegende Datenbank einschleusen und die Datenbank manipulieren oder Zugriff auf enthaltene Informationen erlangen.

Wie umfangreich der Zugriff durch das Einschleusen einfachster SQL-Befehle ist, zeigt das folgende Fallbeispiel. Es basiert auf fiktiven forensischen Daten, beschreibt aber eine typische Kompromittierung, wie sie häufig vorkommt.

Die Sicht des Angegriffenen

Im Mittelpunkt des Geschehens steht der Webserver `www.example.com`. Er läuft mit aktuellem Apache auf einem Ubuntu 9.04 und bietet 541 Benutzern Webspace. Es werden Ajax wie auch PHP eingesetzt, um den Benutzern das Design interaktiver Webseiten zu ermöglichen. Zudem gibt es unzählige Virtual-Host-Einträge für Webseiten der Benutzer, die ebenfalls auf den Webhost verweisen.

Der Zugang aus dem Internet beschränkt sich auf einige wenige Ports: Port 21 für FTP, 22 für SSH, 80 für HTTP sowie 443 für HTTPS. Alle übrigen blockiert die Firewall.

Eines Tages bricht beim Versuch eines Entwicklers, sich mit dem SSH-Client zu einem anderen System zu verbinden, der Client mit der Fehlermeldung ab, er könne nicht nach `/tmp/libz.11` schreiben. Bei der Überprüfung der Datei stellt der Entwickler fest, dass dort die unverschlüsselten Benutzernamen und Passwörter der Webspace-Inhaber mitgeloggt wurden. Und nicht nur das: Es existieren noch einige andere, ähnlich benannte Dateien, in denen sich ebenfalls unverschlüsselte Benutzernamen und Passwörter der Benutzer befinden.

Verdächtige Dateien und auffällige Requests

Eine alarmierende Entdeckung, denn das unverschlüsselte Speichern von Benutzernamen und Passwörtern ist bei ordnungsgemäßer Funktionsweise des Webhostings nicht vorgesehen. Der Administrator nimmt die Spur auf und findet verschiedene Einträge in dem Log-

file `/var/log/wtmp`, die zeigen, dass sich am Abend zuvor mehrfach verschiedene Benutzer, die in der verdächtigen Datei `/tmp/libz.111` stehen, über SSH von derselben IP-Adresse aus angemeldet haben. Eine kurze Überprüfung mit `which ssh` fördert die Datei `/usr/local/share/ssh` zutage, die nicht zum Installationsumfang der genutzten Linux-Distribution gehört. Dem Administrator ist klar, dass er es hier mit einem Sniffer zu tun hat, der Benutzernamen und Passwörter von `example.com`-Usern an einen unbekanntem Außenseiter übermittelt hat.

Beim Abgleich der SSH-Login-Zeiten mit dem Apache-Logfile fallen HTTP-Browser-Requests auf, die SQL-Syntax beinhalten:

```
GET /padawan/index.php?id=1+ORDER+BY+7- 7
HTTP/1.0 200 2326
GET /padawan/index.php?id=1+ORDER+BY+9- 7
HTTP/1.0 200 2326
```

Da die Seite `index.php` normalerweise keine SQL-Befehle entgegennimmt,

weisen diese Log-Einträge auf einen Angriff gegen eine der auf `example.com` betriebenen Webseiten hin. In diesem Fall wurde die Webseite `www.11nuxadmin.com` angegriffen, die der Benutzer `padawan` auf dem Server von `example.com` betreibt. Das Vorhandensein eines Sniffers auf dem eigenen System zeigt dem Administrator, dass es dem Angreifer gelungen sein muss, aus der Webseite `11nuxadmin.com` auszurechnen und Zugriff auf den Webhost selbst zu erlangen.

Sofortmaßnahmen nach dem Hack

Sofort ergreift der Administrator alle Gegenmaßnahmen, die an einem Produktivsystem durchführbar sind:

– Er überprüft die `passwd`-Binärdatei, um sicherzustellen, dass sie nicht durch einen Trojaner ersetzt wurde, und ändert das Root-Passwort.

– Er sperrt den Benutzer `padawan` und informiert ihn über die Kompromittierung seiner Webseite sowie die Schwachstelle in seinem PHP-Skript.

– Alle Benutzer, die in den Logfiles des Sniffers protokolliert sind oder auf deren Accounts von der IP-Adresse des Angreifers zugegriffen wurde, erhalten via E-Mail ein neues Zufalls-Passwort für eine einmalige Passwortänderung.

– Den Sniffer `/usr/local/share/ssh` mit allen zugehörigen Logfiles kopiert der Administrator zur Analyse auf einen externen Datenträger und löscht ihn vom Produktivsystem.

– Er durchsucht alle restlichen Logfiles und das Dateisystem nach Anzeichen, ob eine Kompromittierung des Root-Accounts vorliegt.

Nach einem langen Arbeitstag schließlich hat der Administrator einer Vielzahl langjähriger Kunden die unangenehme Nachricht überbracht, dass ihr Passwort gestohlen und möglicherweise ihr gesamter Account kompromittiert wurde. Er hat Stunden damit verbracht, Logfiles zu sichten und das System von dem Sniffer zu befreien. Doch es bleibt die Ungewissheit, ob es ihm gelang, den Angreifer nun wirklich dauerhaft auszusperrern, oder ob dieser noch immer Zugriff auf vertrauliche Daten hat.

Die Sicht des Angreifers

Der Angreifer sitzt am Rechner und sucht wie üblich nach leichten Zielen im Internet. Heute sucht er mit Strings



- SQL-Injection-Risiken bei Webanwendungen sind gefährlicher denn je: Vor Kurzem sind sie in der OWASP-Top-Ten-Liste der Gefährdungen auf Platz 1 gerückt.
- Basierend auf der „normalen“ SQL-Injection haben kreative Angreifer zahlreiche neue Varianten entwickelt. Bei diesen „Advanced SQL-Injections“ nutzen sie weitere Variablen für das Einschleusen von Befehlen.
- Mit verschiedenen Härtings- und Gegenmaßnahmen kann man Injection-Angriffe zumindest erschweren. Generell gilt aber: Das Absichern von Anwendungen fängt schon beim Programmieren und Überprüfen des Codes an.

Anzeige

SQL-Injections erschweren

Mit den folgenden Maßnahmen kann man sich gegen solche Injection-Angriffe auf Webanwendungen schützen.

– Es sollte eine Härtung aller Dienste wie Apache und MySQL vorgenommen werden. Im beschriebenen Fall wäre eine Einschränkung der Zugriffsrechte auf das WWW-Root der einzelnen Benutzer von erheblichem Vorteil gewesen.

– Man sollte eine Web Application Firewall (WAF) installieren. Der Installations-Aufwand einer WAF ist je nach Anwendung zwar enorm, erlaubt aber sehr effektiv das Filtern beziehungsweise Maskieren von Serverantworten (so werden Fehlermeldungen beispielsweise nur noch mit einer generischen „Error“-Meldung dargestellt) sowie das Filtern von Anfragen. Zu diesem Zweck lassen sich Regeln definieren, die typische Angriffe (etwa SQL-Injections) herausfiltern.

– Es sollten keine Dateien mit Privilegien 777 existieren. Es besteht so gut wie nie der Bedarf, Dateien mit Zugangsdaten für die Gruppe „Others“ lesbar zu machen. Daher sollte man die Rechte beim Anlegen einer Datei systemweit festlegen und überdies einen Cronjob einrichten, der regelmäßig nach Dateien mit den Rechten 777 (rwx rwx rwx) sucht und sie beispielsweise in 640 (rw- r--) bzw. 740 (rwx r--) ändert.

– Benutzer sollten generell nur das eigene Home-Verzeichnis sehen und lesen können. Daher sind entsprechende Rechte für das Verzeichnis `/home` festzulegen.

– Gleiche Passwörter sollten generell nicht für verschiedene Accounts verwendet werden. Das kann zu einem Domino-Effekt bei Kompromittierung eines der beteiligten Konten führen. Administratoren sollten hier mit Hinweisen bei Passwordeingaben und Sanktionen bei Missachtung gegensteuern.

– Die SQL-Datenbank sollte nur lokal erreichbar sein. Dabei ist zu beachten, dass zum Beispiel MySQL für den Root-Benutzer mindestens zwei Passwörter vergibt: eins für den lokalen Zugang (Quelle: 127.0.0.1) sowie eins für den externen Zugang. Hier muss man zwingend die Default-Passwörter ändern.

– Den SSH-Zugang sollte man gegebenenfalls einschränken. Ersteller von Webinhalten benötigen in der Regel keinen. Meist reicht ein FTP-Zugang, um den Inhalt hochzuladen. Sofern kein SSH- oder Telnet-Zugang zum System möglich ist, wäre das Hochladen einer PHP-Shell die zweitgrößte Gefahr. Es lässt sich aber durch Härten des Apache ebenfalls einschränken oder gar gänzlich unterbinden.

wie `inurl:index.php` und `id=` nach selbst geschriebenen PHP-Seiten. Er sucht gerne nach solchen Seiten, da diese normalerweise keiner besonderen Richtlinie für sicheres Design unterliegen. Folgeschwere Programmierfehler, die eine unmaskierte Übergabe von Benutzereingaben an SQL-Interpreter und damit SQL-Injection-Angriffe erlauben, sind bei solchen Seiten noch immer gang und gäbe.

Bei `www.linuxadmin.com` wird der Angreifer fündig. Seine Suchmaschine präsentiert ihm den folgenden Link: `www.linuxadmin.com/padawan/index.php?id=34587`.

Der Trick mit dem Hochkomma

Zunächst versucht er, die URL so anzupassen, dass der Webserver ihm nützlichere Informationen als nur eine `index.html` anzeigt. Dazu kontrolliert er, ob das PHP-Skript die eingegebenen Parameterwerte sauber prüft. Er fügt ein einfaches Hochkomma als Parameter ein: `/padawan/index.php?id='`

Die Eingabe führt sofort zu einem SQL-Fehler auf der Seite. Da die Fehlermeldung nicht abgefangen, sondern ungefiltert ausgegeben wurde, verfügt der Angreifer nun über zwei wichtige Informationen: Erstens, es steht eine SQL-Datenbank hinter der Webseite `linuxadmin.com` und zweitens, es gibt keine Web Application Firewall, die Fehlermeldungen oder Requests filtert.

Schritt für Schritt zum Angriff

Als Nächstes gilt es herauszufinden, ob der Parameter `id` für eine SQL-Injection geeignet ist. Der Angreifer beginnt, die Spalten zu zählen, indem er die Spaltennummer erhöht:

`/padawan/index.php?id=1+ORDER+BY+1--`

Die Webseite zeigt normalen Content.
`/padawan/index.php?id=1+ORDER+BY+--`

Die Webseite zeigt normalen Content.
`/padawan/index.php?id=1+ORDER+BY+3--`

Die Webseite zeigt einen Fehler.

Der Angreifer weiß nun, dass die Webanwendung zwei Spalten der Datenbank auswertet. Als Nächstes gilt es herauszufinden, ob eine der Spalten angezeigt wird:

`/padawan/index.php?id=1+UNION+ALL+SELECT+1,2--`

Der Aufruf zeigt alle Ergebnisse mit der `id=1` an und gibt zusätzlich für jede Zeile in der Datenbanktabelle die Werte 1 beziehungsweise 2 aus, sofern die erste oder zweite Spalte der Datenbanktabelle angezeigt wird.

Erste Daten durch erfolgreiche Injection

Als Ergebnis sieht der Angreifer plötzlich folgende Tabelle auf der Seite:

Name	Spieler	Punkte	Ort
padawan	gnu	3200	Köln
dizzy	horde	2800	2

Die Darstellung der Zahl 2 in der vierten Spalte der Webseite zeigt dem Angreifer, dass er mit seiner SQL-Injection Erfolg hat. Der SQL-Interpreter interpretiert den Befehl, womit die erste große Hürde beim Angriff auf die Webseite genommen ist.

Er schleust die nächste Änderung ein und ersetzt die 2 durch die Funktion `user()`:

`/padawan/index.php?id=1+UNION+ALL+SELECT+1,user()--`

Der Aufruf der Funktion `user()` liefert Informationen über den Besitzer des Datenbankaccounts und wird hier in dem Feld angezeigt, in dem zuvor die 2 stand. Das Ergebnis:

Name	Spieler	Punkte	Ort
padawan	gnu	3200	Köln
dizzy	horde	2800	padawan_db@localhost

Der Account heißt also „padawan_db“ und ist auf den lokalen Zugriff beschränkt. Das Resultat bringt den Angreifer zunächst nicht voran – es sei denn, dem Datenbank-Administrator ist ein Fehler in der Datenbank-Konfiguration unterlaufen. In dem Fall kann der Angreifer vielleicht Dateien über die Datenbank einlesen und als Web-Content darstellen. Er setzt einen weiteren Befehl ab:

`/padawan/index.php?id=1+UNION+ALL+7
SELECT+1,load_file(char(47,101,116,99,7
47,112,97,115,115,119,100))--`

Die Funktion `load_file` liest hier eine Datei mit den Rechten der Datenbank.

Zuvor konvertiert die Funktion `char()` den String `47,101,116,99,47,`

112,97,115,115,119,100, der lediglich die Dezimaldarstellung des Strings */etc/passwd* ist. Anschließend zeigt der Webbrowser nun:

```
Name   Spieler Punkte Ort
padawan gnu      3200 Köln
dizzy   horde    2800 root:x:0:0:root:/root:/bin/bash
...
user1:x:502:502:/home/user1:/bin/false
malcolm:x:503:503:/home/malcolm:/bin/bash
...
jimmi:x:1057:1057:/home/malcolm/www/jimmi:/bin/bash
padawan:x:1058:1058:/home/padawan:/bin/bash
```

Der Angreifer speichert die Liste als *passwd*. Vielleicht kann er sie ja später noch gebrauchen.

Seine Anfrage `wc -l passwd` liefert ihm 548 Zeilen. Ein weiteres `grep bash passwd | wc -l` zeigt 80 Zeilen. Der Angreifer kann sich freuen: Er hat es hier mit um die 500 Benutzern zu tun, von denen 80 einen Shell-Zugang haben. Der Server, von dem er die Datei

passwd geladen hat, scheint also deutlich mehr Webseiten zu beherbergen als nur 11nuxadmin.com.

Zugriff über Umwege

Zudem ist der Server so schlecht gesichert, dass er durch die SQL-Injection mit den Rechten der Datenbank von 11nuxadmin.com Daten auf dem Dateisystem des darunterliegenden Servers auslesen kann. Dadurch kann er über die Webseite 11nuxadmin.com nun Angriffe auf den Webhost durchführen. In erschreckend kurzer Zeit hat er durch Eingabe einiger weiterer SQL-Befehle auch Zugriff auf die *.bash_history*, in der alle vom Benutzer padawan ausgeführten *bash*-Befehle geloggt sind.

Anhand der Dateien *passwd* und der *.bash_history* lässt sich die Verzeichnis-Struktur des Servers schnell rekonstruieren. Alle Benutzer haben ihren Web-Content offenbar in *\$HOME/www*.

Nach erfolgreicher Suche nach Zugangsdaten im Home-Verzeichnis von padawan findet der Angreifer eine Datei namens */home/padawan/www/inc/*

mysqlaccess.pl, die er ebenfalls durch eine SQL-Injection ausliest. Die Datei enthält einen Eintrag der Form:

```
$username = "jimmi"
$password = "38ff3f"
```

Beim Versuch, das Datenbank-Passwort für das SSH-Login zu verwenden, erhält der Angreifer plötzlich eine Shell. Von da an wird es wieder Routine. Er sucht nach allen Dateien, die mit den derzeitigen Privilegien lesbar sind, in ihnen sucht er nach Zugangsdaten, er testet die Passwörter mit dem Benutzernamen, in dessen Homeverzeichnis er die Datei mit den SSH-Zugangsdaten gefunden hat. Das Ergebnis: Bei ungefähr 30 Benutzern funktioniert das gefundene Passwort auch als Shell-Account bei diesem Webhost. Zu guter Letzt durchsucht der Angreifer die lokale Datenbank mit allen Zugangsdaten nach Account-Tabellen. Er hofft auf eine direkte Ausgabe wie:

Name	uname	email	pass
alice	ally	ali@gmx.com	wonderland
Eduardo	eddy	edd@yahoo.com	elmariachi
jarvis	jarvis	jarvis@msn.com	secured4hack!
...			

Anzeige

Stattdessen erhält er eine Tabelle, die lediglich mit MD5 gehashte Passwörter enthält. Für den Angreifer stellt dies aber kein großes Hindernis dar. Mithilfe eines Skripts gleicht er die Hashes mit Onlinedatenbanken ab. Komplexe Passwörter sind damit zwar nicht zu knacken, doch das spielt keine große Rolle. Immerhin hat die gefundene Tabelle 940 Einträge. Selbst wenn nur die Hälfte dieser Benutzer ein schwaches Passwort verwendet und davon wiederum nur die Hälfte dasselbe Passwort aus Nachlässigkeit

auch für den E-Mail-Account benutzt, bleiben dem Angreifer hier trotzdem noch ungefähr 230 E-Mail-Accounts, deren Zugang ihm praktisch offensteht.

Einfacher geht's kaum: Daten mitschneiden

Der Angreifer findet noch vier Tabellen in ähnlicher Größe, bevor der Abend zu Ende geht. Er installiert noch schnell eine Hintertür (Backdoor), die die SSH-Passwörter der Benutzer loggt, sollten

diese sich noch mit anderen Systemen verbinden wollen. Ein Eintrag in der Datei *.profile* beziehungsweise *.bashrc* im Home-Verzeichnis des Benutzers reicht für die Eintragung eines weiteren Pfades in die Path-Variable aus. So wird

```
PATH="/usr/local/bin:/usr/local/sbin:/usr/kde3/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games"
```

geändert in

```
PATH="/usr/local/share/bin:/usr/local/bin:/usr/local/sbin:/usr/kde3/bin:/usr/lo
```

Fortgeschrittene SQL-Angriffe

Basierend auf einer „normalen“ SQL-Injection, die durch Einfügen eines Hochkommata erlaubt, Befehle auf einem SQL-Server auszuführen, erweitern Advanced SQL-Injections dieses Angriffs-Szenario. Zwei Beispiele verdeutlichen das Vorgehen einer Advanced SQL-Injection.

Beispiel 1: Einschleusen von UNION- und LOAD-FILE-Statements

MySQL stellt die LOAD-FILE-Funktion bereit, die das Einlesen von lokalen Dateien auf dem System bewirkt. Wird sie in Kombination mit einer SQL-Injection verwendet, kann ein Angreifer lokale Dateien auf dem Datenbankserver mit den Rechten des Serverprozesses auslesen. Ein möglicher Angriff auf eine lokale *passwd*-Datei könnte wie folgt aussehen:

```
/index.php?id=3' UNION SELECT LOAD_FILE('/etc/passwd')--
```

Als Folge würde der Angreifer eine Liste aller eingerichteten Systembenutzer erhalten. Ebenso kann er Zugriff auf alle weiteren Dateien des Systems erlangen. Sind die Rechte der Home-Verzeichnisse auf dem Webserver nicht restriktiv genug gesetzt, lassen sich über den obigen Angriff in vielen Fällen auch *.htaccess*-Dateien auslesen.

Da inzwischen viele IDS/IPS derartige Angriffe erkennen können und entsprechende Strings filtern, könnte ein Angreifer bei MySQL den String einfach in Hex kodieren. Für Windows sähe das etwa so aus:

```
/index.php?id=3' UNION SELECT LOAD_FILE 7  
(0x633A5C626F6742E696E69)
```

Ein IDS/IPS könnte das übersehen und zulassen, dass man die Datei *c:\boot.ini* auslesen kann.

Beispiel 2: INSERT-/UPDATE-Statement-Injection

Eine weitere Angriffsvariante besteht darin, Skripte durch SQL-Injection bei einem INSERT- oder UPDATE-Statement auf den Server zu laden. So kann ein Angreifer etwa permanente Cross-Site-Skripte in der Datenbank speichern und beim Aufruf der Webseite vom Webserver ausführen lassen. Wie das funktioniert, zeigt das folgende Beispiel eines PHP-Skripts, dessen Zweck das Anlegen neuer Benutzer mit vordefinierten INSERT-Statements ist. Hier wird die Variable „user“ dem INSERT-Statement ungefiltert übergeben, was eine SQL-Injection ermöglicht:

```
<form action="register.php" method="post">  
Firstname: <input type="text" name="user" />  
<input type="submit" />  
</form>
```

Sofern nun das Skript *register.php* keine Variable überprüft, übernimmt es die Benutzer-Eingaben ungefiltert in das SQL-Statement und übergibt es an die Datenbank. Übergibt der Angreifer ein Skript, das der Webserver beim späteren Auslesen interpretiert, sind die Möglichkeiten zum Ausnutzen der Schwachstelle zahlreich. Denkbar wäre hier eine PHP-Shell zum Steuern des Servers oder ein Javascript-Programm, das Session-Informationen sammelt und verwertet.

Ein alternativer Angriff über eine HTTP-GET-Anfrage könnte folgendermaßen aussehen:

```
/register.php?user=<script>alert("xss")</script>
```

Würde der Angreifer hier ein Skript in der Datenbank ablegen, das die Cookies ausliest und versendet, könnten im schlimmsten Fall die Sessions aller Benutzer, die den Namen des Angreifers angezeigt bekommen, kompromittiert werden.

Immer neue Injection-Varianten

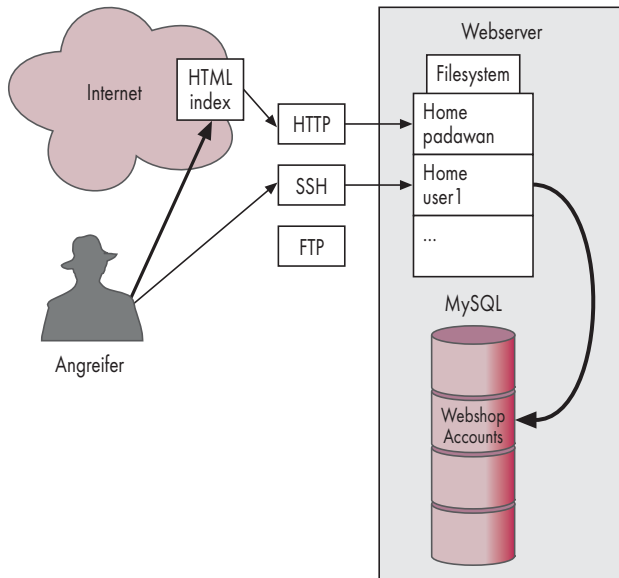
Man sollte zudem beachten, dass sich SQL-Injections nicht nur durch \$_GET-Variablen einschleusen lassen. Kreative Angreifer haben inzwischen Wege gefunden, SQL-Injections in \$_POST, \$_COOKIE und HIDDEN FORM FIELDS zu platzieren. Selbst wenn ein IDS häufig alle \$_GET- und \$_POST-Variablen prüft, vernachlässigt es oftmals die Kontrolle der Inhalte von HIDDEN FORM FIELDS.

Weitere interessante Ansätze bieten hier zudem Technologien wie Ajax. Formulare, die mit Ajax-Requests kommunizieren, können in der Adressleiste des Browsers typischerweise so aussehen:

```
http://www.sampleshop.fi/01_ShoppingCart/cart.php
```

Ändert jemand die Menge der Artikel im Einkaufswagen, generiert Ajax einen Request, der mit Tools wie TamperData nicht mehr verändert werden kann. Verwendet man jedoch beispielsweise das OWASP-Sicherheitstestwerkzeug WebScarab, lassen sich die Anfragen vor dem Absenden an den Server modifizieren. An dieser Stelle ist es dem Angreifer wieder möglich, SQL-Injections in die Variablen einzusetzen und einen Fehler zu provozieren. Das Prinzip der Injection ist somit gleich geblieben, wohingegen sich der Angriffsvektor geändert hat. Hier gehen Entwickler oft fälschlich davon aus, dass der Benutzer die Werte nicht verändern kann.

Was kann man dagegen unternehmen? Bedenkt man, dass nahezu alle Advanced SQL-Injections auf den Schwachstellen einer „normalen“ SQL-Injection basieren, greifen bei ihnen die gleichen Abwehrstrategien, etwa das Filtern SQL-spezifischer Sonderzeichen. Für die meisten Programmiersprachen existieren vordefinierte Funktionen und Frameworks, die bereits die gängigen Maßnahmen zur Abwehr von SQL-Injections implementiert haben. Das erleichtert die Entwicklung sicherer Webanwendungen deutlich.



Die größten Einfallstore für einen Angreifer sind die Dienste an der Schnittstelle zwischen Webserver und Internet. Man sollte sie daher auf das notwendige Minimum reduzieren.

bei sämtlichen auf dem Webhost betriebenen Webseiten deutlich reduzieren.

Man muss sich allerdings darüber im Klaren sein, dass es sich dabei lediglich um kompensierende Maßnahmen handelt, die das Ausnutzen von durch Programmierung entstandenen Schwachstellen der gehosteten Webseiten verhindern sollen. Einen absoluten Schutz gegen SQL-Injection-Angriffe bieten auch sie nicht.

Schutz auf allen Ebenen

Vielmehr ist es notwendig, dass auch die Programmierung der gehosteten Webseiten gewissen Sicherheitsanforderungen genügt. Dazu gehört insbesondere das Einhalten anerkannter Coding-Richtlinien zur Vermeidung gängiger Sicherheitsschwachstellen. Gute Hinweise liefern unter anderem Best-Practice-Ansätze zur Absicherung von Webseiten, beispielsweise der OWASP-Guide des Open Web Application Security Project (www.owasp.org).

Darüber hinaus können Sicherheits-scanner wie Nessus, Grendel, Nikto oder Firefox-Plug-ins wie SQLme eigene Webseiten automatisiert testen, um die schwerwiegendsten Programmierfehler zu erkennen, bevor sie auf dem Produktivsystem landen. Unabhängig davon bietet es sich an, vor Inbetriebnahme der Webseite ein Code Review von einem zweiten, an der ursprünglichen Programmierung nicht beteiligten und auf sichere Programmierung spezialisierten Entwickler vornehmen zu lassen.

Eine angemessene Absicherung gegen SQL-Injection-Angriffe kann man weder alleine auf der Ebene des Webhosts noch der Webseite erreichen. Vielmehr müssen die Verantwortlichen auf beiden Ebenen Sicherheitsmaßnahmen ergreifen, um der Gefahr gemeinsam zu begegnen. (ur)

DANIEL MARNER

ist Berater und Experte für Penetrationstests (Certified Ethical Hacking) sowie Sicherheit von Webanwendungen bei der SRC GmbH in Bonn.

MATTHIAS HAUSS

ist Berater und Experte für Sicherheit in Prozessen und Sicherheitsmanagement bei der SRC GmbH in Bonn.



`cal/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games`

Die kleine Änderung führt dazu, dass ein Benutzer, der SSH ausführen will, stattdessen den Sniffer in `/usr/local/share/bin/ssh` ausführt.

Die Sicht des Forensikers

Die dargestellte Vorgehensweise zeigt eine typische Kompromittierung durch einen SQL-Injection-Angriff. Der Beschreibung liegt ein realer Fall zugrunde, bei dem der Angreifer tatsächlich Zugriff auf große Tabellen mit Benutzernamen und Passwörtern hatte. Eine Analyse der Methodik legt die Vermutung nahe, dass der Angreifer nicht professionell agierte. Auch verhielt er sich dem System gegenüber nicht destruktiv. Dennoch verschaffte er sich Zugriff auf eine Vielzahl vertraulicher Daten, darunter einige Tausend E-Mail-Accounts, 30 MySQL-Accounts und 25 lokale Shell-Accounts.

Ob und in welchem Umfang der Angriff weitere Folgen hatte, bleibt im Dunkeln. Nicht selten geht die Kompromittierung solcher Accounts aber mit weiteren Kompromittierungen einher, zum Beispiel dem Verlust von Kontodaten zu Internethändlern wie Amazon.de, aber auch zu PayPal oder eBay und ähnlichen Portalen. Es ist anzunehmen, dass dies hier ebenso der Fall war.

Die Abbildung stellt die groben Angriffsvektoren des Einbruchs dar. Am meisten gefährdet, so ist zu sehen, sind die Dienste, die sich an der Schnittstelle zwischen Internet und Webserver be-

finden. Um das Risiko einer SQL-Injection zu minimieren, sollte man sie daher auf das Notwendigste reduzieren. Im beschriebenen Fall jedoch hatte der Angreifer bereits erfolgreich stattgefunden. Für die Betreiber des Webhosts haben daher zunächst Schadensbegrenzung und forensische Aufarbeitung Priorität. Neben den eingangs bereits beschriebenen Gegenmaßnahmen sollten sie daher vor allem drei direkte Maßnahmen ergreifen:

- Falls das System zum Abwickeln von Zahlungsverkehr dient, sollte man unmittelbar, nachdem man einen Angriff bemerkt hat, ein Image (mit Prüfsummen) des Systems erstellen. Ohne ein solches Image sind forensische Analysen im Nachhinein erheblich erschwert.
- Es sollte ein Rootkit Hunter (etwa *chkrootkit* für Linux) sowie zuverlässige Anti-Virus-Software installiert sein oder zumindest nach der Image-Erstellung installiert werden.
- Von allen Logdateien des Apache und der MySQL-Datenbank sowie eventuell beteiligter Intrusion-Detection-/Intrusion-Prevention-Systeme (IDS/IPS) und Firewalls sind Backups zu erstellen, die Forensiker später analysieren und aufarbeiten können.

Vorbeugen ist besser als „heilen“

Um zukünftige SQL-Injection-Angriffe zu verhindern, sollten die Hostanbieter die im Kasten „SQL-Injections erschweren“ genannten präventiven Gegenmaßnahmen ergreifen. Durch ihr konsequentes Umsetzen lässt sich die Gefahr eines SQL-Injection-Angriffs