Techniken zur Verhinderung von SQL Injection Attacken

Datum: 6. Dezember 2007

Michael Pröstler - 0304384
e0304384@student.tuwien.ac.at

Zusammenfassung

SQL Injection Attacken stellen ein großes Sicherheitsrisiko in Webapplikation dar. Alleine im Monat November (2007) wurden laut milw0rm.com 25 neue mögliche Attacken auf diverse Webapplikationen veröffentlicht. In diesem Artikel wird die prinzipielle Problematik von SQL Injection Attacken erklärt. Weiters werden zwei verschiedene Ansätze zu deren Verhinderung auf ihre Tauglichkeit untersucht und die jeweiligen Vor- und Nachteile dieser Techniken erläutert.

1 Einleitung

SQL Injection Attacken (SQLIA) sind eines der wesentlichen Sicherheitsproblemen in Webapplikationen.(HOM06) Alleine im Monat November (2007) wurden auf milw0rm.com 25 mögliche SQL Injection veröffentlicht.(mil) Im Durchschnitt kommt man hier auf fast eine neue Angriffsmöglichkeit pro Tag, wobei natürlich die Anzahl an unveröffentlichten Sicherheitslöchern weit aus größer sein kann. Dies alleine zeigt die Popularität von SQL Injections Attacken und leider auch die unzureichende Absicherung gegen dieselben von Seiten der Entwickler der Webapplikationen. Es gibt nun verschiedene Lösungsansätze die sich in ihrem jeweiligen Ansatz stark von einander unterscheiden. In diesem Artikel werden zwei verschiedene Ansätze genauer beleuchtet und die jeweiligen Vor- und Nachteile erläutert.

2 SQL Injections

Die Methodik einer SQL Injections Attacke besteht hauptsächlich darin eine Abfrage auf eine Datenbank zu verändern, sodass die eigentliche Funktionalität abgewandelt wird. Gelingt dies, so können beliebe SQL Befehle vom Angreifer eingeschleust werden, wodurch dieser unautorisierten Zugang zu Daten erlangen kann, beziehungsweise im schlimmsten Fall die Kontrolle über die gesamte Datenbank übernimmt. Um eigene Befehle einschleusen zu können, muss es dem Benutzer natürlich möglich sein, eigene Bestandteile in eine Datenbank Abfrage einzubauen. Dies gelingt in den meisten SQL Injection Attacken über die Modifikation gewisser Eingangsparameter. Eine Webapplikation könnte im Grunde folgenden vereinfachten Code (entnommen aus) enthalten:

```
1. String login = getParameter("login");
2. String pin = getParameter("pin");
3. Statement stmt = connection.createStatement();
4. String query = "SELECT acct FROM users WHERE login='";
```

```
5. query += login + "' AND pin=" + pin;
6. ResultSet result = stmt.executeQuery(query);
7. if (result != null)
8. displayAccount(result); // Show account
9. else
10. sendAuthFailed(); // Authentication failed
```

Hier sieht man wie in Zeile 5 die gesamte Abfrage aus einer Kombination von vordefinierten Zeichensätzen und aus den zwei Parametern login und pin zusammengesetzt wird. Wenn hier die Werte „Willy“ und 123 angegeben werden so entsteht die folgende Abfrage, welche auf der Datenbank exekutiert wird.

```
SELECT acct FROM users WHERE login='Willy' AND pin=123
```

Diese Abfrage retourniert `null` falls kein entsprechender User mit dem pin 123 gefunden wurde, ansonsten wird der Account retourniert. Wenn man als Angreifer allerdings als login den Wert „admin '–“ wählt, wird folgende Abfrage erstellt.

```
SELECT acct FROM users WHERE login='admin' -- 'AND pin=
```

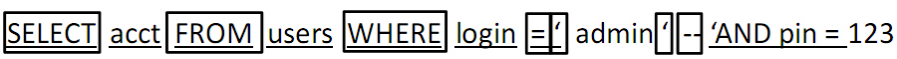
Da in SQL „- -“ das Kommentarzeichen darstellt, wird der Rest hinter dem „–“ einfach ignoriert und der Angreifer hätte es somit geschafft, die Logik der Abfrage zu verändern. In diesem Fall wurde die Bedingung auf Gleichheit des Pins durch die Manipulation übergangen. Natürlich ist es möglich die Eingaben vom User zu filtern und zu versuchen sämtliche Keywörter und Zeichen zu filtern um so Angriffe zu verhindern. Hier entsteht allerdings das Problem der Vollständigkeit. Mit dieser Methodik ist es relativ schwer und aufwendig sämtliche Angriffsmöglichkeiten zu filtern. Deshalb wurden andere, leichter anwendbare Verfahren entwickelt von denen hier zwei genauer erläutert werden.

3 Positiv Tainting and Syntax-Awarness

Der erste Ansatz um SQL Injection Attacken zu verhindern basiert auf zwei Techniken. Zum einen das Tainting zum andere das Miteinbeziehen des Syntax der SQL Abfragesprachen.
Prinzipiell geht es beim Tainting um das Markieren von Daten die ein bestimmtes Merkmal aufweisen. Während beim negativen Tainting die Werte markiert werden die potentiell gefährlich sind, werden beim positiven Tainting hingegen jene Werte markiert, denen vertraut werden darf. Als potentiell gefährlich werden beim negativen Tainting beispielsweise Daten angesehen, die vom User eingegeben werden, wie im obrigen Beispiel die Werte von „login“ und „pin“.
Der Vorteil beim positiven Tainting gegenüber dem negativen liegt vor allem im Verhalten bei nicht vollständigem, falschen Tainting. Beim negativen Tainting können Daten die fälschlicherweise nicht als unsicher markiert sind Sicherheitslücken hervorrufen. Im Gegensatz dazu, werden beim positiven Tainting die Sicherheitskriterien enger gezogen, sodass keine Sicherheitslücken entstehen können. Somit sind beim positiven Tainting keine „false negatives“ möglich, während das bei negativ tainting sehr wohl der Fall sein kann.(HOM06)
Bei Syntax-Awarness handelt es sich prinzipiell um Techniken die auf den Syntax einer Sprache eingehen, beziehungsweise auf dem Syntax einer Sprache beruhen. Im Falle

von SQL würden eine Syntax-Aware Technik besonders Wert auf die Keywörter und auf bestimmte Zeichenfolge Wert legen. Keywörter in SQL sind unter anderem Wörter wie „SELECT“,“FROM“, „WHERE“, „AND“, etc. Unter Metazeichen würde zum Beispiel der Kommentar („- -’“) fallen.
Diese zwei Techniken wurden in dem Artikel(HOM06) verwendet um Webapplikationen gegen SQL Injection Attacken zu schützen. Dabei wird positive Tainting eingesetzt um sämtliche Daten, die bereits bei der Programmerstellung vorhanden sind, als sicher zu markieren. Die Markierung der Daten beschränkt sich dabei auf den wesentlichen Datentyp String der bei der Erstellung von Datenbankabfragen eingesetzt wird. Auf der Programmiersprache „Java“ aufbauend, wurde ein Prototyp entwickelt, der genau diese Technik des positiven Taintings in den Bytecode des Java-Programmes einschleust. Um dies zu bewerkstelligen, werden sämtliche Aufrufe der Klasse „String“ durch eine eigens modifizierte und um Metadaten erweiterte Klasse ersetzt. Diese Klasse agiert wie die ersetzte und enthält zusätzliche die Tainting spezifischen Daten. Die Syntax-Awarness kommt dann beim nächsten Schritt zum Tragen. Jene Methode, die eine SQL Abfrage an die Datenbank weiterleitet, wird ebenfalls im Bytecode gering modifiziert. Dabei wird vor dieser Methode eine Prüfung durchgeführt die auf dem Syntax der SQL Sprache beruht. Dabei wird die Anfrage in ihre Keywörter und Zeichen zerlegt und überprüft ob die Keywörter nur aus sicheren Daten bestehen. Falls das der Fall ist, wird die Abfrage an die Datenbank weitergeleitet, anderenfalls wird sie verworfen. Die folgenden aus (HOM06) entnommenen Grafiken verdeutlichen die Funktionsweise.

Die mit einem Rechteck umrandeten Zeichenfolgen stellen jeweils Keywörter beziehungsweise spezielle Zeichen der SQL Sprache dar. Die unterstrichenen Zeichenfolgen gelten jeweils als sicher. Aus der ersten Grafik geht hervor, dass Daten die vom Benutzer stammen nicht vertraut wird. Das ist aber in Ordnung da sämtliche Keywörter und Sonderzeichen vertrauenswürdig sind, und somit wird die Abfrage zur Datenbank durchgereicht.



Bei der zweiten Abfrage ist wiederrum nur der Input vom Benutzer nicht als sicher markiert. Hier tritt allerdings unsicherer Input als Sonderzeichen in Form von „’“ und „- -“ auf. Dies wird erkannt und dadurch nicht zur Datenbank zugelassen.

Dieser Prototyp wurde gegen diverse Attacken von unabhängigen Personen geprüft und erzielte dabei sehr gute Ergebnisse wie aus der aus (HOM06) entnommenen Abbildung 1 zu sehen ist.

4 Instruction-Set Randomization

Eine zweite Möglichkeit sich vor SQL Injection Attacken zu schützen wir in dem Artikel (BK04) beschrieben. In dem Artikel wird eine Möglichkeit beschrieben, wie man die SQL Abfragen selbst schützt. Hierzu werden quasi neue Keywörter für die Sprache generiert indem einfach ein randomisierter Integer-Wert hinzugefügt wird. Eine solche

Subject	Total # Attacks	Successful Attacks: Original Web Apps	Successful Attacks: WASP Protected Web Apps
Checkers	4,431	922	0
Office Talk	5,888	499	0
Empl. Dir.	6,398	2,066	0
Bookstore	6,154	1,999	0
Events	6,207	2,141	0
Classifieds	5,968	1,973	0
Portal	6,403	3,016	0

Abbildung 1: Grafik entnommen aus (HOM06)

Transformation wird durch die folgenden zwei SQL-Abfragen verdeutlicht (Entnommen aus (BK04).

```
select gender, avg(age)
  from cs101.students
     where dept = %d
  group by gender
```

Durch Anhängen eines randomisierten Integer-Wertes entsteht folgende Abfrage.

```
select123 gender, avg123(age)
  from123 cs101.students
     where123 dept = %d
  group123 by123 gender
```

Durch diese Transformation kann ein potentieller Angreifer keine eigenen SQL Befehle mehr einschleusen, da ihm grundsätzlich das Wissen über die Sprache fehlt. Im Grunde verhindert man hier nur indirekt eine SQL Injection indem man dem Angreifer das Wissen über die zu Grunde liegende SQL Sprache nimmt. Eine Abfrage die Keywörter ohne angehängten Integer enthält würde nicht mehr dem Syntax der Sprache genügen und somit einen syntaktischen Fehler verursachen. Natürlich wäre es nicht wirklich sinnvoll den Interpreter der Datenbank selbst zu ändern, da dies relativ umständlich wäre, beziehungsweise die Datenbank dann nur mehr über diese modifizierte Sprache ansprechbar wäre. Deshalb schaltet man einen Proxy vor die Datenbank der im Grunde nichts anderes als eine syntaktische Analyse durchführt, bevor er eine SQL Abfrage wieder in die ursprüngliche Sprache übersetzt und zur Datenbank abschickt. Abbildung 2 zeigt eine solche Systemarchitektur (Entnommen aus (BK04)).

Prinzipiell ist es bei diesem Ansatz natürlich sehr wichtig, dass die SQL Abfragen vor einem potentiellen Angreifer versteckt werden. Also die abgesetzte Anfrage sollte auch nicht im Fehlerfall dem Benutzer retourniert werden, da er sonst den randomisierte Integer-Wert auslesen könnte und somit wieder SQL Injection Attacken ausführen könnte.

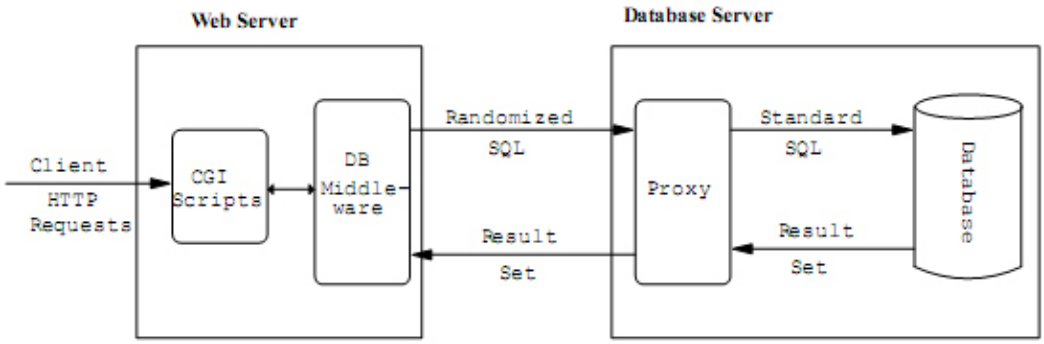


Abbildung 2: Systemarchitektur

Users	Min	Max	Mean	Std
10	74	1300	183.5	126.9
25	73	2782	223.8	268.1
50	73	6533	316.6	548.8

Abbildung 3: Proxy Overhead (in mikrosekunden)

Um diesen Ansatz zu prüfen wurde ein Prototyp in der Programmiersprache C implementiert. Weiters wurden Entwicklern Tools zur Transformation der jeweiligen SQL Statements zur Verfügung gestellt. Die Architektur die in Abbildung 2 gezeigt wird, kann natürlich auch hingehend verändert werden, sodass der Proxy auf einer anderen physikalischen Maschine läuft. Da der Proxy ja als „Man in the Middle“ fungiert und eine syntaktische Analyse durchführen muss, benötigt er natürlich auch einen modifizierten SQL Parser. Dieser wurde mit Hilfe von „flex“ und „yacc“ implementiert. Ein wesentliche Schwierigkeit bei der Implementierung bestand darin, das Protokoll der remote abgesetzten SQL Abfragen nach zu implementieren und in den Proxy zu integrieren. Im Prototyp wurde ein Proxy implementiert der das MySQL Protokoll unterstützt. Dieser Prototyp wurde mit verschiedene Webapplikationen getestet unter anderem mit dem phpBB Bulletin Board und auch mit Php-Nuke. In beiden Fällen ließ der Proxy keine schadhaften SQL Abfragen zur dahinterliegenden Datenbank durch. Auch der Overhead viel sehr gering aus, was in der aus (BK04) entnommenen Tabelle in Abbildung 3 deutlich wird.

5 Vergleich

Im Grunde sind beide Ansätze vertretbar und es wurden auch in beiden Bereichen Proof-of-Concept Prototypen implementiert. Dennoch scheinen beide Varianten Vorteile sowie Nachteile zu haben. Ein wesentlicher Vorteil dem ersten Ansatz stellt sicherlich die Integration in bestehende Applikationen dar. Da hier im Grunde nur der Bytecode verändert wird, sind keine Änderungen im Source Code beziehungsweise der darunterliegenden Runtime notwendig. Als Nachteil kann natürlich angesehen werden, dass der Ansatz bei Sprachen wie C, deren Kompilate Maschinenbefehle sind, nur schwer durchzuführen ist.

Ein wesentlicher Vorteil des zweiten Ansatzes ist natürlich die Sprachenunabhängigkeit. Durch den Einsatz eines Proxies vor der Datenbank ist dieser Ansatz von der verwendeten Programmiersprache unabhängig. Der wesentlichste Nachteil besteht allerdings in der Austauschbarkeit der Datenbanken. Da jede Datenbank ihr eigenes Remoteprotokoll implementiert muss dies natürlich berücksichtigt werden. Weiters ist es natürlich im Gegensatz zum vorherigen Ansatz ein Produkt bei dem man nur das Kompliat besitzt zu schützen.

6 Konklusio

Beide Ansätze haben mit Sicherheit ihre Vor- und Nachteile, wobei hier doch die Vorteile des ersten Ansatzes überwiegen. Die Möglichkeit seine Applikationen ohne Rekompilierung und ohne größeren Aufwand gegen SQL Injections schützen zu können überwiegt den eher geringen Nachteil der Sprachenabhängigkeit. Dennoch hat auch der zweite Ansatz seine Berechtigung, wobei hier die Nachteile doch schon wesentlich mehr ins Gewicht fallen.
Beide Techniken haben jedoch bewiesen, dass sie gegen SQL Injection Attacken resistent sind und noch dazu geringen Overhead aufweisen und sind somit sicherlich für den Einsatz geeignet.

Literatur

[BK04] Stephan W. Boyd and Angelos Keromytis. *SQLrand: Preventing SQL Injection Attacks*, pages 292–302. Springer Berlin / Heidelberg, 2004.

[HOM06] William G.J. Halfond, Alessandro Orso, and Panagiotis Manolios. Using positive tainting and syntax-aware evaluation to counter sql injection attacks. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 175–185, 2006.

[mil] www.milw0rm.com.