


After reading this chapter and completing the exercises, you will be able to:

- Describe a SQL injection and identify how injections are executed
- Identify how a Web application works and its role in SQL injections
- Define how to locate SQL vulnerabilities using error messages
- Apply inferential testing
- Review source code manually to locate injection vulnerabilities
- Describe methods for automatic traversing of source code to locate injection vulnerability



Security In Your World

Mazlo, an experienced security professional, was hired as a consultant by TJRigging, a large warehouse and storage facility in the Pittsburgh area. Clients of TJRigging have direct access to the company database through TJRigging's Web site. The Web site provides a convenient way for customers to check on their own stored product inventory.

Mazlo was told that his first task on the job was to conduct a Web server security audit for the TJRigging's Web site. From his home, and without any prior knowledge of the database, Mazlo attempted to obtain access to the company's database. He typed the company's URL into his browser and was immediately prompted for a username and password. Unaware of the authorized credentials, he decided to use a common SQL injection statement — `' or '1' = '1 --'` — placing this statement in the username and password fields. The site returned the following error message: "Your password can only contain numbers, letters, or the underscore." Based on this message alone, Mazlo was able to tell that the database was vulnerable. Next, he used his browser to view and modify the source code for the company's site. He removed the JavaScript error handling, changed the maximum length of the password, and saved the file on his local hard drive. This time, instead of typing the company's URL, he used the path to the file of his newly modified version kept on his local hard drive.

Voilà! He obtained access and with administrative privileges to the database as well!

This scenario is not an exaggeration. SQL injections are not that difficult to execute and they accounted for about 30 percent of the over 500,000 Web site intrusions last year. With the right amount of knowledge, or access to the Internet, any vulnerable site can be attacked within a matter of minutes. Protection and awareness is becoming more important each day!

Understanding SQL Injections

It only takes one brief Internet search to understand the ease with which SQL injections can be used for unauthorized access to company databases and Web sites. In fact, on popular vlogging sites, such as YouTube and Metacafe, you can find several tutorials less than two minutes long on using SQL injections to break into organizations. The information is extremely accessible to anyone who wants it, and if not appropriately prepared, an organization can be devastated by the results. SQL injections are the methods by which intruders use bits of SQL code and SQL queries to gain database access. SQL injections are extremely dangerous, as they create vulnerabilities and provide the means by which an intruder can obtain full administrator privileges to sensitive data such as usernames, passwords, Social Security numbers, credit card numbers, and general account information. Having access to execute all queries within a database gives an

intruder the power to also manipulate anything that is available to the database, including the operating system.

There are three common strategies for execution of SQL injections: single channel, multichannel, and observational. In a single-channel attack, an intruder uses only one channel for which to execute SQL injections and obtain the returned results. For example, using a Web application to input SQL injections in place of text to obtain immediate use of unauthorized data would be considered a single-channel attack. A multichannel attack involves the intruder using one avenue, such as with Web applications, to initiate the injection and a different one to obtain the results. In this scenario, an intruder could use a Web application to exploit a system and then change the Web script so that the results from a Web application query are sent to another account, such as the intruder's personal e-mail account. Intruders can also send SQL injections with no intention of receiving data from the database; these are known as inferential injections and they are used only to observe and learn from the returned behavior of the server for later attacks.

Injections and the Network Environment

7

Most SQL injections are performed through a Web application that acts as an interface to a back-end database. **Web applications** are programs that are available on a network and provide a way for users to interact with remote systems or databases. With the popularity of the Internet and the prominence of networks in today's society, the use of Web applications to access a database has been widely adapted within the database industry. In fact, most of the Web pages that exist on today's Internet can be considered a Web application of some type; in addition, well over half of the time that the average Internet browser spends online involves using these applications to access some type of sensitive data (e.g., e-mail messages, bank account information, credit card information). This is the reason that Web applications are the primary target for today's intruders. Common uses for Web applications online include e-mail access (e.g., Gmail, OWA), auctions (e.g., eBay, ubid), shopping (e.g., Amazon, Overstock), banking, bill paying, vlogging, blogging, and online gaming.

Let's take a look at the method for which data is retrieved and manipulated using Web applications beginning at the Web browser. The process is pretty straightforward and involves the following general steps:

1. A user accesses the specific Web site where the Web application resides by opening up a Web browser on their local machine and typing in the desired Web address (*www.amazon.com*). The Web site displays forms with fields and buttons that the user can use to input requests. The form resides on the Web server and is built using a combination of Hypertext Markup Language (HTML) and some type of scripting language (e.g., PHP, JSP, NET). HTML organizes and formats the form, while the scripting language makes it interactive by correlating some type of action with its components (buttons and fields).
2. A scripting language (PHP) residing on the Web server (*amazon.com*) reacts to the user's submission and passes the SQL statements to the company's application, or middleware server.
3. The middleware server contains applications that enforce business and database rules. These servers act as the interfaces to the database by managing the number of concurrent connections to the database as well as the Web content that is returned. The two primary responsibilities for middleware in a Web-based data retrieval environment are to initiate the database connection and to pass along the query.

4. Once received, the database executes the query and returns the results to the application server.
5. The application server returns the results to the scripting language on the Web server.
6. The scripting language, along with HTML, displays the results on the screen.

SQL injections are deployed in the very beginning of this process. There are two ways that SQL injections can cause destruction and create vulnerabilities. Lethal SQL code can be directly placed into user input fields and executed at the database, or ill-written code can be sent to be stored in the database. Either way, if Web applications are not prepared to detect and filter SQL injections, our most sensitive databases are at risk. The most common way to prepare an application to detect injections is to ensure that the application validates the data being received by the user before sending it to the database. SQL is used to communicate to Microsoft SQL Server, MySQL, and Oracle, so all three databases are at risk if applications are not properly secured.



It is important to note that SQL injections can exploit any client/server system that uses SQL Server and that relies on remote authentication. Although Web applications account for the majority of SQL injection intrusions, vulnerabilities also exist elsewhere.

In an environment where SQL statements are sent remotely through a Web application or client/server architecture, static, hard-coded SQL statements are not the optimal choice because query statements in this type of environment are reliant on users' input into the Web application form. A **dynamic SQL statement** is a SQL statement that is generated on the fly by an application (such as a Web application), using a string of characters derived from a user's input parameters into a form within the application itself. A **static SQL statement** is a statement that is built by the user and the full text of the statement is known at compilation. With dynamic SQL statements, developers build applications that handle most of the SQL code in real time, building the full query before being executed within the database because the full query is not known until the user inputs the information.

To illustrate the difference between a dynamic and static SQL statement, suppose that Smith is working on a Web application that displays input fields to users, who fill in certain criteria that the application uses to search the database. Let's say there are three fields, name (ClieNm), title (ClieTitl), and department (Depart). If Smith were to create this code using a static SQL statement, he would have to take into account all combinations of the various possible user inquiries because the statement depends on user input. This can be very cumbersome as possible parameters increase. For instance, imagine the number of potential combinations of possible inquiries search parameters are enabled on: client's name, address, phone number, department, title, age, starting date, status, sex, and race—and the use of AND or OR is allowed.

Dynamic SQL statements are best for Web database access, yet, as mentioned previously, if the input is not validated before being sent to the database, an attacker could input SQL statements, rather than search criteria into the input fields of the form. These statements will be interpreted as code and results will be generated based on the attacker's SQL syntax. Consider this scenario: A user types a URL into the browser to obtain access to a company's Web application and is prompted to provide a username and password. Rather than supplying a real username and password, the attacker can place SQL syntax into the fields to pull data from the database itself to be used for authentication. For example, the syntax `'or '1'='1--` often returns

the first entry in a given table. So if the attacker places `'or '1'='1--` into the username and `'or '1'='1--` into the password fields of a vulnerable Web application, the Web application will create a SQL statement that essentially checks the table that holds the username and password information and uses the first username and password from this table as credentials for the attacker. There is always certainly at least one user in the database, so authentication is inevitable, and the attacker gains access with the privileges of the first user listed in the database. Furthermore, because usernames are often listed in this table in alphabetical order, the user account the attacker could potentially log in as is the *administrator* account.



TIP

Developers often write script within Web applications to handle a user error, such as if a user is missing data required for a field, an error message might state “Missing or invalid information.” These error messages can also give clues as to how an error is handled within a system. These messages, if descriptive enough, provide the attacker with just enough information to change the error-handling code to meet his or her needs. For example, some sites have restrictions as to what symbols can be used for a person’s username or password. Given the previous scenario, let’s assume that the attacker attempts to use the code `'or '1'='1--` to obtain access into a database in which equal signs are not valid username characters. If the developer has written a script that informs users of this error by displaying the message “Invalid password characters,” the attacker has just learned the error of her ways. Armed with this information, the attacker can review the scripting language by choosing to “view source” of the HTML page, and change the script by allowing the use of equal signs or removing error handling altogether. Therefore, developers should take caution when using error-handling messages.

7

Identifying Vulnerabilities

As an administrator, the primary step toward securing data is identifying vulnerabilities with the system. Without knowledge of the system weaknesses, security actions would be fruitless. One of the most effective ways to find vulnerabilities within an environment is to play the role of an intruder. Just as with any other area of security, having an understanding of the possible attacks that an environment might face is vital to the protection against SQL injections. Determining possible SQL injection attacks is not a simple task. There are several different types of attacks, different areas of the network can be attacked, and different methods can be used to deploy injections. Understanding these things from both a user and administrator perspective is very important to the protection of any database environment. This section explores these topics and explores ways to observe the behavior and code of the applications and systems to find SQL vulnerabilities. Much of the focus is placed on Web applications. As mentioned previously, Web applications are prominent in today’s networks and they currently provide the primary means for SQL injection intrusions.

Inferential Testing for Locating SQL Injections

This section looks for clues of SQL injections using behaviors that are returned from the database in response to a controlled attack.

Copyright 2011 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

To find the SQL injection vulnerabilities through server behavior responses, an administrator must approach testing from the database user perspective, inputting parameters from the client side of the database environment. Once inputs have been made, the database server behaviors can then be analyzed and abnormal responses documented as potential vulnerabilities. For this type of testing, the administrator must be able to recognize abnormal server response and be capable of distinguishing these responses from the more typical or expected ones. The first step to acquiring this skill is becoming very familiar with the way the application and Web browser behave during normal data retrieval. This includes a basic understanding of the applications, the scripting language, the way the browser typically sends statements to the application server, and normal database returns given a specific return. Basically, an administrator must first familiarize herself with the environment in its normal state.

Using HTTP

All network communication is based on the same basic principles. A request to obtain a resource is sent from a source machine or application (client) to a destination machine or application (server). The request is received and processed and the client's privilege is checked to determine allowable permissions for the requested resource. Once approved, the requested resource is packaged and sent from the server to the client. Each step is handled using a set of standards or protocols that determine the who, what, when, where, and how of the communication process. For example, Transmission Control Protocol or TCP defines a set of rules that ensures a reliable virtual connection for delivery of data from one machine to another, and Hypertext Transfer Protocol (HTTP) is a set of rules that defines the method by which hypertext requests and responses (Web requests and responses) are formatted and packaged.

HTTP plays an important role when using Web applications for access into a database. When a request is made from a Web application to a database server, HTTP submits these requests by first initiating a TCP connection as the transfer agent. HTTP includes eight predefined actions that can be performed on a resource during the request or a response from a server: HEAD, GET, POST, PUT, DELETE, TRACE, OPTIONS, and CONNECT. Each of these define some way a request or response of a client or server should be handled. Although administrators should become familiar with all of these actions, only two are relevant to this discussion of SQL injections, as they deal primarily with network forms. These are GET and POST.

GET requests are encoded by the browser into a URL and the server will execute whatever parameters are appended to the URL itself. In other words, once a user fills out a form and clicks the submit button, the information that the user inputted into the fields of the form, which in this case is a query, is attached to the URL and sent to the server. The Web server will look for queries found within the URL and send them on to the database.

If using GET for Web forms, the information that the user has inputted into the fields of the form is included within the body of the request, rather than the URL. In this case, the application server recognizes the POST action and searches within the body of the request for particular statements to send on to the database to execute.

So, let's review this process. A Web application is presented to a database user through their local Web browser. This application includes some type of form, which the user must fill out to initiate a query and which sends a request to the Web server. The information that is provided within the form (user input) is sent either by HTTP GET or by HTTP POST to the Web server, where the scripting language retrieves the information and dynamically builds

the SQL statement to be forwarded to the application server or middleware. The middleware applies business logic (depending on the type of middleware) and based on restrictions within the application server, it forwards the SQL statement to the database to be executed. The database server executes the query and returns the results to the application server, which in turn sends it back to the Web server. The Web server displays the results using HTML to format the view for the users.

This is an important concept to understand because user requests can be intercepted during transit from Web application to the Web server and user data can be switched with SQL injections. The user (if unauthorized) can also place SQL statements into the fields of the form instead of normal text. In both of these cases, the user input is switched or inaccurate prior to reaching the scripting language, so without filters and protection, the scripting language dynamically creates corrupt SQL statements to send on to the application server. An administrator searching for these vulnerabilities requires a close analysis of the data throughout the process, from the user browser to the database. Intercepting the data as an intruder would help provide a clear view as to what is being sent and received at the Web server.

Intruders can intercept and even modify GET and POST parameters in a few different ways. Administrators should use the following techniques to test their own environment and observe what is found when data is intercepted:

7

- *Third-party applications*—Several applications are accessible for free download on the Internet to intercept and manipulate HTTP data. Some of these applications are specific for either GET or POST, but most can parse both GET and POST information from a data transfer session.
- *Browser add-ons and plug-ins*—Because the data is sent directly from the Web browser, intercepting and modifying GET and POST requires browser-specific applications. Easier than third-party applications, add-ons have been created to provide browser-specific interception. These add-ons are also available online and can often be found on the individual browser support page. For example, a well-known add-on named TamperData is available for Mozilla Firefox and can be downloaded from Mozilla's browser add-on page.
- *Proxy servers*—Proxy servers are computers or applications that are used to monitor and mediate internal and external communication. Proxy servers are the “gatekeepers” of the network; they monitor all incoming and outgoing requests and use an administrator-defined set of criteria to filter traffic that is coming into and leaving a specific network or area of a network. As part of this process, proxy servers intercept client requests and analyze these requests based on a given set of rules. Therefore, a proxy server enables the interceptions and modification of HTTP requests, including GET and POST parameters.
- *Using one of the preceding tools*—An administrator can intercept and observe a network's GET and POST parameters prior to the SQL statement build to identify any abnormal requests.

Determining Vulnerability Through Errors

As mentioned a few times throughout this chapter, finding data vulnerabilities requires administrators to gain a strong understanding of the overall database request and response process. Although malicious SQL code can be injected at several points of this process, it is

not executed until it reaches the database. The Web server does not test the data for errors or inappropriate database responses. Middleware could be designed to apply some logic to the statement, but in general, once injections are executed by the database, nothing gets in the way of returning the unauthorized results to users. With this said, administrators testing an environment for potential vulnerabilities may be presented with a few errors along the way. Although the errors come directly from the database, the scripting language determines how to present the errors and what to state within an error, and errors will not always occur because of SQL injection. It takes a strong familiarity with the errors that a system returns as well as knowledge of how these errors are presented to users to find vulnerabilities. If not filtered correctly, an error can give potential intruders information to help in the attack. In addition, as an administrator is testing the environment, the error messages will give clues as to what input has caused the issue, providing more information about potential SQL injection vulnerabilities.

Errors can be handled by application developers in different ways. The application's scripting language can be coded to display specific error messages in response to a user error. These messages, such as "usernames and password cannot include numbers," can give intruders information about the system that they can use to find a way to break in to the system. For example, the message "usernames and password cannot include numbers" tells an intruder that usernames and passwords do not include numbers and saves an intruder a great deal of time in his efforts to obtain authorized credentials for a particular system. Web applications can also be configured to respond to user error with generic messages. This approach would include errors such as "Please check your username/password," and is the best way to minimize the opportunity for intruder inference because these types of error messages do not typically give the user any information about the system. Administrators can also choose not to handle errors at the Web application, allowing HTTP to handle the errors instead (generating "Server Error" messages). This can help to minimize intrusions as well, yet, as we discussed previously, this technique is not without faults. Inferential testing conducted on error messages offers administrators and security professionals great insight into their own systems. Using inferential techniques, professionals can take note of the location in which error messages are being handled as well as the content of the messages that are being displayed and use this information to minimize potential vulnerabilities.

This section explores the different types of errors that can occur and should be monitored as events within a system. It is intended to help administrators and security professionals learn ways to distinguish between general errors and those that indicate vulnerability.

When testing for SQL injection vulnerability by observing error messages, you can organize information in four categories. These categories are typical conditions with no error, typical conditions with typical error, injection conditions with no error, and injection conditions with injection-caused error.

Typical Conditions with No Error

To determine vulnerability of a system using error messages received, you must first understand the typical error-free condition. This section displays a typical or baseline condition in which data is processed successfully and no error messages are present.

Imagine a scenario in which a local online specialty grocer allows customers to search and purchase their products on their Web site www.yum.com. This site allows patrons to choose

from different types of foods, such as dairy, produce, and meat. Each category has its own respective button on a Web form; when visitors click a button, a new page appears displaying a list of available products for that food category. Dairy's button may link to the following URL:

http://www.yum.com/index.asp?category=dairy

For example, imagine that the Web site contains a button titled "Click here to view all dairy products." This button, when pressed by users, will result in the preceding URL. The scripting language will receive the output URL from the button and send a statement to the database requesting for all items whose category=dairy. In this case, the scripting language is ASP and so the request sent to the database will look something like this:

```
food_cat = request("category")
sqlstr= "SELECT * FROM products WHERE Food_Category = '&food_cat&'"
set rs=conn.execute(sqlstr)
```

In this case, the ASP would create the following SQL statement to be executed by the database:

```
SELECT * FROM products WHERE Food_Category = 'Dairy'
```

In this case, the database would return one or more rows that match the WHERE clause, which in this case is Dairy. No errors are generated for this scenario. Keep in mind that when using SQL, alphanumeric values must be enclosed using single quotes (e.g., 'Dairy').

7

Typical Conditions with Typical Error

Even under the most typical data-processing conditions, errors can occur. Administrators and security professionals should be very familiar with the error messages that are produced primarily as a result of a common user error. These types of errors are often not viewed as potential threats or vulnerabilities to a system, so they are often overlooked, but unfortunately, any error that is produced within a database environment can be an indication of system vulnerability. Therefore, these messages should be carefully monitored and considered over time.

Let's assume that through inferential testing, an administrator changes the URL manually to state:

http://www.yum.com/index.asp?category=Hungry

In this case, the ASP would create the following SQL statement to be executed by the database:

```
SELECT * FROM products WHERE Food_Category = 'Hungry'
```

Knowing that the value of *Food_Category* determines what should be displayed to the user, errors will be generated with this scenario if *Hungry* is not a category of food within the system. These errors might or might not indicate vulnerability, as mentioned previously, as it could be returned due to user error or due to an intruder's attempt to obtain information from the database. For this scenario, the database would return an error informing the user that the column does not exist in the products table. Although this is not necessarily an indication of injection vulnerability, it displays that URLs can be changed manually if database information is not hidden.

Injection Conditions with No Error

SQL injections executed without an error returned from the database are known as a successful injection. These are often caused by unprepared Web applications that do not filter user input and URLs that do not hide database information. When testing inferentially for vulnerabilities, injections that are successful need immediate attention.

Below is a common SQL injection technique, which when not filtered correctly, does not generate an error. The statement, `'or '1'='1` is used often in SQL injections. It always returns *true*, which can prove to be very useful when attempting to gain information about an environment. This statement's counterpart is `'or '1'='2`, which always returns *false* and can be very useful in injections and testing as well.

Several variations of this statement accommodate the different types of databases and code structures that exist in the networking environment. For example, an equivalent true statement for Microsoft SQL Server is `'or 'ab'='a'+ 'b`, for MySQL it is `'or 'ab'='a' 'b`, and in Oracle the statement is `'or 'ab'='a'\\ 'b`. Attacks using statements similar to these examples are most often used in *blind injections*. Blind injections are attacks made with little to no knowledge of the system. Similar to throwing a dart at a dartboard while blindfolded, sometimes blind injections will be successful and other times they will miss the target altogether. A series of true and false SQL statements is used to attempt to discover information from a system. Blind attacks can be very difficult to detect due to their subtle nature.

Assume that the Yum URL can be manually changed to state:

```
http://www.yum.com/index.asp?category=dairy 'or '1'='1 --
```

This would result in the SQL statement:

```
SELECT * FROM products WHERE Food_Category = 'Dairy' 'or '1'='1 --
```

In this case, the database (if not filtered) returns everything from the products table even if Food_Category is not equal to *Dairy*. This is because, as mentioned previously, the statement `1=1` is always true. The statement `'or '1'='1 --` can be quite dangerous and can be used to provide access to an unauthorized user (by placing it in a password field) and help an intruder retrieve all of the data from a database (by being injected into a statement while it is dynamically created). This statement works effectively, causing no errors in return. The single quote in the beginning of the statement is the correct syntax to complete a statement, avoiding the return of errors, and adding double dashes at the end starts a comment in SQL, so all values after the injection are ignored by the system.

Single quote characters can be extremely helpful during inferential testing. They can provide a great resource for detecting injections and exploitations. When testing a database response, single quote characters are inserted in different places to determine vulnerability. If a database reacts the same way with single quotes added to the SQL statement through manual manipulation, as it reacted without changes being made to the statement, this is a great indication that vulnerability exists. To test this, the specific database SQL language needs to be considered, as syntax rules will be different. For example, in Microsoft SQL Server and MySQL, a manual change of the variable value using single quotes, such as in changing `category=dairy` to `category=da' 'iry`, should result in an error. Because Oracle uses PL/SQL, the manual change required to test this would be similar to changing `category=dairy` to `category=da' + 'iry`. If the type of language is not considered, the error generated would be caused from true syntax errors and a false sense of security would exist.

Injection Conditions with Injection-Caused Error

Depending on the database being used, some error messages will provide clear indications that a vulnerability has occurred. To effectively secure an environment, administrators need to rigorously test their environments to become aware of those error messages that indicate that a vulnerability does exist. These messages will differ from one type of database to another and will depend on how the environment's application is built to handle errors, so there is no definitive list that can be provided. Obtaining a strong understanding of one's own environment's behavior will help an administrator and security professional develop a list of their own. This scenario provides one example of how an injection can cause an error. The difference between this and the examples previously given is that the syntax being used is not appropriate for the database being queried and Web applications will not typically and dynamically create statements that are using incorrect syntax. This scenario uses `'or '1'='1`, which is a variation of the statement in the previous example.

Assume that the Yum URL can be manually changed to state:

```
http://www.yum.com/index.asp?category=dairy "or "1"="1 --
```

```
SELECT * FROM products WHERE Food_Category = 'Dairy' "or "1"="1 --
```

7

Here an error would be returned. The database would claim a syntax error because of the double quotes that have been appended to the SQL statement from the SQL injection. This error is also often a result of blind injections, as the attacker is attempting to determine the dialect of SQL (e.g., PL/SQL versus T-SQL), the type of database (e.g., MySQL versus Oracle), and the error handling of the Web application. With blind injections, the attacker does not have knowledge of the system, and because SQL comes in a number of different varieties, the database or SQL dialect must be determined before an injection can take place. An attacker will often use trial and error to determine these things.

Generic Error Messages

As mentioned previously, error messages can be displayed in many different ways and by many different means. Application developers typically make the determination as to how, and what, an error message will display by creating code within the Web application itself. In some cases, applications will show generic error messages with no reference to the type of event that returned the error, and on some occasions, no message will be presented at all. This poses a challenge for intruders and database administrators alike because it is difficult to determine whether the error has occurred within the database or exists within the application itself. It is possible that the application is not built to handle errors, that there isn't code created to display a message or redirect a user to a page developed for error handling. A number of generic errors returned by a Web application during testing can mislead administrators by giving the perception that the system is extremely secure, while in reality the Web application could be causing the error. To rule out an error within the application error-handling system, an administrator must use a SQL statement that does not initiate an application error, but tests the database error returns instead. This can be done by inserting controlled successful SQL statements into parameters, such as the appropriate true statement for that particular database. For example, using `'or '1'='1` as input parameters into a field would not cause an application error but would test the database for SQL injection vulnerability.

Direct Testing

Using inferential testing, you can uncover a great deal of information from the behavior of the database. Potential weaknesses are identified through system behavior observation and at this point, theories about vulnerabilities have been made based on abnormal behavior that was identified. Before the exploitation or removal of the injections can take place, further investigation is necessary. The next goal is to actively execute potential SQL injections, testing theories that were made during inferential testing. Doing this requires an understanding of the whole process as it is translated into programming language. Thinking like a program developer, you need to obtain a clear vision of the transfer of data from the Web application to the scripting language, to the middleware, and on to the database. In the mind-set of an intruder, SQL statements are built to actively test the environment. Determine just how far an outsider is able to reach into the system, bypassing authentication, and actively manipulating the data. Find out to what extent unauthorized access can be obtained and how much data is available for view. Active testing prepares an administrator for the removal of the injection, narrowing down the vulnerabilities and adding a focus to the exploitation efforts.

Using the Code for Locating SQL Injections

Inferential testing is a great way to find SQL injection vulnerabilities, much like security penetration testing. It allows administrators to view the system security from an attacker's point of reference. The second most common approach to locating SQL injection is through source code analysis. This strategy requires less time and resources, but it might require the database administrator to work with the application developer. Much information can be gained from reviewing the source code. Administrators can ensure that dynamic statements are being created and filtered without SQL vulnerability as well as determine how and where user input is being accepted—both of which are important aspects of maintaining a safe environment. This section discusses these topics as well as provides tips on how to identify vulnerable code.

Source Code Analysis

Analyzing source code can be quite a tedious and painstaking task. Depending on the approach, reviewing the source code of a piece of software one line at a time can take months, yet identifying problems at the source code level is one of the most effective ways to manage SQL injection vulnerabilities. Several tools are available to help automate the process, but many of these focus primarily on security and lack strength in locating errors that SQL injections can exploit. Source code can be analyzed in different views—while the code is running and while it is not. Analyzing the code while it is running is known as conducting a *dynamic analysis*. Dynamic analysis is an attempt to find errors or vulnerabilities in the source code of a program dynamically while it is being executed, whereas *static analysis* is an effort to find problems while the program is inactive. Static code analysis requires less resources and expense and is much more effective at identifying vulnerabilities within the code.

To effectively sanitize source code and rid it of its SQL vulnerabilities, you must first be able to identify potential problem areas. Problem areas can be poorly written functions or user input that is not verified. As shown earlier in the chapter, SQL injections can be input as parameters within fields on a form. Vulnerable sites are often those that do not validate a user's input by ensuring that it meets a certain predefined criteria prior to being included within the dynamic build of the SQL statement. For example, as shown in the scenarios presented earlier in the

chapter, many of the successful SQL injections used a single quote as the starting character as a way to append to the existing statement. For the malicious input to be included within the dynamic statement, it has to be added without any type of filtering process. Including filtering processes between the user input and the dynamically created statement that define certain limitations for a user's input can easily reduce the number of SQL injection attacks.

Identifying these issues can be quite a large task, one for which the time requirements depend on the size of the program and the code's overall logic structure. You must first identify where in the code each SQL statement is built. It is here that the name of the variables that hold user input can be identified. Working backward in the code, the goal is to follow the path of the variable back to its origin: the place where the user input enters the code. Traversing through the code, an administrator must identify if restrictions have been placed on the user input anywhere in the code. Consider the following scenario.

The user inputs some type of information into a text field of a Web application online. The text field has a name assigned to it, such as *TFName*. A variable in the scripting language of the Web application is defined, named, and assigned to *TFName* (e.g., *UserInput* = "TFName"). The variable *UserInput* transfers the content of the text field (*TFName*) to a function (predefined and based on the language being used) that dynamically creates the SQL statement. If at no time the variable (*UserInput*) is verified to have an appropriate set or type of characters, anything can be inserted into the text field and transferred directly into the SQL statement. This is the reason that it is important to trace the path of all variables that contain user input, looking to find any verification that the user's input fits a certain set of defined criteria for that text field.

Keep in mind that this example is only considering the transfer of the data within the scripting language of the Web application. It is equally as important to consider how the data is being transferred from the form into the scripting language. As mentioned earlier in the chapter, this transfer involves the HTTP actions GET and POST. The actions of HTTP are identified in the HTML code, so the HTML code should also be reviewed to ensure the desired settings. How SQL statements are constructed is specific to the scripting language being used, as the functions are predefined. Prior to using functions to construct SQL statements, research the function thoroughly to ensure that validation is being applied.

Tools for Searching Source Code

Throughout this chapter, different strategies for identifying and testing SQL injections have been explored. Many of these strategies are time consuming and resource intensive, yet none are as time consuming as traversing line by line through hundreds of pages of source code. Several tools are available to help facilitate this process, yet none are as reliable as manual searching. There are essentially three methods for analyzing static source code: string-based pattern matching, lexical token matching, and data flow analysis. This section explores these techniques in terms of their reliability and methodology.

String-Based Matching

String-based matching is a simple detection tool that searches for, and locates, user-defined strings and patterns found within source code. It is the most basic of the three strategies and produces the highest number of false results. Signatures are created for typical SQL injection variables, and other types of dangerous source code and string-based matching systems attempt to find strings or patterns that match these signatures.

Data Flow Analysis

Data flow analysis is a method for obtaining information about the way variables are used and defined in a program. It determines the dynamic behavior of a program by examining its static code. Source code is divided into blocks of data for which data flow analysis tries to collect information at each point in a procedure. Data flow analysis uses control flow graphs to display how events in a program are sequenced during execution. Data flow analyzers help database administrators obtain a map of each variable as well as the assigned value each step through the program.

Lexical Analysis

Lexical scanning is the process by which the source code is read from left to right and then grouped into tokens, based on some type of similar criteria. A lexical analyzer can identify common symbols that the initiating programming language defines. Some tools implement an approach known as lexical analysis.

If used correctly, tools that utilize one of these methods to parse source code can provide a great deal of help for detecting dangerous code and potentially risky variables. Yet, this is not an exact science and no one automatic source code parsing tool should be relied on solely to provide protection against inaccurate code. Developers and database administrators ideally would pair up and choose a tool with the method that they desire the most. The tool in combination with an expert can provide an effective and reliable tool for analyzing source code.

Chapter Summary

- SQL injections can create vulnerability and provide the means by which intruders can obtain full administration of a database.
- SQL injections can exploit any client/server environment that uses an application to obtain remote connection of a database.
- There are two types of SQL statements, static and dynamic. Dynamic SQL statements are created by applications in an online environment, whereas static SQL statements are used when accessing the database directly.
- Administrators can test for SQL injections either by sending a series of requests while observing the reaction of the server or by reviewing the source code.
- HTTP and TCP are two of the several protocols used for sending information over the network. When a request is made from a Web application to the database, HTTP submits these requests by first initiating a TCP connection as the transfer agent.
- HTTP data can be intercepted and manipulated using third-party applications, local proxy servers, and browser add-ons and plug-ins.
- Becoming familiar with typical database errors is important for identifying the anomalies and vulnerabilities within a system and its environment.
- It is important to keep in mind that applications and databases handle errors differently. An error that may be returned on one database might not be returned on another under the same circumstances.

- Testing for SQL injections should be included within an environment's security penetration testing process.
- Source code can be analyzed while the program is static, or dynamically as the program is being executed.
- Single quotes are an important aspect to error testing; they can be used to identify vulnerabilities.
- Several tools can automatically analyze source code, yet they are not as accurate as a manual user analysis.
- String-based matching tools review source code looking for patterns and strings of data that match vulnerable and dangerous code. This is the most basic form of automatic source analysis tools.
- Data flow analysis tools use control flow graphs to display the sequencing of procedures and variables found within source code. This helps administrators find and verify the safety of the input of variables.
- Lexical analysis scanning tools group code into tokens based on similar criteria, allowing administrators to find dangerous code through tokens.
- Automatic source code analysis should be used by security professionals and in conjunction with manual source reviews.

Key Terms

dynamic analysis An attempt to find errors or vulnerabilities in the source code of a program dynamically while it is being executed.

dynamic SQL statement A SQL statement that is generated on the fly by an application (such as a Web application), using a string of characters derived from a user's input parameters into a form within the application itself.

SQL injections Methods by which intruders use bits of SQL code and SQL queries to gain database access.

static analysis An effort to find problems while the program is inactive.

static SQL statement A statement that is built by the user; the full text of the statement is known at compilation.

Web applications Programs that are available on a network and provide a way for users to interact with remote systems or databases.

Review Questions

1. Discuss the three common strategies for execution of SQL injections.
2. Discuss the relationship between a Web application and a scripting language.
3. Describe the steps involved in the user input reaching the database.
4. Discuss the challenges of typical conditions that create typical errors.

5. Describe the difference between dynamic and static SQL statements. Provide an example to support your discussion.
6. Explain inferential testing.
7. Explain the difference between the HTTP actions GET and POST. Which is more vulnerable to SQL injections? Provide an example to support your statement.
8. Identify and explain at least one way that HTTP data can be intercepted and changed.
9. Explain how studying error messages can help identify SQL injection vulnerability.
10. Explain why active testing is just as important as inferential testing when searching for SQL injection vulnerability.
11. Identify at least one method that source code analysis tools use to locate dangerous source code.
12. Identify how the common injection `'or ' 1 ' = ' 1` can be used to test for SQL vulnerability.

Case Projects



Case Project 7-1: Web Applications

Search the Internet and find three Web applications. Write a paper that discusses the purpose of a Web application; include the three that you found as support for your discussion.

Case Project 7-2: Source Code Analysis Tools

Search the Internet for at least two static source code analysis tools. Identify which method described in the chapter these tools use for source analysis.

Case Project 7-3: The Data Retrieval Process

Describe the steps involved from the user input to the database results displayed on the screen. Identify the different points of the process described in Review Question 3 in which SQL injections can occur.

Case Project 7-4: SQL Server SQL Injections

Identify at least two T-SQL injection warnings or tips found on the Microsoft Web site or elsewhere online that are specific to Microsoft SQL Server.

Case Project 7-5: Oracle SQL Injections

Identify at least two PL/SQL injection warnings or tips found on the Oracle Web site or elsewhere online that are specific to Oracle Database.

Case Project 7-6: MySQL SQL Injections

Identify at least two SQL injection warnings or tips found on the MySQL Web site or elsewhere online that are specific to MySQL.

Hands-On Projects



Hands-On Project 7-1: Securing the Oracle Environment

You have been hired as the security consultant for TJRiggings. As your first task, you have been asked to complete a security audit of their Web applications. Answer the following questions:

1. What are the first steps that you would take to test the sites for SQL injection vulnerability?
2. How might you apply the concept of inferential testing?
3. What is your strategy for identifying dangerous source code now and far into the future?
4. What suggestions would you offer TJRiggings in reference to their Web clients?