Automating `mysql_secure_installation`

bertvv.github.io/notes-to-self/2015/11/16/automating-mysql_secure_installation/

Many HOWTO's and blog posts about installing MySQL/MariaDB, a LAMP stack, etc. suggest to run the script `mysql_secure_installation` to tighten the security holes in the default installation of the database engine. This includes setting a root password (empty by default), removing anonymous users, and deleting a test database. For a database server that you're going to run in production, it is really important to do this. However, I have a problem with the fact that `mysql_secure_installation` is interactive, i.e. it asks for user input. This makes it very hard to include it in an automated setup. In this post, we'll discuss how the script works and how we can automate what it does.

One approach to automating `mysql_secure_installation` consists of using the `expect(1)` tool that can "sense" when the user is asked for input and can inject the appropriate response. This is a kludge and should be avoided. Expect is written in the scripting language Tcl/Tk, which is typically not installed by default. So installing `expect(1)` also pulls in dependencies that have no added value to the functionality of the system itself. Installing unneccesary software is a bad security practice. Expect also adds an unnecessary layer of complexity to a script that is already too complicated to begin with. If the `mysql_secure_installation` script was written in a way that was more friendly towards automation, it wouldn't be needed at all.

A better approach is to have a look at what the script actually does and then automating that. The MariaDB source code is hosted on Github and the `mysql_secure_installation` script can be found here. The script consists of lots of code to inform the user and interpret user input which are cruft for our purposes. The gems are the SQL statements scattered throughout the code. Look for the string `do_query` to find them. The queries will perform the following actions:

1. Set a password for the database "root" user (different from the Linux root user!), which is blank by default;
2. Delete "anonymous" users, i.e. users with the empty string as user name;
3. Ensure the root user can not log in remotely;
4. Remove the database named "test";
5. Flush the privileges tables, i.e. ensure that the changes to user access applied in the previous steps are committed immediately.

There is only one input that should come from the system administrator and that is the database root password. The other steps can be automated without any interaction with the user.

In the sections below, these steps are explained in more detail, and after that, I'll show how to automate them.

Setting the database root password

There's actually (at least) two ways of doing this: using the SQL query (like the `mysql_secure_installation` script does), or by using the `mysqladmin` command. The SQL statement can e.g. be executed within the `mysql(1)` client tool that can be accessed with the command:

```
$ mysql -u root -p
```

The option `-u` specifies the user (`root`), the option `-p` prompts for a password. You can just press ENTER now, but later, when you have set the root password, you will of course need to type it.

The SQL command to set the root password, copied from the script source, is:

```
UPDATE mysql.user SET Password=PASSWORD('my_new_password') WHERE User='root';
```

Replace `my_new_password` with the desired password.

The alternative method is the `mysqladmin(1)` command:

```
$ mysqladmin password "my_new_password"
```

Delete anonymous users

In the default installation includes anonymous users, apparently for testing. These should be removed before putting the database server into production, as it allows to access the database without a password. Anonymous users have an empty user name and can be deleted with the following SQL statement (from `mysql_secure_installation`):

```
DELETE FROM mysql.user WHERE User='';
```

Ensure the root user can not log in remotely

It's always a good idea to limit access for the database root user to localhost for added security. This won't even affect a setup with phpMyAdmin if it's installed on the same host!

```
DELETE FROM mysql.user WHERE User='root' AND Host NOT IN ('localhost', '127.0.0.1',
'::1');
```

Remove the test database

The default installation contains a database with the name `test` (with associated user privileges). These should also be deleted before putting the database into production:

```
DROP DATABASE test;
DELETE FROM mysql.db WHERE Db='test' OR Db='test\_%';
```

The first statement removes the database, the second privileges on the database.

Flush the privileges tables

The statements above that change user access to the database have no immediate effect. The following command will commit the changes:

```
FLUSH PRIVILEGES;
```

Without flushing the privilege tables, the root password will remain unset.

Putting it all together

We can execute all the SQL statements above with the following code snippet:

```
myql --user=root <<_EOF_
UPDATE mysql.user SET Password=PASSWORD('${db_root_password}') WHERE User='root';
DELETE FROM mysql.user WHERE User='';
DELETE FROM mysql.user WHERE User='root' AND Host NOT IN ('localhost', '127.0.0.1',
'::1');
DROP DATABASE IF EXISTS test;
DELETE FROM mysql.db WHERE Db='test' OR Db='test\\_%';
FLUSH PRIVILEGES;
_EOF_
```

The SQL commands are passed to `mysql(1)` in a here document. The variable '${db_root_password}' needs to be set. I've written a more elaborate shell script that takes the value of the database root password from the command line and has some checks in place. The full source can be found below:

```
#! /bin/sh
#
# Author: Bert Van Vreckem <bert.vanvreckem@gmail.com>
#
# A non-interactive replacement for mysql_secure_installation
#
# Tested on CentOS 6, CentOS 7, Ubuntu 12.04 LTS (Precise Pangolin), Ubuntu
# 14.04 LTS (Trusty Tahr).

set -o errexit # abort on nonzero exitstatus
set -o nounset # abort on unbound variable

#{{{ Functions

usage() {
cat << _EOF_

Usage: ${0} "ROOT PASSWORD"

  with "ROOT PASSWORD" the desired password for the database root user.

Use quotes if your password contains spaces or other special characters.
_EOF_
}

# Predicate that returns exit status 0 if the database root password
# is set, a nonzero exit status otherwise.
is_mysql_root_password_set() {
  ! mysqladmin --user=root status > /dev/null 2>&1
}

# Predicate that returns exit status 0 if the mysql(1) command is available,
# nonzero exit status otherwise.
is_mysql_command_available() {
  which mysql > /dev/null 2>&1
}

#}}}
#{{{ Command line parsing

if [ "$#" -ne "1" ]; then
  echo "Expected 1 argument, got $#" >&2
  usage
  exit 2
fi

#}}}
#{{{ Variables
db_root_password="${1}"
#}}}

# Script proper

if ! is_mysql_command_available; then
  echo "The MySQL/MariaDB client mysql(1) is not installed."
  exit 1
fi

if is_mysql_root_password_set; then
  echo "Database root password already set"
  exit 0
fi

mysql --user=root <<_EOF_
  UPDATE mysql.user SET Password=PASSWORD('${db_root_password}') WHERE User='root';
  DELETE FROM mysql.user WHERE User='';
  DELETE FROM mysql.user WHERE User='root' AND Host NOT IN ('localhost', '127.0.0.1',
'::1');
  DROP DATABASE IF EXISTS test;
  DELETE FROM mysql.db WHERE Db='test' OR Db='test\\_%';
  FLUSH PRIVILEGES;
_EOF_
```

I also put the script on Github, in my scripts repository. If I modify/improve the script in the future, you can find the newer versions there.

Conclusion

The `mysql_secure_installation` script makes the installation of MariaDB/MySQL more secure and the actions that it performs *should always* be applied to database servers that you put into production. However, by making the script interactive, it has become unnecessarily complicated. Worse, it's an obstruction in our quest to "Automate All the Things™".

I'm certainly not the first one to suggest a non-interactive script that does the same as `mysql_secure_installation`, but I haven't seen this approach actually being encouraged. I hope this post at least adds to the discussion.