Security

One of the most important aspects of both database administration and application design is also one of the most overlooked: security. While there are dozens of definitions of security that can apply to computer software, we can sum up our needs simply. Security is the ability to allow authorized users to access data while preventing access from unauthorized users. As simple as this definition is, it provides flexibility through the terms ‘users’ and ‘access’. Different kinds of security allow different kinds of users different kinds of access.

In this chapter we will examine the three main areas where security is important with regards to MySQL: MySQL data security, server security and client security. A lack of foresight in any of these areas can open up valuable data to attackers. However, with a little preventative care, your MySQL server can safely provide data in any environment.

MySQL data security

MySQL provides several mechanisms to control access to the data of the system. Looking back at the definition of security above, we can define MySQL data security be specifying meaning of ‘user’ and ‘access’ in this context.

A ‘user’, to MySQL, is an authenticated connection to the MySQL server. Any time a program attempts to connect to a MySQL server it must provide credentials that identify the user. Those credentials then define the users to MySQL for that connection.

To a MySQL server, ‘access’ is simply access to the functions the server provides. While this usually means access to the data in the server, via SQL queries, it can also mean access to administrative functions, such as setting access-rights for other users and shutting down or reloading the server.

Protecting MySQL data is the major job of the MySQL security system. The ‘data’ in this context can have two seperate meanings: actual data stored in the database, and information about that data (also called meta-data).

An example of actual database data would be any information stored within an actual database table. For example, consider a database named ‘mydb’ that had a table ‘People’ with the columns ‘firstName’ and ‘lastName’. This table has two rows with the data

'John'/'Doe' and 'Mary'/'Smith'. The actual database data in this example is 'John', 'Doe', 'Mary' and 'Smith'. This is the content that is the reason the database exists.

Meta-data, (or data about data), is the information that describes the structure of the database content. In the example above, the fact that there is one database is a piece of meta-data. That the database is named 'mydb' is also meta-data. Other meta-data includes the fact that there is one table, named 'People', containing two columns named 'firstName' and 'lastName'. The fact that there are currently two rows of data in the table is also meta-data. At first thought it might not seem like that should be meta-data since it is directly dependant on the actual data content. However, even though it is dependant on the data content, it is not actually the data content. Therefore it is meta-data.

The MySQL security schema protects both content data and meta-data. This is important because a system that protected only the data of the system would still be open for abuse. Consider an attacker that gained access to the meta-data in the above example. Simple changing the table name from 'People' to 'ConvictedCriminals' could make quite a change in the meaning of the data, even though the data has not been touched.

MySQL protects its data (and meta-data) through the use of special set of tables called 'grant tables'. These grant tables reside in the the 'mysql' database, which is a special system database that exists on every MySQL server. MySQL provides two ways of interacting with the grant tables: directly and through SQL queries.

We will cover using SQL queries to interact with the grant tables first, and then move into directly manipulating the tables. For many users the SQL interface will be all that is needed and the details of the underlying tables can be skimmed or skipped altogether. However, if you have every had any problems setting up MySQL security, of if you have complex security needs, interfacing directly with the grant tables may be necessary.

SQL Interface

Standard ANSI SQL provides two statements specifically designed for managing access to the database server: GRANT and REVOKE. MySQL supports both of these statements, with several semantic extensions that allow control over the meta-data of the system as well as the data itself.

The GRANT statement is used to provide a user access to functionality on the MySQL server. Conversely, the REVOKE statement removes access for a user. Both statements are executed the same way as other SQL statements such as SELECT, INSERT, etc.

GRANT

```
GRANT privilege [(columns)] [, privilege [(columns)] ...] ON table(s)
    TO user [IDENTIFIED BY 'password']
    [, user_name [IDENTIFIED BY 'password'] ...] [WITH GRANT OPTION]
```

The GRANT statement can be broken into three sections: what is being granted, where the grant takes effect, and who is being granted the privilege. This can be seen most

clearly by looking at the structure of GRANT when all of the optional attributes are remove:

```
GRANT privilege ON table(s) TO user
```

All other information given simply refines the ‘what’, ‘where’ and ‘who’ given in this simple format.

What

The type of privilege granted determines what abilities the user will have as a result of the GRANT statment. There are 15 privileges currently defined in MySQL:

ALL PRIVILEGES
: Despite its name, this does not grant all privileges to the user. It does grant complete control over the data, and databases, within the MySQL server. This privilege includes the following privilges: ALTER, CREATE, DELETE, DROP, INDEX, INSERT, REFERENCES, SELECT and UPDATE. It does not automatically grant FILE, PROCESS, RELOAD and SHUTDOWN. Those privileges effect the raw system state the MySQL server and must be granted explicitly. The privilege ‘ALL’ is provided as a synonym for ‘ALL PRIVILEGES’.

ALTER
: This provides the ability to alter the structure of an existing table. In particular it allows the user to execute the ALTER SQL statement for any purpose that does not involve table indexes (see INDEX, below).

CREATE
: This provides the ability to create new databases and/or tables. In particular it allows the user to execute the CREATE SQL statement.

DELETE
: This provides the ability to delete data from a table. Note that this does not grant the ability to delete the actual tables or databases (which are meta-data), only the data itself. Specifically, this allows the users to execute the DELETE SQL statement.

DROP
: This provides the ability to remove entire tables and/or databases. Conversely to DELETE, this does not provide the ability to remove specific elements of data from the tables, only to erase the entire table or database. This grants the user the ability to execute the DROP SQL statement.

FILE
: The allows the user to (directly or indirectly) read, write, modify or delete any operating system level file on the server with the same privileges as the MySQL server process. The purpose of this privilege is to allow users to use the LOAD DATA INFILE and SELECT INTO OUTFILE SQL statements to read and write server-side data files.

  However, as stated above, a side effect of this privilege is that the user is trusted with the same operating system level rights as the MySQL server for accessing files. See the ‘Server Security’ section below for tips on how to secure the server against abuse of this privilege.

INDEX
: This allows the user to create, alter and/or drop indexes on a table. This allows the user to execute the ALTER SQL statement only with regards to indexes.

INSERT
: This allows the user to insert data into a database table. In particular it allows the user to execute the INSERT SQL statement.

PROCESS
: This provides the user with the ability to view the list of MySQL process threads as well as the ability to kill any of the threads. MySQL process threads exist for each connection to the server. In addition, serveral utility threads exist to for overall server functionality. Therefore this priviliege should be careful granted as it can be used to arbitrarily terminate client connections or shutdown the entire MySQL server. This privilege allows the user to execute the SHOW PROCESSLIST and KILL SQL statements.

REFERENCES
: This privilege currently does not do anything. It is provided for SQL compatibility with servers such as Oracle that provide “foreign key” functionality.

RELOAD
: This provides the user with the ability to make the MySQL server reload information it keeps cached, such as privilege information, log files, table data, etc. In particular it allows the user to execute the FLUSH SQL statement.

SELECT
: This allows the user to read data from a database table. Specifically, it allows the user to execute the SELECT SQL statement.

SHUTDOWN
: This allows the user to completely shutdown the running MySQL server. This privilege should be granted carefully for obvious reasons.

UPDATE
: This allows the user to modify data within a database table. This does not grant the ability to add new data or remove old data, but only to change existing data. Specifically, it allows the user to execute the UPDATE SQL statement.

USAGE
: This provides the user with no privileges whatsoever. It can be used to create a user who can do nothing but connect to the server.

Multiple privileges can be specified in a single GRANT statement. In addition, there is one final privilege, GRANT, that cannot be specified with the rest of the privileges. Instead, the clause `'WITH GRANT OPTION'` must be included at the end of the GRANT statement. If this is done, any users given privileges in the statement will be able to re-GRANT those privileges to any other user.

This is a very powerful ability and should only be given to trusted users. Because of the nature of the MySQL grant system, the ability to grant new privileges is not limited to those granted initially. That is, if a user is given the GRANT ability during a GRANT statement, then the user is later given new privileges without specifying `'WITH GRANT`

OPTION', the user will still be able to re-GRANT those new privileges. Once a user has the ability to grant a privilege, they can grant any privilege they have at any time.

Examples

```
/* Note that these are incomplete SQL statements.
   They simply illustrate the format of the first
   section of the GRANT statement */
GRANT SELECT                  -- The user can execute SELECT statements
GRANT ALL PRIVILEGES          -- The user has all privileges except for
                              -- the system-wide privileges
GRANT INSERT, UPDATE, DELETE  -- The user can execute INSERT, UPDATE
                              -- or DELETE SQL queries
GRANT SHUTDOWN                -- The user can shutdown the server
GRANT CREATE, DROP ... WITH GRANT OPTION -- The user can execute CREATE
          -- and DROP statements. In addition, the user can execute
          -- the GRANT statement and re-GRANT the CREATE and DROP
          -- privileges, as well as any other privilege the user
          -- already has. Furthermore, if the user is given new privileges, they
          -- will automatically be able to re-GRANT those privileges as well.
```

Where

Some of the privileges discussed above apply only in very specific contexts. For instance, the SHUTDOWN privilege only has meaning when shutting down the entire MySQL server. However, most of the privileges can apply in a number of places. The CREATE privilege, for example, could apply to creating a new database or a new table. Privileges can apply to the entire server, specific databases, table and even individual columns within a table. Table XX-1 shows which privileges apply in which contexts.

Privilege	Column	Table	Database	Server
ALTER		X		
CREATE		X	X	
DELETE		X		
DROP		X	X	
GRANT		X	X	X
FILE				X
INDEX		X		
INSERT	X	X		
PROCESS				X
RELOAD				X

SELECT	X	X		
SHUTDOWN				X
UPDATE	X	X		

Every privilege granted must be granted at a specific location. For table, database and server-wide privileges, the location is specified in the ON clause of the GRANT statement. The ON clause can take several different location formats:

table_name
: This will grant the privilege for the given table name within the currently active database.

database.*
: This will grant the privilege for all tables within the given database

.
: This will grant the privilege for all tables within all databases. In other words, the privilege will be granted globally. This should be used when granting a server-level privilege such as SHUTDOWN.

*
: If there is a currently active database selected, this will grant the privilege on all tables within that database (the same as database.* using the active database). If no database is selected, this will grant the privilege globally (the same as *.*)

The one type of location that is not specified in the ON clause are column-level privileges. Column-level privileges should have the table (or tables, using the syntax above) specified in the ON clause as usual. The specific columns within the table(s) are specified after the individual privileges, in parenthesis, seperated by commas. Those privileges will this only take effect for those specific columns within the tables given in the ON clause.

Examples

```
/* As before, these are not complete GRANT statements, but only fragements
illustrating the
    portions covered so far *.
GRANT SHUTDOWN ON *.*   -- Grant SHUTDOWN globally. Since SHUTDOWN is a
                        -- server-wide privilege, this is the only context where it
                        -- makes sense.
GRANT SHUTDOWN ON * -- Same as above *if* no database is currently selected.
                        -- If a database is selected, this makes no sense as
                        -- SHUTDOWN cannot be granted for a database or it's tables.
GRANT CREATE ON mydb.*  -- Allow the user to create new tables within
                        -- the 'mydb' database.
GRANT CREATE ON *.* -- Allow the user to create new tables for any database.
                        -- In addition, allow the user to create new databases.
GRANT DROP ON mydb.* -- Allow the user to drop tables within the
                        --'mydb' database. This does not give the user permission
                        -- to drop the mydb database itself, the user can drop
                        -- every table within in, which is as effective.
GRANT DROP ON *.*   -- Allow the user to drop any table and/or any database
GRANT INSERT ON mydb.People -- Allow the user to insert new rows into
                        -- the mydb.People table.
```



```
GRANT SELECT (firstName, lastName) ON mydb.People -- Allow the user to execute
                    -- SELECT statements on the mydb.People as long as the
                    -- statements only read data from the 'firstName' and
                    -- 'lastName' columns.
GRANT SELECT (firstName, lastName, phone),
         INSERT (firstName, lastName), UPDATE on mydb.People
                    -- Allow the user to execute INSERT, UPDATE and SELECT
                    -- statement on the mydb.People table. The user can only
                    -- use SELECT when reading data from the firstName,
                    -- lastName or phone columns. The user can only use
                    -- INSERT to insert rows containing only firstName and
                    -- lastName data. The user can use UPDATE to change the
                    -- value of any columns in any of the rows of existing data.
```

Who

Now that we know what we are granted and where it will apply, the last step is to specify who we are granting the privileges to. This is specified by the TO clause of the grant statement. The format of the TO clause is a list of users, with optional passwords (given after the `'IDENTIFIED BY'` clause), separated by commas.

Unlike most other database servers, the format of the user within MySQL includes not only a username, but the user location as well. A valid MySQL username is any string of characters that is 16 characters or less. Any characters can be used (this allows usernames to be written in non-ASCII languages), however, standard ASCII characters are recommended as some clients cannot handle other character sets. However, if you know for sure your clients can handle the characters, there is no reason to stick to standard ASCII. If the user string contains any characters other than the standard ASCII alphanumerics it must be enclosed in quotes (either single or double quotes will do).

Note: When performing authentication, the MySQL server performs a case-insensitive check on the username. That means that if your username is defined as 'myuser', you can use 'MYUSER', 'MyUsEr', 'myuser' or any other permutation to log in. This also means that if you grant a privilege to 'myuser' and another privilege to 'MYUSER' you are really granting two privileges to the same user.

The location can be specified as a fully qualified domain name (my.server.com) or as IP addresses in standard dotted decimal notation (10.20.30.40). In addition, the SQL wildcards '%' and '_' can be used to specify a range of addresses. If a wildcard is used (or any character other than the standard ASCII alphanumerics), the location must be enclosed in quotes (single or double). In addition, if the numeric IP format is used, a netmask (also in dotted decimal notation) can be supplied after a '/' character. The netmask will be applied to any connecting user before checking it against the IP address.

For convenience, a username maybe provided without a location (and within the '@' symbol). This is equivalent to user@"%", which specified the username coming from any location.

Every user within the MySQL security system has a password. The password must be specified along with the username whenever the user connects to the MySQL server. When creating a new user using a GRANT statement, the password of the user can be specified by using the `'IDENTIFIED BY'` clause after the username/location in the TO

clause. The password can be any number of characters, and like the username can contain any characters. If the password contains anything except the standard ASCII alphanumerics (and a good password should), it must be enclosed in quotes (single or double).

If the GRANT statement is creating a new user, and no `IDENTIFIED BY` clause is provided, the user will be created with a blank password. This creates an immediate security hole and should never be done. If the GRANT statement is modifying the privileges of an existing user, an `IDENTIFIED BY` clause will change the password of that user, and no `IDENTIFIED BY` clause will leave the user's password unchanged.

Note: When MySQL is first installed it creates a couple of default users. The first is the 'root' user. This user is the only user that initially has the ability to GRANT other privileges. In fact, the 'root' user is given all privileges and can do anything to the server. In the default installation, the root user is enabled only for localhost and can not be accessed remotely. In addition, the default user is given a blank password. This should be changed immediately after installing MySQL as anyone (on the server machine) could take complete control of your MySQL server while root has a blank password.

Beyond the root user, MySQL also creates default permissions for any user connecting via localhost. By default, MySQL forbids everything for these users. They will be able to connect to the MySQL server, but not execute any operations. All other users (that is, any user connecting remotely) are not even allowed to connect to the MySQL server by default.

Examples

```
/* Now that we've seen all of the parts of the GRANT syntax,
   we can look at some complete GRANT statements... */
GRANT ALL ON *.* TO super@"%" -- Give all privileges
               -- (except for system-wide functions) to the user 'super',
               -- when connecting from any location. If 'super' did not
               -- already exist as a user, it will be created without a
               -- password. Consider that this user has been granted
               -- complete control over all of the data on the server
               -- this would not be a good idea.
GRANT SELECT, INSERT, UPDATE, DELETE ON mydb.People TO 'people_user'@localhost
               -- Give the ability to read, write, modify and delete
               -- information in the People table of the 'mydb' database to
               -- the user 'people_user', only if they are connecting from the
               -- same machine as the server. As in the previous examine, if
               -- 'people_user' did not already exist, it would be created
               -- with no password; a bad idea.
GRANT PROCESS, RELOAD, SHUTDOWN ON *.* TO admin@localhost IDENTIFIED BY 'pass'
               -- Give the ability to execute server-wide functions,
               -- such as killing processes, reloading cached data and
               -- shutting down the server to the user 'admin' when connected
               -- from the local machine. The password for the user is set
               -- to 'pass'.
GRANT SELECT, UPDATE(firstName, lastName) ON *
        TO joe@'%.server.com' IDENTIFIED BY 'mypass'
               -- If a current database is selected, this statement grants
               -- the ability to read data from any table in that database,
               -- as well as update any columns named 'firstName' or
               -- 'lastName' in those tables. If there is no current database,
```



```
                -- this statement grants those privileges on all tables in all
                -- databases in the server. Whichever is the case, the rights
                -- are granted to the user 'joe' when connected from any
                -- hostname that ends with '.server.com'. The password for the
                -- user 'joe' is set to 'mypass'.
GRANT USAGE ON *.* TO guest, dummy@"%.%" IDENTIFIED BY 'password'
                -- This grants only the ability to connect the server,
                -- no other functionality is allowed. This takes effect for
                -- the user 'guest' when connected from any location, as well as
                -- the user 'dummy' when connected from any location that
                -- contains a '.' (this eliminates the localhost). If the user
                -- 'guest' is being created, it is given a blank password. The
                -- password for the user 'dummy' is set to 'password'.
GRANT SELECT ON *.* TO joe@'10.0.0.0/255.0.0.0' -- Grant the ability to
                -- read any table in any database on the server to the user
                -- 'joe' that is connecting from any IP address starting with
                -- '10.' This is because the netmask 255.0.0.0 is applied to
                -- connecting address first, leaving only the first segment,
                -- which must match '10'.
```

REVOKE

The REVOKE statement has a structure virtually identical to GRANT.

```
REVOKE privilege [(columns)] [, privilege [(columns)] ...] ON table(s) FROM user [,
user_name ...]
```

Just as with GRANT, a 'what', 'where' and 'who' must be specified that details what privilege will be revoked from which user. All of the options and syntax of REVOKE is identical to GRANT with a couple of exceptions.

- To revoke the ability to grant privileges simple use 'GRANT' as the name of the privilege to revoke (this replaces the 'WITH GRANT OPTION' clause in the GRANT statement).
- The privilege 'ALL' actually refers to all privileges in this case, as opposed to GRANT, where 'ALL' really meant 'most of the privileges except for the really dangerous ones.' Using REVOKE with the ALL privilege will reduce a user to the level of the USAGE privilege. They will be able to connect to the server, but not perform any actions.
- There is no 'IDENTIFIED BY' clause within the 'TO' clause of REVOKE.

For each user given, the specified privileges will be immediately removed.

Examples

```
REVOKE ALL ON *.* FROM olduser -- Removes all privileges from the user 'olduser'
                        -- when connected from any location
REVOKE CREATE, DROP ON mydb.* FROM anotheruser@"%.server.com"
                        -- Removes the ability to create and drop tables in the
'mydb' database from the user
                        -- 'anotheruser' when connected from any location
                        -- ending in '.server.com' users names 'anotheruser'
                        -- connecting from any other location are not affected
                        -- by this statement.
REVOKE INSERT (phone) ON mydb.People FROM user@localhost
                        -- Removes the ability of the user to insert data
                        -- into the People table of the 'mydb' database that
```



```
                        -- contains data for the 'ohone' column. If the user
                        -- had INSERT privileges for any other columns of that
                        -- table, they can still insert new rows, as long
                        -- they do not give any data for the 'phone' column.
                        -- This affects the user 'user' when connected from the
                        --  same machine as the server.
REVOKE GRANT ON *.* FROM olduser -- Removes the ability to grant new
                        -- privileges from the user 'olduser' when connected
                        -- from any location.
```

Direct Interface

The SQL GRANT and REVOKE commands described above give very complete access to the MySQL security mechanism. However, sometimes it is necessary to fine tune the security settings by going directly to the security tables used internally by MySQL to store the policies. These tables are present in every MySQL server and are installed by default when the server is first set up.

While examining the GRANT SQL statement, we saw that MySQL privileges fall into four contexts: server-wide, database, table and column. Furthermore, a MySQL user contains information about both the username and the location of the user. All of this information is stored internally by MySQL in five tables within the 'mysql' database.

user - The 'main' privilege table and contains the username, user location and global privileges.

db - Database-level privileges for specific databases.

host - Location (hostname) level privileges for specific databases

tables_priv - Table level privileges for specific tables within databases

coumn_priv - Column level privileges for specific columns within tables

User

```
+-----------------+-----------------+------+-----+---------+-------+
| Field           | Type            | Null | Key | Default | Extra |
+-----------------+-----------------+------+-----+---------+-------+
| Host            | char(60) binary |      | PRI |         |       |
| User            | char(16) binary |      | PRI |         |       |
| Password        | char(16) binary |      |     |         |       |
| Select_priv     | enum('N','Y')   |      |     | N       |       |
| Insert_priv     | enum('N','Y')   |      |     | N       |       |
| Update_priv     | enum('N','Y')   |      |     | N       |       |
| Delete_priv     | enum('N','Y')   |      |     | N       |       |
| Create_priv     | enum('N','Y')   |      |     | N       |       |
| Drop_priv       | enum('N','Y')   |      |     | N       |       |
| Reload_priv     | enum('N','Y')   |      |     | N       |       |
| Shutdown_priv   | enum('N','Y')   |      |     | N       |       |
| Process_priv    | enum('N','Y')   |      |     | N       |       |
| File_priv       | enum('N','Y')   |      |     | N       |       |
| Grant_priv      | enum('N','Y')   |      |     | N       |       |
| References_priv | enum('N','Y')   |      |     | N       |       |
| Index_priv      | enum('N','Y')   |      |     | N       |       |
| Alter_priv      | enum('N','Y')   |      |     | N       |       |
```



```
+-----------------+-----------------+------+-----+---------+-------+
```

The primary key if the ‘user’ table is a joint key including both the Host field (the location of the user) and the User field (the username of the user). This means that users within MySQL are defined by from where they connect. The user ‘joe’ connecting from localhost is not the same user as the user ‘joe’ connecting from ‘my.server.com’. The Host field can contain SQL Wildcards (‘%’ and ‘_’) to indicate multiple hosts.

Also contained in the ‘user’ table is the Password field which is a scrambled version of the password of the user. It is important to note that this is not an encrypted version of the password, as in other security systems such as Unix logins. The password here is merely scrambled and the original password can be recovered from the scrambled version.

The other fields of this table are each of the possible privileges. A value of ‘Y’ or ‘N’ in these fields determine whether the user is granted that privilege or not.

db

```
+-----------------+-----------------+------+-----+---------+-------+
| Field           | Type            | Null | Key | Default | Extra |
+-----------------+-----------------+------+-----+---------+-------+
| Host            | char(60) binary |      | PRI |         |       |
| Db              | char(64) binary |      | PRI |         |       |
| User            | char(16) binary |      | PRI |         |       |
| Select_priv     | enum('N','Y')   |      |     | N       |       |
| Insert_priv     | enum('N','Y')   |      |     | N       |       |
| Update_priv     | enum('N','Y')   |      |     | N       |       |
| Delete_priv     | enum('N','Y')   |      |     | N       |       |
| Create_priv     | enum('N','Y')   |      |     | N       |       |
| Drop_priv       | enum('N','Y')   |      |     | N       |       |
| Grant_priv      | enum('N','Y')   |      |     | N       |       |
| References_priv | enum('N','Y')   |      |     | N       |       |
| Index_priv      | enum('N','Y')   |      |     | N       |       |
| Alter_priv      | enum('N','Y')   |      |     | N       |       |
+-----------------+-----------------+------+-----+---------+-------+
```

The primary key of the db table is a joint key containing the Host (the location of the user), the Db field (the database) and the User field (the username of the user). The Host and Db fields can contain SQL wildcards (‘%’ and ‘_’) to indicate multiple values.

The other field sof this table are each of the possible privileges applicable to databases. A value of ‘Y’ or ‘N’ in these fields determine whether the user is granted that privilege or not.

host

```
+-----------------+-----------------+------+-----+---------+-------+
| Field           | Type            | Null | Key | Default | Extra |
+-----------------+-----------------+------+-----+---------+-------+
| Host            | char(60) binary |      | PRI |         |       |
| Db              | char(64) binary |      | PRI |         |       |
| Select_priv     | enum('N','Y')   |      |     | N       |       |
| Insert_priv     | enum('N','Y')   |      |     | N       |       |
| Update_priv     | enum('N','Y')   |      |     | N       |       |
| Delete_priv     | enum('N','Y')   |      |     | N       |       |
| Create_priv     | enum('N','Y')   |      |     | N       |       |
```



```
| Drop_priv       | enum('N','Y')   |      |     | N       |       |
| Grant_priv      | enum('N','Y')   |      |     | N       |       |
| References_priv | enum('N','Y')   |      |     | N       |       |
| Index_priv      | enum('N','Y')   |      |     | N       |       |
| Alter_priv      | enum('N','Y')   |      |     | N       |       |
+-----------------+-----------------+------+-----+---------+-------+
```

The primary key of host table is a joint key containing both the Host field (the location of the user) and the Db field (the database). Both the Host and Db fields can contain SQL wildcards '%' and '_' to indicate a range of options.

The other field sof this table are each of the possible privileges applicable to databases. A value of 'Y' or 'N' in these fields determine whether the user is granted that privilege or not.

tables_priv

```
+-------------+---------------------------------+------+-----+---------+-------+
| Field       | Type                            | Null | Key | Default | Extra |
+-------------+---------------------------------+------+-----+---------+-------+
| Host        | char(60) binary                 |      | PRI |         |       |
| Db          | char(64) binary                 |      | PRI |         |       |
| User        | char(16) binary                 |      | PRI |         |       |
| Table_name  | char(60) binary                 |      | PRI |         |       |
| Grantor     | char(77)                        |      | MUL |         |       |
| Timestamp   | timestamp(14)                   | YES  |     | NULL    |       |
| Table_priv  | set(...)                        |      |     |         |       |
| Column_priv | set(...)                        |      |     |         |       |
+-------------+---------------------------------+------+-----+---------+-------+
```

The primary key of the 'tables_priv' table is a joint key containing the Host (location) field, Db (database) field, User (username) field and Table_name field. The Host and Db field can contain SQL wildcards ('%' and '_') to indicate multiple values. The Table_name field can contain the '*' wildcard to indicate every table within a database.

The table also contains a Grantor field that contains the name of the user that granted this particular privilege and a Timestamp that contains the time the privilege was created or last modified.

The final two columns of this table are 'Table_priv' and 'Column_priv'. The Table_priv column contains a set of privileges that are applicable to the table as a whole. The Column_priv column contains a set of privileges that are application to individual columns.

columns_priv

```
+-------------+---------------------------------+------+-----+---------+-------+
| Field       | Type                            | Null | Key | Default | Extra |
+-------------+---------------------------------+------+-----+---------+-------+
| Host        | char(60) binary                 |      | PRI |         |       |
| Db          | char(64) binary                 |      | PRI |         |       |
| User        | char(16) binary                 |      | PRI |         |       |
| Table_name  | char(64) binary                 |      | PRI |         |       |
| Column_name | char(64) binary                 |      | PRI |         |       |
| Timestamp   | timestamp(14)                   | YES  |     | NULL    |       |
| Column_priv | set(...)                        |      |     |         |       |
```



```
+-------------+--------------------------------+------+-----+---------+-------+
```

The primary key of the ‘columns_priv’ table is a joint key containing the Host (location) field, Db (database) field, User (username) field and Table_name field. The Host and Db field can contain SQL wildcards (‘%’ and ‘_’) to indicate multiple values. The Table_name field can contain the ‘*’ wildcard to indicate every table within a database.

The table also contains a Timestamp column that indicates the time the privilege was created or last modified. The Column_priv column contains a set of privileges that are application to individual columns.

These tables are consulted at two times during a session between the MySQL server and a client: during the initial connection and whenever a query is executed.

Initial Connection

Whenever a client attempts to connect to a MySQL server, the server consults the data in the ‘user’ table to determine whether is allowed to connect. The connecting user must have a location and username that matches an entry in that table.

You may recall that the values of the Host column can contain SQL wildcards. Given this, it is a distinct possibility that more than one row in the ‘user’ table may apply to a connecting user. The MySQL server will always use only one row to determine the access rights of a user. It decides which row to use by the following algorithm:

- More specific values for the ‘Host’ column are considered by less specific values. That is, host values that contain no wildcards are considered first, following by values that contain wildcards mixed with characters, with a singe ‘%’ that matches anything considered last.
- Rows that contain the same value for Host are considered by their ‘User’ value. Values of user that are not blank are considered before a blank User. Therefore a blank User is a ‘default’ that is used if no other user name matches.

Consider the following Host/User pairs:

```
root       localhost
joe        localhost
           localhost
joe        "%"
jan        "%.server.com"
mary       "%"
```

Using the basic principle of ‘most specific first’, various connection outcomes are possible.

- ‘root’ connecting from localhost - Matches the first line ‘root'/'localhost’ since both are specific.
- ‘joe’ connecting from localhost - Matches the second line ‘joe'/'localhost’ since both are specific.

- ‘jane’ connecting from localhost - Matches the third line ‘’/'localhost’ since the host is specific and there is no matching user
- ‘joe’ connecting from ‘my.server.com’ - Matches the fourth line ‘joe’/"%” since no specific host matches ‘my.server.com’ with a user ‘joe’, but the unspecific “%” has a user ‘joe’
- ‘mary’ connecting from localhost - Matches the third line ‘’/'localhost’ since the host is specific and there is no matching user. This is probably the most common mistake in MySQL access configuration. Intuitively, the line ‘mary/'%” would be used since it specifies the the username ‘mary’. However, Host is checked before User and a perfectly matching host (localhost) with no user takes precedence over an unspecific host (“%”) with a specific.
- ‘root’ connecting from ‘my.server.com’ - No line that matches ‘my.server.com’ has a user that matches ‘root’. Connection is denied.

If a user does not match any of the rows within the user table, the connection is rejected. If a match is made, the password is checked against the password supplied by the client. If the password matches the connection is allowed.

Query Execution

Once a client is allowed to connect to the MySQL database server, the next stage where security is checked is whenever the client attempts to execute a query or execute some function of the server. This stage of security involves all of the security tables.

The first stage of security checked here is the same ‘user’ table checked when the client first connected. Whichever row of this table was used to allow the user to connect also contains the ‘global’ rights for the user. Any privileges that are granted at this level apply to every database, table and column on the server. If the user has the necessary privilege at this level, the permission is granted and no further check is made.

If the global privileges did not grant sufficient permissions for the operation, the MySQL server then checks database-level privileges. The ‘db’ table is checked for the name and location of the user and the name of the database involved in the query. Just as in the case of the ‘user’ table, a ‘most specific first’ rule is used in the ‘db’ table. If a row of this table matches the username, host and database name, the privileges granted in that row are used.

If there is no match of username, host and database in the ‘db’ table, the server then looks for a row that matches the username and database but have a blank host column. If such a row exists, the server then moves the ‘host’ table, which can be considered an extension of the ‘db’ table. Within the host table, it is possible to have multiple rows for each database, by specifying different hosts. In this manner, it is possible to create rules that take effect for some locations but not for others.

The server looks to see if there is a row in the host table that has a matching location and database. If such a row is found, the user is granted any privileges that are both in this row and the corresponding row of the ‘db’ table. This is an important point, as a privilege that

is not in both places will not be granted. This allows rule to be set up where a privilege is granted to most people (using the row in the 'db' table with the blank host field) but selectively denied for certain locations (but including a row in the 'hosts' table that does not have that privilege).

If the 'db' and 'host' tables have been resulted in sufficient privileges, the query is executed. If the privilege is still not suffecient and the query is one that only effects the database, but not any tables (such as a DROP database query), then the query is forbidden. If the query does involve accessing a table, the server then moves to the 'tables_priv' table.

In the tables_priv table, the username, location, database and table name are all checked. As in the previous cases, a 'most specific first' rule is used when multiple rows match. If a matching row is found, the 'table_priv' column is checked to see if the required privileges exist. If so, the query is executed. If not, the 'column_priv' column is check to see if the required privileges are present. If they are also not there, the query is denied. If the privileges do exist in the 'column_priv' column, the server moves to the 'columns_priv' table for a final check.

When 'columns_priv' is used, each of the columns accessed in the query are checked. The username, location, database, table and column name must have a match in this table for each column used in the query. If the columns all have a match with the sufficient privileges the query is executed. If any of the columns do not have a match, or if any of the matches do not grant sufficient privileges, the query is denied.

Consider the following hypothetical excerpts from each of the privilege tables:

```
'user':
Host            User   Select_priv    Insert_priv
localhost       root       Y              Y
localhost                  N              N
"%"             joe        Y              N
localhost       joe        Y              Y
"%"             mary       Y              N
localhost       jane       N              N
"%.server.com"  john       N              N
"%"             jill       N              N

'db':
Host            Db         User        Select_priv   Insert_priv
localhost       mydb       mary            Y             N
"%.server.com"  "%"        joe             Y             Y
                mydb       john            Y             Y

'host':
Host            Db       Select_priv   Insert_priv
localhost       mydb        Y             Y
"%"             mydb        Y             N
"%"             "%"         N             N

Tabes_priv:
Host            User      Db     Table       Table_priv          Column_priv
"%.server.com"  john      mydb   People      'Select,Insert'
localhost       jim       mydb   People      'Select'            'Insert'
```



```
columns_priv:
Host            User      Db       Tables_name   Column_name   Column_priv
localhost       jim       mydb     People        firstName     'Insert'
localhost       jim       mydb     People        lastName      'Insert'
```

Given the above security information, lets look at the outcome of a couple of SQL queries from various users:

```
'SELECT * from mydb.People'
```

- 'root' connecting from 'localhost': Succeeds -- An exact match in the 'user' table gives 'root' global Select privileges.
- 'mary' connecting from 'my.server.com': Succeeds -- A match against the wildcard "%" for host and the exact username 'mary' in the 'user' tables gives global Select privileges to 'mary'
- 'mary' connecting from 'localhost': Succeeds -- In 'user' an exact match for 'localhost' and a default (blank) user, give 'mary' no permission to Select. Therefore 'db' is consulted next, where an exact match for 'localhost' and 'mary' for the database 'mydb' results in the Select privilege for 'mary'.
- 'john' connecting from 'my.server.com': Succeeds -- In user, a wildcard match for '%.server.com' an the exact username 'john' result in no permission to Select. The 'db' table is then consulted, where no match is found for the host, db and user. However. a match is found for the db 'mydb' and user 'john' with a blank host field. Therefore, the 'host' table is consulted. There, a wildcard matches the host, with an exact match for the db 'mydb', resulting in Select privilege for 'john'.
- 'jim' connecting from 'localhost': Succeeds -- In 'user', an exact match for the host 'localhost' and the default user results in no Select privilege, so we move to 'db'. There, no match is found for the db, user and host, or for the db, user and a blank host. Therefore the 'tables_priv' table is used next. There, an exact match is found for the host, user, database and table in question resulting in the Select privilege for 'jim'.
- 'jill' connecting from 'my.server.com': Fails -- In 'user', a match is found for the wilcard host and the exact username 'jill', resulting in no Select privilege. The 'db' table is used next, where no match is found for the db, user and host; or for the db, user and a blank host. The 'tables_priv' column is consulted, resulting again in no match for the db, user, host and table. Therefore, the 'N' privilege in the 'user' table for 'jill' stands and the query is not executed.

```
'INSERT INTO mydb.People (firstName, lastName) VALUES ('john', 'doe')
```

- 'root' connecting from 'localhost': Succeeds -- An exact match in the 'user' table gives 'root' global Insert privileges.
- 'mary' connecting from 'my.server.com': Fails -- A match against the wildcard '%' and the exact username 'mary' result in no Insert privilege for 'mary'. The 'db' table is then consulted where no match is found the the host, db and user (or for the db and user and a blank host). Finally, the 'tables_priv' table is used where again no match is found for the host, db, user and table. This causes the 'N' Insert privilege from the 'user' table to stand and the query is not executed.

- ‘mary’ connecting from ‘localhost’: Fails -- A match against the exact host ‘localhost’ and default user gives no Insert privilege. The ‘db’ table is used next, where an exact match against the host ‘localhost’ and the user ‘mary’ against results in no Insert privilege. The ‘tables_priv’ table is checked last where no match is found for the host, db, user and table. Therefore the ‘N’ Insert privilege from the ‘db’ table stands and the query is not executed.

- ‘john’ connecting from ‘my.server.com’: Succeeds -- A match against the wildcard host ‘%.server.com’ and the exact username gives no Insert privilege in the ‘user’ table. In the ‘db’ table, no match is found for the username, host and db, but a match is found for the username, db and a blank host, which causes the ‘host’ table to be used. There, a match again the wildcard host and the exact db gives no Insert privilege. The ‘tables_priv’ is checked next where a match against the wilcard host ‘%.server.com’, and an exact username, database and table result in the Insert privilege for ‘john’ and the query is executed.

- ‘jim’ connecting from ‘localhost’: Succeeds -- A match again the exact host and the default user gives no Insert privilege in the ‘user’ table. The ‘db’ table does not match the user, host and database (or the user, database and a blank host). Therefore the ‘tables_priv’ table is used, where an exact match for the username, host, database and table result in no table-wide Insert privilege, but rather a column-specific Insert privilege. This causes the ‘columns_priv’ table to be used. There, an exact match in for the username, host, database, table and both columns used in the query cause the permission to be given and the query is executed.

Server Security

While controlling access to data within MySQL is the probably the most important aspect of security with regards to MySQL, it is not the whole story. For as good as your MySQL security rules are, they can be bypassed if someone can gain access to the actual files that MySQL uses to store the data. Protecting these files, as well as protecting unwanted network access the MySQL server falls under the real of server security.

In this context, a ‘user’ is any operating system-level user on the same machine as the MySQL server. Anybody with network access to the MySQL server machine is also considered a user in this context.

The concept of ‘access’ in this context is the ability to read or write the operating system-level files used by MySQL. Also, any network connection to the MySQL server is considered access in this context.

The definition of ‘access’ we are using for server-side security involves two distinct concepts that we will deal with individually: operating-system security and network security

Operating System Secuity

The data within a MySQL database server is only as secure as the files that contain that data. The most well designed access-rights schema will mean nothing if a user can copy the database files to another machine with different access-rights where the data can be accessed.

Therefore, any files containing data used by MySQL must be inaccessible by a normal user. However, we cannot simply make all MySQL-related files forbidden. Most of the MySQL files, such as the executable binaries and configuration files must be readable by normal users for the normal operation of MySQL.

This creates a situation where a specific set of files must be protected, while other related files must be made accessible. This situation can be resolved through proper installation and configuration of MySQL:

- MySQL data files should be in a separate directory

Whatever directory schema is used for the rest of the MySQL installation, the data files themselves should be kept in their own directory. The default MySQL installation does this by creating a ‘var’ directory to store the database (or optionally using a system-wide /usr/var/mysql). This allows the data files to have a seperate security setting than the rest of the MySQL installation

- The MySQL server should run as a special user and group

Since any user who has access to the MySQL data files has access to the MySQL data, the MySQL server should be run as a special user, created just for MySQL. The default MySQL installation does this by creating a ‘mysql’ user and a ‘mysql’ group. This user and this group have full access to the MySQL data and should never be used for any other purpose than running the MySQL server.

- The permissions on the data directory should be properly set

If the previous two precautions have been taken, we have a special directory just for the MySQL data files and a special user and group just for running MySQL. The last step is to make sure only the special MySQL user has access to to the MySQL data directory. This is done by setting the proper permissions for the server operating system.

On a Unix system, the data directory and all of the files underneath it should be owned by the MySQL user and group. The ‘owner’ read, write and execute permission should be set on the data directory and the ‘owner’ read and write should be set on each of the data files. No other permissions should be allowed. The default MySQL installation uses these permissions.

On a Windows system it is possible to perform this same type of security set up using user and groups. However, if the Windows is using any other filesystem except for NTFS (e.g. FAT or FAT32), any user will still be able to read the data files. In addition, MySQL on

Windows currently does not create the users and groups automatically, as it does on Unix. If you are installing on Windows, you must manually set up this scheme yourself.

Once a MySQL server has been secured using these steps, a local user on the same machine as the server will not be able to access the MySQL data files.

Network Security

If your MySQL server is intended only for use by local clients, network security is easy. However, most MySQL servers are used for applications that require network access. When this is the case it is important to take the proper steps to ensure network safety.

There are three main dangers to a MySQL server that involve network security. The first is an attack that would directly compromise the MySQL server itself. The second is an attack that would allow an unauthorized user to access the MySQL server. The third is an attack that would prevent the MySQL server from performing its duties.

Direct Compromise

The first form of attack is the most dangerous, but the least likely. This type of attack would grant the intruder complete control over the MySQL server process itself. Most commonly in these type of attacks, the intruder uses the process to gain further entry into the server machine, ultimately leading to root access. For this type of attack to succeed there would have to be some sort of flaw with either the MySQL network protocol or the MySQL server code itself.

The MySQL network protocol is an open protocol that can be examined by any interested party. Although the protocol currently has very little documentation, the source code is easy to follow. In addition, there have been several widely-used applications that use the protocol directly (mainly language bindings such as the MySQL JDBC driver). If there were some backdoor or flaw in the protocol, it would be very easy to spot.

An attack exploiting a bug in the MySQL server code is slightly more possible, but still very unlikely. These attacks usually take advantage of poor C programming practices that lead to the possibility of buffer overflows, allowing the attacker to execute arbitrary code. While MySQL has never undergone an official public code audit, in several years of popular use it has not gained a history of vulnerability. This does not at all mean that there aren't undiscovered exploits in the code, but it does speak well for the quality of the code. And of course, like any open source project, the source code of the MySQL server is freely available for scrutiny.

In all, the possibly of a direct compromise of the MySQL server is very slim. However, the difference between slim and none can mean the safety of your data. Any available precautions against this type of attack should always be taken. For instance, the 'libsafe' package provides system-wide protections against buffer overflow attacks. Also never run the MySQL server as the 'root' user. Should the server become compromised the damage done by the attacker can by greatly limited if they are stuck in a restricted account, such as a special user created just to run MySQL.

Unauthorized Access

This is probably the most real and common danger to a MySQL server system. An attacker, eavesdropping on the network traffic on the server machine's network, can intercept the authentication information used by MySQL clients to connect to the server. The attacker can then create their own connection, and gain access as an authorized user.

By default MySQL performs all network communication using unencrypted data. Any one monitoring the network traffic can watch the entire conversation between the MySQL server and a client. Since the MySQL protocol is open, it is simple to decode the conversation and extract information such as the authentication data used by the client, or perhaps sensitive data straight from a query result.

One way to minimize this problem is to access MySQL over a network as little as possible. While that may seem like a useless solution, it is quite common with modern applications. With the advent of the Web application, it is common to have a situation where the MySQL 'client' is actually a Web application accessed through a Web server. In this scenario, it is usually desirable to use local (Unix socket) communication between the 'client' Web application and the MySQL server (assuming they are on the same machine). The outside world communicates with the application via the Web server, leaving the MySQL server securely in a non-networked environment.

When you have to use a network connection, the next best thing is to limit your network visibility by using a firewall or similar setup. There is no reason to expose the MySQL server to the Internet as a whole unless you have to.

Even if you do have make your MySQL server visible on an open network (like the Internet), all hope is not lost. Recent versions of MySQL have introduced the ability to use SSL encrypted communications between the MySQL server and clients. To use this feature, the SSL libraries must exist on both the MySQL server and client machines, before MySQL is installed.

It is always a good idea to configure a MySQL server to use SSL if it is available on your machine. Once a MySQL server has been equipped with SSL, it will automatically attempt to use SSL whenever contacted by a client. Only if the client is incapable of SSL does the server fall back to plain unencrypted communication. For more information about configuration the MySQL server to use SSL, see Chapter XX: Installation.

Denial of Service

The final type of network attack on a MySQL server is the most insidious. Denial of Service attacks have gained popularity among the cracker community in recent years because of its difficulty to defeat. In a denial of service attack, the attacker floods the server machine with network data. If the attacker can send data faster than the server can respond to it, network traffic to the server will come to a halt as it tries to process the data. The problem with this type of attack is that it is extremely difficult to defeat. As long as an attacker can communicate with the server machie, they can bombard the server with data in order of cause a denial of service.

The only real way to prevent a denial of service attack is to eliminate an attackers ability to contact to the server machine. The easiest way to do this is to limit network access to the MySQL server to only those clients that need it. If all of the users directly connecting to the MySQL server are on the same network, it is possible to configure a firewall to deny access to the MySQL server to anyone else.

Likewise, as mentioned earlier, if the only client access to MySQL is through a Web application on the same server machine, it is possible to disable MySQL network available completely, since all remote users will only be connecting directly to the Web server.

Client Security

The final area of security we will consider does not directly impact the MySQL server but is nonetheless important. This is this the realm of client security. For as good as your internal security is, if an authorized client can be hijacked, they will be able to access the MySQL server will all of the rights of that client, and the server will have no way of knowing the difference.

In the realm of the client, a 'user' is the user of the client program. That is, any operating system-level user on the same machine as the client.

The concept of 'access' for a client is the ability to execute the client program. Unless the client program performs some sort of authentication whenever it is run (such as prompting for a password), anyone who executes the client has full access to all of its abilities.

Client security is often overlooked when dealing with these matters, usually because the client is not necessarily on the same machine as the server. It is tempting to think that if your server is locked down then the possibility of malicious activity is eliminated. However, if malicious users gain access to secure clients, they also gain access to the server.

This becomes an even greater problem when considering clients on multi-use machines. For example, consider a Web server that is used by multiple sites. Depending on the setup of the server, a web application that accesses a database may provide it's authentication information to any other user on the server.

To better understand this problem, it is necessary to review how MySQL clients communicate with the server. To do this, we will consider two scenarios: a client on a single-user machine and a client on a multi-use machine separate from the MySQL server.

Scenario 1

Single-user machines are the easiest to secure, but should still not overlooked. The recent emergence of Web-based attacks such as the ILOVEU trojan can expose a machine that

has no servers to attack. In addition, there is always the possibility of someone gaining physical access to your personal machine. Who doesn't step away occasionally without activating a screen lock?

The danger posed by an attacker gaining access to a single-user machine is that they could then use the client to access the MySQL server with the same rights as the actual user. For this to happen, the MySQL client has to have some knowledge of the users authentication (that is, the username and password). There are four ways a MySQL client can obtain the authentication information for a user:

- The client client asks for the authentication
- The client uses a configuration file for authentication
- The client has the authentication information hard-coded
- The client uses the default authentication

Prompting

In this method, the client presents a prompt to the user to enter a username and password. This method is the most secure out of all of the possibilities because the authentication information is not stored permanently anywhere on the machine. An attacker who gains access to the machine would have to know the username and password in order to use the client to access the MySQL server. This makes the machine useless to the attacker, since they could use any machine as the client if they knew the username and password before hand.

While this is the most secure method of running a MySQL client, it is often inconvenient. In fact, it can be argued that this method actually leads to decreased security for some users. The argument is that if a user is forced to type in their passwords for something they use infrequently, they are likely to forget the passwords. Therefore, in order to make remembering the passwords easier, the user will tend to pick more insecure passwords.

Whether or not this is true in practice, it is undeniable that having to type in the username and password repeated for a commonly used application can be tedious. The main factor to consider when evaluating this option is the nature of the user. If they can put up with entering the username and password whenever they want to access the client, this is definitely a secure way to go.

Configuration File

In this method, the client reads a configuration file that contains the username and password. This configuration file could be a standard MySQL configuration file or a configuration file specific to the client. The MySQL client libraries look for configuration files in the following places:

`/etc/my.cnf` (Unix)

`$HOME/.my.cnf` (Unix, where *`$HOME`* is the home directory of the user)

`%WINDOWS%\my.ini` (Windows, where *`%WINDOWS%`* is the system Windows directory)

`C:\my.cnf` (Windows)

In both cases (Unix and Windows), the second listed configuration file takes precedence over the first. And any client-specific configuration file will take precedence over either.

The most important thing to remember when using a configuration file for storing authentication information is that the information is stored in the configuration file as plain text. Anyone who can read the configuration file can read the authentication data. On a single-user machine, this generally means that anyone with access to the machine, locally or remotely, will have full access to this information. Since a single-user machine is meant for only one person's use, this is usually an acceptible risk. However it is still a risk. It should be assumed that anyone who has accessed the machine knows the MySQL username and password used by the client.

If you are writing a MySQL client, you may be deciding whether use the default MySQL configuration files or your own for accessing authentication information. The advantage to using the MySQL configuration files is that you do not have to do anything. If you leave the connection information blank the client libraries will automatically use any username and password given in the standard configuration files. This makes programming the client easy, while still giving the user the option of configuring their username and password.

There are no direct advantages to using a custom configuration file. However, if your application already has a configuration file, including the MySQL options in it will let your users configure the entire application in one file. In addition, using a custom configuration file provides a weak form of 'security by obscurity'. If an attacker were looking for the MySQL authentication data, he or she would sure look in all of the standard configuration files. However, the attacker may not know about the configuration file for your application, thus protecting the data. Like any form of security by obscurity, this should not be the basis for your security plan, but rather a small 'bonus' received when using a custom configuration file.

Hard Coding

This method involves encoding the username and password directly into the application. For this to happen, the application has to be custom written (or at least, custom-compiled) for the client machine. This is rarely done for single-user machines, so this method is hardly used. However, it does remain and option.

As far as security goes this method falls between the security of prompting the user and the openness of using configuration files. It is tempting to think that this method is just as secure as prompting because the authentication data is hidden in the client binary and not visible to any user of the system. However, this is not generally the case.

With most programming languages, text strings are stored within a binary program as a plain text string. Therefore, using methods such as the Unix string program, it is possible to examine the binary and read the authentication information. Even worse is the case where the client is written in a non-precompiled language such as Perl or Python. In this case, the username and password would be visible directly within the script file in plain text.

Default

In this method, the client does not make any attempt at all to find out the authentication information for the user. Instead it simply uses the default username and password used by the MySQL client library. On Unix, this default username is the Unix-username that owns the client process. The default password is bank.

As far as pure client-security is concerned this method is just as secure as prompting. No authentication information is stored permanently on the machine, rendering it useless to an attacker. However, for this method to be useful, the MySQL access rights have to be configured to allow a user to connect with no password.

Therefore, while this method is secure from a client perspective, it requires a compromise of the MySQL access rights that is unacceptable in most instances.

SSL

So far, when considering our single-user machine the types of attacks we have considered have involved direct access to the machine in order to execute the client. However, this is not the only type of attack that could comprise the MySQL authentication data.

If a malicious user were monitoring the network traffic coming from the machine, they would be able to eavesdrop on the communication between the machine and the MySQL server. With this access they may be able to obtain the authentication information used to connect to the server.

As mentioned in the section on server security, MySQL transmits authentication information by default as scrambled plaintext. Therefore, the actual username and password can be extracted (with some work). SSL connections should be used whenever available, as they encrypt the traffic between the client and the server, eliminating the effectiveness of eavesdropping attacks.

Scenario 2

While multi-user computers used to be the rule, the desktop computing explosion of the 1980‘s and '90‘s has made them the exception for workstations and home computers. However, for server applications, multi-user computers are still very common. There are two common ways in which a MySQL client is used within a multi-user machine:

- Most obviously, users of a multi-user machine can connect to the machine and directly execute MySQL clients. For instance, a user of a Unix server can log in and run a MySQL client on the command line.

- More commonly, but perhaps less obvious, is the case where users do not directly connect to a server, but rather run MySQL clients remotely (or automatically). For example, consider a Web server that hosts multiple sites. If these sites include database-driver applications, each Web application acts as a MySQL client when access the database. Therefore, a user's application acts as a proxy for the user, creating the same concerns that exist when users directly connect.

The major concern when dealing with a multi-user machine is that one user could access the authentication information for another user. Since MySQL uses the same methods to retrieve authentication information on a multi-user machine as on a single-user machine, we have the same four possibilities to consider. However, the implications of some of these possibilities are different in a mutli-user environment:

Prompting

Just like in the single-user scenario, prompting the user for the username and password is the most secure method of securing that information. However, there are a couple of additional considerations to be aware of when in a multi-user environment.

Firstly, it may be somewhat easier for an attacker to intercept the username and password of the user while they are typing it in. On a single-user machine, the attacker must be able to intercept the network traffic remotely. In the case of a multi-user machine, the attacker has much easier access to the network traffic, and could possibly even intercept the keystrokes of the user locally without even looking at the network traffic.

Secondly, there is always the possibility that a malicious user replaces the MySQL client with a trojan copy that works just like the real client, except that it also captures the username and password.

While both of these possibilities exist, they involve the concentrated effort of a malicious attacker. Prompting is still the surest way to keep authentication safe from casual snoopers. In addition, there are several steps you can take to minimize the effectiveness of the above attacks:

- As mentioned in in the first scenario, use SSL connections to MySQL whenever possible. This eliminates the danger of network snooping as the authentication information will not be decipherable within the encrypted data.

- Also use encryption (such as SSH) when connection to the multi-user machine remotely. Even if your client is secure, if you are using plain telnet to connect to the machine, an attacker can read your username and password as you type it in.

- Make sure the device permissions are properly set on the machine. While you may not have control over this if you are not the systems administrator of the machine, it is important to check to see of the terminal devices can be read by any user or only

the owner. If any user can read the terminal device, a user logged in at the same time as you can read your keystrokes as you type them.

- Make sure your path is secure. A common method of tricking users into running trojan programs is to put a program with the same name as the real client in a commonly used directory, such as '/tmp'. If your path checks the current directory first, it will run the trojan instead of the real client whenever you are in '/tmp'. The current directory should always be the last entry in the path, if it's there at all.
- Make sure the client remains untouched. The systems administrator of the machine should keep a list of checksums for all of the executable programs on the machine. This list should be periodically checked with the actual binaries to make sure they haven't changed. Again, if you don't have system administrator access to the machine, the best you can do is bug the admin to make sure this happens.

The method of prompting only applies to the first type of user described above, that is, a user directly connecting to the machine (such as via SSH). Users running applications on a shared server (such as a Web server) cannot use prompting, naturally, since they are not connected to the machine.

Configuration File

Storing the authentication information within a configuration file has the same benefits as in the case of the single-user machine and fewer drawbacks. On a single-user machine, there is always the threat that someone could access the machine and look at the configuration files.

For a multi-user machine this is actually less of a problem if proper precautions are taken. Since the MySQL client will be executed by a logged-in user of the machine, it is possible to set the permissions of the configuration file so that only that user can read it. Any other user of the machine will not be able to access the configuration file. Therefore, to get the authentication information, an attacker would have to be able to become the logged-in user. At this point, the problem becomes one of system security and not one of MySQL security.

This makes properly secured configuration files a very good choice for multi-user machines. However, this becomes tricky when considering the two different types of multi-user access mentioned above.

For users directly accessing the machine (such as via SSH), everything is fine. Each user can have their own configuration file, which is only readable by that user. For users running applications, especially Web applications, on a shared server, however, the situation becomes tricky.

Most Web servers run applications (such as CGI programs) as the same user that owns the web server process itself. That is, if the web server is running as user 'www', and a CGI program is owned by user 'joebob', the web server will run the program as user 'www'.

Therefore, any configuration file used by the Web application must be readable by ‘www’.

This creates a problem if applications belonging to more than one user are on the Web server. If each user has to make their configuration file readable by ‘www’, then any user’s web application can read any other users configuration file. A malicious user can create a web application which reads other users’ configuration files and retrieves the usernames and passwords.

One solution to this is to configure the web server to execute web applications as the user who owns the executable, rather than the user that owns the web server process. In other words, using our previous example, the web server would execute a web application belonging to the user ‘joebob’ as the user ‘joebob'.

This brings back the per-user security of the applications, and allows them to read configuration files that are private to each user. However, it also opens up security risks on the server. If a user were to obtain access to the server via a flaw in a web application, they would gain access as the user who owned the web application. In the previous model where all applications ran as the web server user, the attacker would only gain access as that user, which should have limited access to the rest of the machine.

Besides the security implications of running the web applications as the individual user, this method simply doesn’t work well with some applications. Environments that share all of the applications in a single process space (such as Java servlets and Apache mod_perl) must use the same user for each application (because only one actual process is being used). In these cases, every web application will always have access to the configuration files of any other web application.

Hard Coding

Hard coding the authentication information into the application has the same implications with multi-user servers as it does with single-user machines. These implications become even more apparent in a multi-user environment. Unless the application binary is somehow encrypted (which is not very practical with an executable binary), any user on the machine will be able to examine the binary and extract the username and password.

Therefore, while this method will keep out casual snoopers, it should not be used in a multi-user environment where any of the other users are not trusted.

Default

Using the default MySQL authentication provides the same benefits and problems in a multi-user environment as on a single-user machine. It creates great client security, as the username and password are not stored anywhere on the machine, but it requires that the user have a blank password, eliminating the usefulness of MySQL authentication.

Final Note

A section on data security cannot be considered complete with a mention of the two most common attackers of all: user error and natural disaster. Many times more data is lost through equipment failure or software design error than is lost via malicious intruders. Fortunately, protecting against these types of incidents is much more straightforward than the steps needed to protect against a determined human attacker. Please read Chapter XX: Administration and make sure your MySQL data is frequently backed up, and replicated if necessary. Accidents always happen.

